A soros illesztőprogram-réteg

Greg ismerteti az új soros illesztőprogram-réteg felületét, illetve az új soros illesztőprogramok és az egyedi soros kapuk bejegyzésének módját.

Előző két cikkemben (Linuxvilág 2002. szeptember és november) a tty-rétegről volt szó, illetve egy alapvető szolgáltatásokat nyújtó illesztőprogram készítését is ismertettem. Szó esett néhány `ioctl` függvényről, illetve a `termios` adatszerkezet értelmezéséről. Ezek a cikkek nagyszerű kiindulópontot adnak, ha beágyazott rendszeredhez új tty jellegű eszközt kell létrehoznod – például egy soros kaput. Minden új rendszer megalkotásakor az eszköz tervezője szereti valamilyen más címre helyezni a soros kaput, illetve hajlamos más UART használata mellett dönteni – sokszor pedig egyszerűen elmarad a soros kapu, és USB váltja fel. A legtöbb fejlesztő tehát teljesen új tty illesztőprogramot kénytelen az új eszközhöz készíteni, ha Linuxszal is rendesen akarja használni. Szerencsére a tty-réteg felett további rétegek helyezkednek el, amelyek segítenek elfedni bonyolultságát, illetve olyan lehetőségeket biztosítanak a fejlesztő számára, amelyek egyrészt szükségesek a soros illesztőprogramhoz, másrész jobban illeszkednek az UART vagy az USB modellhez. Ezek a rétegek a soros és az USB-soros illesztőrétegek. Ebben a cikkben a soros illesztőrétegről lesz szó, míg az USB-soros réteg egy későbbi írás témáját szolgáltatja majd.

Soros őrület

Ha megnézed az általános PC soros illesztőprogram kódját a 2.2-es vagy a 2.4-es rendszermagban (a ***drivers/char/serial.c*** fájl), egy meglehetősen összetett kódot találsz, tele `#ifdef` sorokkal, amelyek az eszköz típusától függően jutnak szerephez. Az állományt számtalanszor másolták, így kapott támogatást jó néhány olyan eszköz, amelyik nem illeszkedik az általános PC UART-eszközök sorába (ilyen például a ***serial_amba.c*** és a ***serial_21285.c*** illesztőprogram). Szerencsére néhány fejlesztő – az ARM Linux karbantartója, ***Russell King*** vezetésével – átdolgozta a soros illesztőprogramot, ami immár egy általános soros magból, és kisebb, az adott eszközökhöz igazodó illesztőprogramokból áll. Kódjuk a fő rendszermagfában valahol a 2.5.28-as változat környékén jelent meg. A cikkben szereplő példák a 2.5.35-ös változatból származnak, így nem árt ellenőrizni, hogy az általad használt rendszermagban mely dolgok változtak meg.

Soros illesztőprogram bejegyzése

A soros réteg két dolgot vár el az illesztőprogramtól: jegyezze be önmagát a soros magnál, majd jegyezze be a rendszerben a PCI-számbavétel (PCI enumeration) vagy egyéb eszközfelsorolási módszer révén megtalálható soros kapukat. Az illesztőprogram bejegyzéséhez az `uart_register_driver()` függvényt kell meghívni egy az `uart_driver` nevű adatszerkezetre irányított mutatóval. A függvény kiveszi az `uart_driver` adatszerkezetben található adatokat, és ennek alapján elvégzi a tty-réteg beállítását.

A soros illesztőprogram által az `uart_driver` adatszerkezetben keresett adatmezők a következők (*lásd a következő oldalon*):

```
struct uart_port {
  spinlock_t        lock;       /* a kapu zÆrolÆsÆra */
  unsigned int      iobase;     /* be/ki[bwl] */
  char              *membase;   /* olvasÆs/ rÆs[bwl] */
  unsigned int      irq;        /* irq szÆma */
  unsigned int      uartclk;    /* alap uart  rajel */
  unsigned char     fifosize;   /* k ldØs fifo mØrete */
  unsigned char     x_char;     /* xon/xoff karakter */
  unsigned char     regshift;   /* regiszthereltolÆs */
  unsigned char     iotype;     /* be- Øs kiviteli hozzÆfØrØs
                                   jellege */
  unsigned int      read_status_mask;
                                /* illesztıprogramt l f ggı */
  unsigned int      ignore_status_mask;
                                /* illesztıprogramt l f ggı */
  struct uart_info  *info;
                               /* mutat  a sz lı adataira */
  struct uart_icount icount;   /* statisztikÆk */
  struct console    *cons;
                    /* konzol-adatszerkezet, ha sz ksØges */
#ifdef CONFIG_SERIAL_CORE_CONSOLE
  unsigned long sysrq;          /* sysrq idıhatÆr */
#endif
  unsigned int      flags;
  unsigned int      mctrl;
                    /* pillanatnyi modemvezØrlı-beÆll tÆsok */
  unsigned int      timeout;
                    /* karakteralapœ idıhatÆr */
  unsigned int      type;       /* kapu t pusa */
  struct uart_ops   *ops;
  unsigned int      line;       /* kapuindex */
  unsigned long     mapbase;    /* be- Øs kiviteli œjra-
                                   hozzÆrendelØs cØljÆb l */
  unsigned char     hub6;
                    /* a 8250 illesztıprogramban kell
                       szerepelnie */
  unsigned char unused[3];
    };
```

```
struct uart_ops {
   unsigned int     (*tx_empty)(struct
                                uart_port *);
   void    (*set_mctrl)(struct uart_port *,
                        unsigned int mctrl);
   unsigned int     (*get_mctrl)(struct
                                uart_port *);
   void    (*stop_tx)(struct uart_port *,
                      unsigned int tty_stop);
   void    (*start_tx)(struct uart_port *,
                       unsigned int tty_start);
   void    (*send_xchar)(struct uart_port *,
                         char ch);
   void    (*stop_rx)(struct uart_port *);
   void    (*enable_ms)(struct uart_port *);
   void    (*break_ctl)(struct uart_port *,
                        int ctl);
   int     (*startup)(struct uart_port *);
   void    (*shutdown)(struct uart_port *);
   void    (*change_speed)(struct uart_port *,
                           unsigned int cflag,
                           unsigned int iflag,
                           unsigned int quot);
   void         (*pm)(struct uart_port *,
                      unsigned int state,
                      unsigned int oldstate);
   int         (*set_wake)(struct uart_port *,
                           unsigned int state);

/* A kapu t pusÆt visszaad  karakterlÆnc.*/
   const char *(*type)(struct uart_port *);

   /*A kapu Æltal hasznÆlt be- Øs kiviteli
    *mem ria-erıforrÆsok elengedØse.
    *Sz ksØg szerint a be- Øs kiviteli c mek
    *felszabad tÆsÆt is magÆban foglalja.*/
   void    (*release_port)(struct uart_port *);

   /*A kapu Æltal hasznÆlt be- Øs kiviteli
    *mem ria-erıforrÆsok lefoglalÆsa. RØsze
    *a be- Øs kiviteli c mek lefoglalÆsa is,
    *ha sz ksØges.*/
   int     (*request_port)(struct uart_port *);
   void    (*config_port)(struct uart_port *,
                          int);
   int     (*verify_port)(struct uart_port *,
                          struct serial_struct *);
   int     (*ioctl)(struct uart_port *, unsigned
                    int, unsigned long);
};
```

```
struct module   *owner;
const char      *driver_name;
const char      *dev_name;
int              major;
int              minor;
int              nr;
struct console  *cons;
```

Az ***owner*** (tulajdonos) mező egy mutató a soros illesztőprogramot birtokló modulra. Általában a `THIS_MODULE` makróra mutat.

A ***driver_name*** mező az illesztőprogram leírását tartalmazó karakterláncra mutat, ennek tartalma általában egyezik a ***dev_name*** mezőével. A ***dev_name*** mező megadásakor természetesen a `devfs` is figyelembe veendő, és a `devfs` választása esetén `%d` karaktereket kell hozzáadni a mező végéhez. Ennek oka a `devfs` eszközcsomópont létrehozási módszerében keresendő. Például az ***amba.c*** illesztőprogram az alábbi ***driver_name*** és ***dev_name*** mezőket használja:

```
   .driver_name          = "ttyAM",
#ifdef CONFIG_DEVFS_FS
   .dev_name             = "ttyAM%d",
#else
   .dev_name             = "ttyAM",
#endif
```

A major (fő) és a minor (mellék) mezők az illesztőprogram fő- és melléksorszámát tartalmazzák.

Az ***nr*** mező adja meg, hogy az illesztőprogram legfeljebb hány soros kaput támogat.

A ***cons*** mező egy mutató, a `console` adatszerkezetre mutat, és csak akkor jut szerephez, ha az illesztőprogram támogatja a soros konzol használatát. Ha az illesztőprogram nem támogat a soros konzolt, a mezőnek `NULL` értéket kell adni.

Soros kapu bejegyzése

Most, hogy a soros illesztőprogramot a soros illesztőprogram-rétegnél sikeresen bejegyezted, a soros kapukat is egyenként be kell jegyezni egy-egy `uart_add_one_port()` hívással. A függvény lefoglal egy mutatót az eredeti `uart_driver` adatszerkezetre, amelyet az `uart_register_driver` függvénynek adtál át, illetve egy másikat az `uart_port` adatszerkezetre.

Az `uart_port` adatszerkezetet az *1. lista* tartalmazza. Ezeknek az adatmezőknek a nagy részét működés közben az egyedi illesztőprogramok használják annak meghatározására, hogy az adott kapu hogyan csatlakozik a processzorhoz (`hub6`, `iobase`, `membase`, `mapbase` és `iotype` változók).

Az adatszerkezet egyik legérdekesebb változója az `uart_ops` adatszerkezet-mutató, amely a soros mag által a kapuhoz egyedileg készített illesztőprogram meghívására használt függvények listáját adja meg. Az adatszerkezet felépítését a *2. listában* láthatjuk.

Meglehetősen nagyméretű adatszerkezetről van szó, számos függvénymutatóval – legalább olyan ronda, mint a `tty_driver` adatszerkezet.

A `startup` (indítás) függvény lefuttatása minden egyes `open(2)` (megnyitás) hívásnál megtörténik. Futtatására csak az után kerül sor, hogy a soros mag elvégezte a számos erőforrás-ellenőrző művelet mindegyikét, és úgy találta, hogy a kaput valóban meg kell nyitni. A soros illesztőprogram általában megad bizonyos eszközfüggő beállításokat, hogy a kaput a függvényben használni lehessen.

A `shutdown` (leállás) függvény a `startup` ellentettje. Meghívására akkor kerül sor, ha a kapu lezárult, és több adat nem halad át rajta. Akkor történik ilyen, ha az eszközt leállítottuk, és a `startup` függvény lefutása során lefoglalt erőforrásokat fel kell szabadítani.

A `request_port` és a `release_port` függvények a soros kapu használatához szükséges memória és egyéb erőforrások lefoglalására szolgálnak. A `config_port` függvény nagyon hasonlít a `request_port` függvényre, ám akkor hívódik meg, amikor a gép végigpróbálja (autoprobe) a csatlakozó soros kapuk mindegyikét; illetve ugyanazokat az erőforrás-foglalásokat végzi el, mint a `request_port`.

A `change_speed` függvény használható a kapu vonali beállításainak módosítására. A függvénynek átadott értékek már

letisztultak az eredeti, a tty-rétegen keresztül átadott termios adatszerkezethez képest, így a soros illesztőprogram működése egyszerűbb lehet.
A vonali és kapuállapot lekérdezésére és módosítására számos függvény használható:

- set_mctrl: új értéket ad az MCR UART regiszternek.
- get_mctrl: lekérdezi az MCR UART regiszter pillanatnyi értékét.
- stop_tx: leállítja az adatküldést a kapun.
- start_tx: elindítja az adatküldést a kapun.
- tx_empty: megadja, hogy a kapu küldő része üres-e vagy sem.
- send_xchar: ***XOFF*** karakter küldésére utasítja a kaput a gép felé.
- stop_rx: leállítja az adatok fogadását.
- break_ctl: a BREAK értéket küldi át a kapun.
- enable_ms: engedélyezi a modemállapot-megszakításokat.

Két, a soros kapu energiakezelésével kapcsolatos függvény létezik: a pm és a set_wake. Ha a számítógép támogatja az energiagazdálkodási lehetőségeket, ezekkel a függvényekkel lehet kezelni az eszköz indítását és leállítását.
A verify_port használható a neki átadott értékek ellenőrzésére a tekintetben, hogy azok használhatók-e a megadott kapuhoz. Meghívása akkor történik, amikor a felhasználó TIOCSSERIAL ioctl(2) hívást végez a kapun. (A TIOCSSERIAL szerepével kapcsolatban lásd: „A tty-réteg 2. rész”, Linuxvilág 2002. november.)
A soros illesztőréteg számos általános soros ioctl kezelésére képes, úgymint TIOCMGET, TIOCMBIS, TIOCMBIC, TIOCMSET, TIOCGSERIAL, TIOCSSERIAL, TIOCSERCONFIG, TIOCSERGETLSR, TIOCMIWAIT, TIOCGICOUNT, TIOCSERGWILD és TIOCSERSWILD. Bármely más ioctl-nek a soros kapura történő meghívása esetén az uart_ops adatszerkezet ioctl hívása segítségével átadás történik az adott kapura.
Az utolsó függvény a type, ami a soros kapu leírását megadó karakterlánccal tér vissza. Ezt a */proc/tty/driver/* könyvtárban elérhető *proc* állomány használja, illetve a rendszerindító üzenetek között jelenik meg a kapu térképezés.

Hol vannak az adatok?

Bizonyára feltűnt, hogy az uart_ops adatszerkezet semmilyen függvényt nem tartalmaz adatok fogadására vagy küldésére. A felhasználó által write(2) hívással a tty-rétegnek átadott adatokat a soros réteg egy körkörös átmeneti tárba helyezi, ezt követően az egyedi uart illesztőprogram feladata, hogy kivegye és átküldje őket a kapun. Az illesztőprogram általában minden UART megszakításnál ellenőrzi a körkörös átmeneti tár tartalmát, hátha talál benne továbbításra váró adatokat. Az alábbi példafüggvény erre kínál egy lehetséges megoldást:

```
static void tiny_tx_chars(struct uart_port
*port)
{
     struct circ_buf *xmit =
                    &port->info->xmit;
     int count;

     if (port->x_char) {
          /* itt t rtØnik a port->x_char
             k ldØse a kapura */
          UART_SEND_DATA(port->x_char);
          port->icount.tx++;
          port->x_char = 0;
          return;
     }
     if (uart_circ_empty(xmit) ||
        uart_tx_stopped(port)) {
          tiny_stop_tx(port, 0);
          return;
     }
     count = port->fifosize >> 1;
     do {
          /* send xmit->buf[xmit->tail]
             kik ldØs a kapura */
          UART_SEND_DATA
           (xmit->buf[xmit->tail]);
          xmit->tail = (xmit->tail + 1) &
                         (UART_XMIT_SIZE - 1);
          port->icount.tx++;
          if (uart_circ_empty(xmit))
               break;
     } while (--count > 0);

     if (uart_circ_chars_pending(xmit)
                    <WAKEUP_CHARS)
          uart_event(port, EVT_WRITE_WAKEUP);

     if (uart_circ_empty(xmit))
          tiny_stop_tx(port, 0);
}
```

A függvény azzal indul, hogy ellenőrzi, az x_char* éppen kiküldésre vár-e. Ha igen, akkor elküldi, majd megnöveli a kapun átküldött karakterek számlálójának az értékét. Ha nem, a körkörös átmeneti tár tartalmát ellenőrzi, hogy talál-e benne adatot, illetve azt, hogy éppen nem állította-e le valami a kaput. Ha a feltételek teljesülnek, megkezdjük a karakterek kivételét a körkörös átmeneti tárból, és átküldjük őket az UART-nak. A példában legfeljebb a FIFO méretének a felét kitevő adatmennyiséget küldünk el, ami nagy átlagban megfelelő mennyiségnek mondható. A karakterek elküldése után láthatjuk, hogy elegendő karaktert vettünk-e ki a körkörös átmeneti tárból, és kérhetünk-e újabbakat. Ha igen, az EVT_WRITE_WAKEUP átadott értékkel meghívjuk az uart_event() függvényt, ezzel felkérjük a soros magot arra, hogy értesítse a felhasználói réteget a további adatok elküldésének lehetőségéről.
Minden, a soros illesztőprogram által fogadott adat a tty-réteghez kerül, pontosan úgy, mintha egy általános tty illesztőprogram tenné ezt: a tty_insert_flip_char() függvény segítségével. A művelet általában az UART megszakítási függvényen belül játszódik le.

Rövid soros példa

Az 1. kódrészlet azt szemlélteti, hogyan lehet soros illesztőprogramot bejegyezni a soros illesztőrétegnél, illetve hogyan lehet egy soros kaput bejegyezni. Ez a soros kapu viselkedésében nagyon hasonlít a korábbi soros tty példa-illesztőprogramra, a kapu nyitott állapotának fennmaradásáig két másodpercenként érkezik egy karakter. A neki küldött karaktereket egy write(2) hívással a rendszermag hibakereső naplójába írja.

Linux Journal 2002. december, 104. 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.