RTOS eszközmeghajtók illesztése beágyazott Linuxhoz

Alakítsuk hagyományosan durva és kusza RTOS kódunkat szépen formázott Linux eszközmeghajtókká.

A *Linux* viharként tört be a beágyazott rendszerek piacára. Az ipari elemzők szerint az új beágyazott 32- és 64-bites tervek fele-egyharmada a *Linuxra* épít. Több alkalmazási területet már most is a beágyazott *Linux* ural (például *SOHO* hálózatok és a képalkotó/többfunkciós perifériák), de öles léptekkel halad a tárolórendszerek (*NAS*/*SAN*), a digitális otthoni szórakoztatás (*HDTV*/*PVR*/*DVR*/*STB*) és a kézi/vezeték nélküli készülékek elsősorban a digitális mobiltelefonok piaca felé is.

A beágyazott *Linux* alkalmazások persze nem Minervát utánozva pattannak ki a fejlesztők fejéből. A legtöbb projekthez több ezer vagy akár több millió soros örökölt forráskód szükséges. Igaz ugyan, hogy beágyazott rendszerek százai sikeresen hozták át Linux alá a már meglévő kódot (jó példa erre a *Wind River's VxWorks* vagy a *pSOS*, *VRTX*, *Nucleus* és más *RTOS* rendszerek) a módszer gyakorlata azért még nem teljesen kitaposott út.

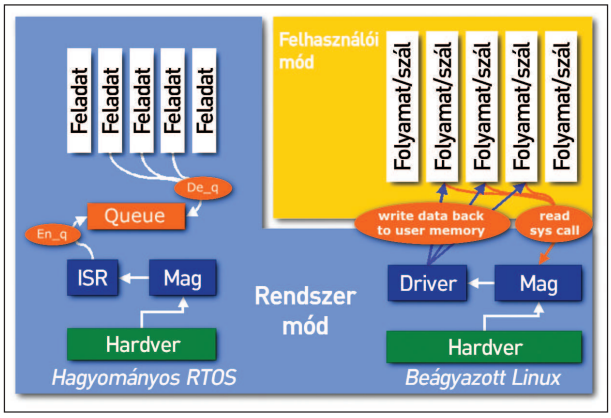
Mind a mai napig az örökölt *RTOS* alkalmazások áthozatalával foglalkozó irodalom nagy része az *RTOS* API-ra, feladatkezelésre és ütemező modellekre és ezek *Linux* felhasználó térbeli megfelelőire koncentrál. Az erősen I/O-függő beágyazott programozási témában legalább ilyen fontos az *RTOS* alkalmazás alkatrész csatoló kódjának a szabványos *Linux* eszközmeghajtó modellhez igazítása.

Cikkünkben a hagyományos beágyazott alkalmazásokban leggyakrabban használt memóriatérkép alapú I/O megközelítéseket szeretnénk sorra venni. A kiszolgáló rutinok (*ISR*) és felhasználói-szálak alkatrész eléréseinek ad hoc szerű felhasználásától kezdve egészen a bizonyos *RTOS* eszköztárakban megtalálható félig-formális meghajtó modellekig jutunk el. Bemutatjuk azokat a heurisztikus és egyéb módszereket melyekkel az *RTOS* kódot szépen formázott *Linux* eszközmeghajtóvá alakíthatjuk. A cikkünk különös figyelmet szentel az *RTOS* kód és a *Linux* memória belapozási eltéréseinek, a sor alapú I/O sémák átírásának, valamint részletesen foglalkozunk az *RTOS* I/O, helyi *Linux* meghajtók és démonok számára használható formára hozásával.

RTOS I/O megoldások

Számos *RTOS* alapú rendszerben alkalmazott I/O technikára leginkább talán a „kötetlen" szó illik. A legtöbb *RTOS* régebbi *MMU* nélküli processzorokra készült, így aztán akkor is figyelmen kívül hagyják a memóriakezelést, ha történetesen van *MMU* és nem tesznek különbséget a logikai és fizikai címzés között. A legtöbb *RTOS* teljes egészében privilegizált módban (rendszer módban) fut, ami látszólag növeli a teljesítmény. Mint ilyen, az *RTOS* alkalmazás és rendszer kód elérheti a gép teljes címterét, a memóriába lapozott eszközöket és I/O utasításokat. Tulajdonképpen meglehetősen nehéz elválasztani egymástól az *RTOS* alkalmazás kódot a meghajtó kódtól, már ahol egyáltalán létezik ilyen megkülönböztetés.

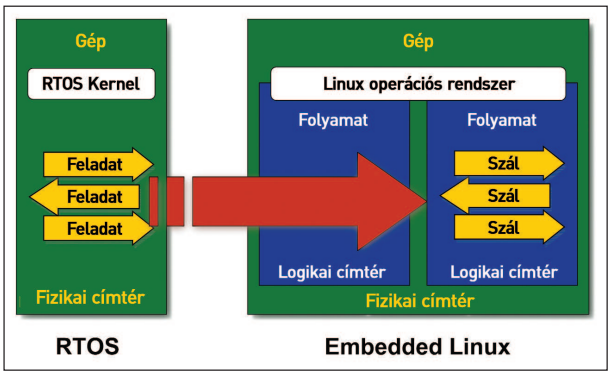
Ez a kötetlen megoldás az I/O megoldások ad hoc jellegű felhasználásához és sok esetben a felismerhető meghajtó modell teljes hiányához vezet. Az ilyen egyenlőség elvű, felosztás nélküli munkakezelés tükrében tanulságos lehet megnézni az *RTOS*-alapú alkalmazásokra vonatkozó néhány kulcselgondolást és gyakorlatot.



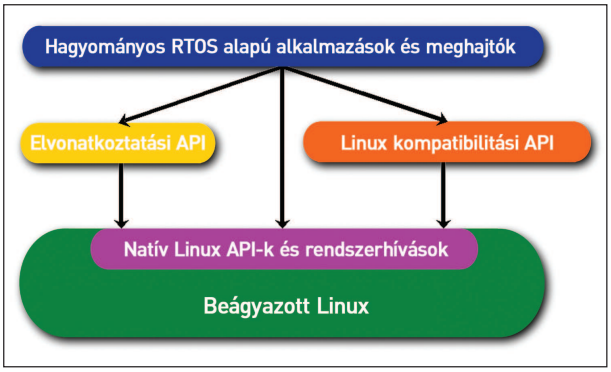
1. ábra Tipikus I/O és adatküldés összehasonlítása hagyományos RTOS és Linux rendszerekben

Sorközi (in-line) memóriába lapozott elérés

Amikor az 1980-as évek közepén megjelentek az üzleti *RTOS* termékek, a legtöbb beágyazott program nagy fő ciklusokat tartalmazott amelyeket az idő-kritikus műveleteket kezelő I/O és *ISR* beszúrások tarkítottak. A fejlesztők első-



2. ábra RTOS feladatok átültetése Linux folyamat alapú szálakká



3. ábra Többágú megközelítés az RTOS kód és API-k Linux átiratához

sorban a párhuzamosság erősítése érdekében terveznek *RTOS* és végrehajtó vezérlést a projektjeikbe, ugyanakkor idegenkedtek bármilyen más szóba kerülő szerkezettől. Így aztán még ha az *RTOS* egyébként lehetővé is tette az I/O formalizmust, a beágyazott programozók inkább a sorközi I/O utasításokat használták:

```
#define DATA_REGISTER   0xF00000F5

char getchar(void) {
  return (*((char *) DATA_REGISTER));
}


void putchar(char c) {
  *((char *) DATA_REGISTER) = c;
}
```

A fegyelmezettebb fejlesztők általában elhatárolták az összes ilyen I/O kódot az eszközfüggetlen kódtól, de rengeteg I/O spagettivel is találkoztam már. Amikor ilyen erőteljes sorközi, memóriába lapozott I/O felhasználással találkoznak, a *Linux* témában zöldfülűnek számító beágyazott fejlesztők gyakran kísértést éreznek, hogy a kódot egy az egyben átvigyék felhasználói térbe és a regisztercímek `#define` megadásait `mmap()` hívásokká alakítsák. Ez a megközelítés bizonyos prototípus megadásoknál jól működik, de nem támogatja a megszakításkezelést, korlátozott valós idejű érzékenységgel rendelkezik, nem igazán biztonságos és üzleti megoldásokban semmiképp sem megfelelő.

RTOS ISR-ek

Linux alatt a megszakítás szolgáltatások kizárólag a rendszermag birodalmába tartoznak. *RTOS* alatt az *ISR* kód szabad formájú és visszatérési módján kívül gyakran semmi nem különbözteti meg az alkalmazás kódtól. Sok *RTOS* ajánl fel olyan rendszerhívást vagy makrót amellyel a kód meghatározhatja saját környezetét. Ilyen például a *Wind River VxWorks* `intContext()` függvénye. Általános megoldás továbbá, hogy az *ISR*-ek szabványos könyvtárakat használnak, ezáltal további újra-belépési és hordozhatósági problémákat vetve fel.
A legtöbb *RTOS* támogatja az *ISR* kódok regisztrációját valamint magára vállalja a megszakítás elbírálást és *ISR* indítást. Bizonyos primitív beágyazott vezérlők ugyanakkor csak az *ISR* kezdőcímek közvetlen illesztését támogatják az alkatrész vektortábláiba. Még ha az írási-olvasási műveleteket sorközi módszerrel a felhasználói térbe helyezzük is, a *Linux ISR* kódnak mindenképpen a rendszermag térbe kell kerülnie.

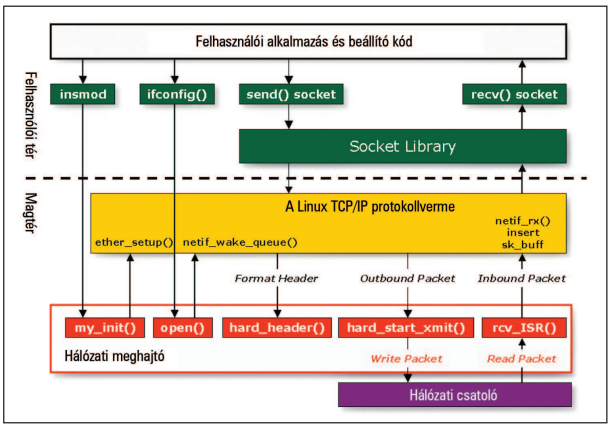
RTOS I/O alrendszerek

A legtöbb *RTOS* testreszabott szabványos C futásidejű könyvtárral rendelkezik (ilyen a *pREPC* a *pSOS* esetében) a *ISV*-khez pedig szelektíven foltozott C könyvtárakat (*libc*) alkalmaz. A *glibc*-vel is ugyanez a helyzet. Így az *RTOS*-nak legalább a szabványos C-stílusú I/O alapelemeit támogatnia kell, ide értve az `open`, `close`, `read`, `write` és `ioctl` rendszerhívásokat. A legtöbb esetben ezek a hívások és hozzátartozóik vékony réteget alkotnak az I/O alapelemek körül. Érdekes módon, mivel a legtöbb *RTOS* nem támogat fájlrendszereket, azok, amelyek Flash vagy forgó médiához szánt fájl absztrakciós réteget is tartalmaznak gyakran teljesen más kódot és/vagy API-t alkalmaznak. Jó példa erre a *pSOS pHILE* rendszere. A *Wind River VxWorks* a legtöbb *RTOS* rendszert lepipálva képességekben gazdag I/O alrendszert is kínál, elsősorban a hálózati csatolófelületek és média kapcsán felmerülő nehézségek kezelésének általánosítása érdekében.
Sok *RTOS* támogatja az alsó fél kezelést, azaz, valamilyen módszert amivel az I/O feldolgozás átvihető a megszakítható és/vagy preemptív környezetbe. Mások nem rendelkeznek ilyesmivel viszont van valamilyen más, hasonló eredményt biztosító mechanizmusuk, például megszakítás egymásba ágyazás.

Tipikus RTOS alkalmazás I/O szerkezete

Az *1. ábra* diagrammja általánosan használt I/O sémát (csak bemenet) és adatközvetítési útvonalat mutat be a vezérlő alkalmazáshoz. A feldolgozás a következő módon zajlik:

- az alkatrész megszakítás kiváltja az *ISR* végrehajtását.
- Az *ISR* elvégzi az alapfeldolgozást és vagy helyben befejezi a bemeneti műveletet vagy az *RTOS*-ra bízza az időzített végrehajtást. Bizonyos esetekben a későbbre halasztott végrehajtást valami olyasmi kezeli, amit Linux alatt felhasználói szálnak itt pedig normál *RTOS* feladatnak nevezünk.
- Ahol és amikor az adat ténylegesen beérkezik (azaz az *ISR* vagy a lehalasztott környezetben) a kész adatok sorba rendeződnek. Az *RTOS ISR* elérheti az alkalmazás sor *API*-t és egyéb *IPC*-ket – lásd az *API* táblát.



4. ábra Linux hálózati meghajtók blokkdiagrammja

- Egy vagy több alkalmazás feladat aztán kiolvassa az üzenetet a sorból és felhasználja az átadott adatokat.

A kimenetet gyakran egyedülálló módszerekkel oldják meg – `awrite()` vagy hasonló rendszerhívások helyett egy vagy több ***RTOS*** alkalmazás-feladat helyez adatokat a sorba.
A sort aztán egy I/O rutin vagy ***ISR*** olvassa be, amely válaszol a küldésre kész megszakításnak, rendszerórának vagy más a sor tartalmára várakozó alkalmazás feladatnak. Ezután az I/O műveletet követlenül végzi, szavazva (polled) vagy DMA-n keresztül.

RTOS I/O hozzárendelés Linux-hoz

A fent bemutatott sor alapú termelő/fogyasztó I/O model csak egy a sok ad hoc megközelítés közül amelyeket az örökölt rendszerek alkalmaznak. Folytassuk ezt az egyszerű példát és nézzünk meg néhány további (újra)megvalósítást a beágyazott ***Linux*** rendszerben.
Azok a fejlesztők akik tartózkodnak a Linux meghajtó tervezés jellemzőinek megismerésétől esetleg akik nagyon sietnek, valószínűleg megpróbálják változatlanul áthozni a felhasználói térbe a sor alapú elképzelést. Ebben a meghajtó beillesztési megoldásban a memóriába lapozott fizikai I/O a felhasználói térben jön létre az `mmap()` által biztosított mutató segítségével:

```
#include <sys/mman.h>

#define REG_SIZE    0x4
/* eszköz regiszter méret*/
#define REG_OFFSET 0xFA400000
/* az eszköz fizikai címe*/

void *mem_ptr;
/* visszahivatkozás a memóriába lapozott
➥eléréshez*/
int fd;

fd=open("/dev/mem",O_RDWR);
/* nyitott fizikai memória (root-nak kell lenni) */

mem_ptr = mmap((void *)0x0,
              ➥REG_AREA_SIZE,
              ➥PROT_READ+PROT_WRITE,
                MAP_SHARED, fd, REG_OFFSET);
/* tényleges  mmap() hívás*/
```

A folyamat alapú felhasználói szál ugyanazt a feladatot látja el, mint amit az ***RTOS***-alapú ***ISR*** vagy halasztott feladat. Ezt követően az ***SVR4 IPC*** `msgsnd()` hívás segítségével a sorba helyezi az üzenetet, ahonnan valamely másik helyi szál vagy folyamat lekérheti azt a `msgrcv()` függvénnyel.
Az ilyen gyors megoldás a prototípusokhoz ugyan elegendő lehet, de fejleszthető kódot készíteni márt komolyabb kihívás. Először szóljunk a megszakítások felhasználói térbe helyezéséről. A ***DOSEMU***-hoz hasonló projektek jelzés alapú I/O megszakításokat használnak a ***SIG***-en keresztül (***silly interrupt generator***, azaz buta megszakításgenerátor) de a felhasználó térbeli megszakítás kezelés meglehetősen lassú – milliszekundumos lappangás a rendszermag alapú ISR pár tíz mikroszekundumos lappangási idejéhez képest. Továbbá a felhasználói környezet ütemezése, még preemptív rendszermag és valós idejű rendszabályok alkalmazása esetén sem garantálnak 100%-ig valós idejű végrehajtást a felhasználói térbeli I/O megszakításoknak.
Mindenképpen javasolt, hogy fogjuk meg a dolog végét és legalább egy egyszerű ***Linux*** meghajtót készítsünk, ami a megszakításokat rendszermag szinten kezeli. Egyszerű karakteres vagy blokkos meghajtó közvetlenül elhelyezheti az alkalmazás adatait a felső félben (top half) vagy elhalasztva a feldolgozást átadhatja azokat egy szálacskának (tasklet), egy rendszermag szálnak vagy a 2.6-os rendszermagban az alsó félben (bottom half) megjelent munkasor mechanizmusnak. Az eszközt egyszerre több alkalmazás szálfolyamat is megnyithatja szinkronizált olvasást végezve rajta, éppen úgy , ahogy az ***RTOS*** alkalmazás végzett szinkronizált sorfogadó hívásokat. Ehhez a megközelítéshez legalább a fogyasztói szál I/O részt újra kell írni, hiszen sor fogadó műveletek helyett most eszközolvasást végzünk.
Amennyiben egyszerűsíteni szeretnénk a beágyazott ***Linuxra*** átvitelt, meghagyhatjuk a sor-alapú sémát és készíthetünk még egy szálat vagy démont amely a frissen elkészített eszköz I/O adataira várakozik. Amikor az adat megérkezett, a szál/démon felébred és az adatot feldolgozó alkalmazás szál vagy folyamat számára elteszi a sorba.

Átalakítási megoldások

Az ***RTOS*** kód beágyazott Linuxhoz alakítása alapelveiben nem különbözik az üzleti alkalmazások átalakításától. Miután az átalakítás logisztikai részével (***make/build*** szkriptek és metódusok, a fordító kompatibilitás, helyek, `include` állományok és így tovább) már végeztünk, a kód szintű átalakítás kihívásai tulajdonképpen az alkalmazás szerkezet és API használat területére szűkülnek.
Témánk további megvitatásához tételezzük fel, hogy az alkalmazás rész (azaz az I/O specifikus részen kívül minden) az ***RTOS***-alapú rendszerből egyetlen ***Linux*** folyamattá alakul át. Az ***RTOS*** feladatok ***Linux*** szálak lesznek a feladatközi ***IPC***-k pedig ***Linux inter-process*** és ***inter-thread*** megfelelőikkel helyettesítjük.

1. táblázat *A Linux és az RTOS-ok előjogokhoz kötött utasításai*

	IPC-k	Szinkronizálás	Feladatkezelés	Névtér
RTOS Alkalmazás	Sorok, jelek, levelesládák kötetlen megosztott memória	Szemafórok, mutexek	Teljes RTOS feladatkezelés Repertoire (Linkelési időben)	Teljes alkalmazás, Könyvtárak és rendszer
RTOS Meghajtó	Sorok, jelek, levelesládák kötetlen megosztott memória	Szemafórok, mutexek	Teljes RTOS feladatkezelés Repertoire	Teljes alkalmazás, Könyvtárak és rendszer (Linkelési időben)
Linux alkalmazás	Sorok, jelek, csővezetékek Intra-Process megosztott memória megosztott rendszer memória	Szemafórok, mutexek	Folyamatok és szálak API-k	Helyi folyamatok, statikus és megosztott könyvtárak
Linux meghajtó (statikus)	Megosztott rendszer memória írható/olvasható folyamatmemória	Rendszermag szemaforok forgózárak (Spinlocks)	Rendszermag szálak, taskletek	Teljes rendszermag
Linux modul (dinamikus)	Megosztott rendszer memória írható/olvasható folyamatmemória	Rendszermag szemaforok forgózárak (Spinlocks)	Rendszermag szálak, taskletek	Modul-szintű és exportált rendszermag szimbólumok

Az átírás alapjait nem nehéz megérteni, a buktató a részletekben rejlik. A leglényegesebb részlet pedig mind között az ***RTOS API*** használata, illetve az, hogyan tudjuk megvalósítani ***Linux*** szerkezetekkel.

Amennyiben a projektünk nem időhöz kötött és célunk hordozható kódot készíteni későbbi projektfejlesztésekhez, akkor érdemes egy kis időt tölteni a jelenlegi ***RTOS*** alkalmazás elemzésével és megvizsgálni miképpen illeszkedik a Linux elképzelésekhez. Az ***RTOS*** alkalmazás kód esetében elgondolkozhatunk rajta, át tudjuk-e Linux folyamat-alapú szálakká alakítani az ***RTOS*** feladatokat egy az egyben avagy inkább a teljes ***RTOS*** alkalmazást több ***Linux*** folyamatra bontjuk. Ettől a döntéstől függően meg kell vizsgálnunk a felhasznált ***RTOS IPC***-ket, hogy a helyes ***intra-process*** vagy ***inter-process*** hatáskört válaszuk.

Meghajtó szinten egyértelmű, hogy valamennyi sorközi ***RTOS*** kódot megfelelő meghajtókká kell alakítani. Amennyiben az örökölt alkalmazásunk már eleve szépen felosztott, akár ***RTOS*** I/O ***API***-kat használ, akár külön rétegekbe szedték szét, feladatunk jelentősen egyszerűsödik. Ha viszont az ad hoc I/O kód az egész örökölt kódbázisunkban szanaszét szórva található saját magunknak kell kiköveznünk az utat. Azok a fejlesztők, akik sietősen szeretnék kiszedni az örökölt ***RTOS*** kódot vagy akik prototípust akarnak összekalapálni, valószínűbb, hogy megpróbálják a lehető legtöbb ***RTOS*** API-t helyben ***Linux*** megfelelővel kiváltani. Az elemeket általában, akárcsak az egyedi ***API***-kat, ***IPC***-ket és rendszer adattípusokat, majdnem teljesen átlátszóan viszik át. Mások a `#define` újradefiniálásban és makrókban hisznek. A többiek újrakódolják a részeket, ideális esetben egy elvonatkoztatási réteg részeként.

Az ***API***-alapú átalakítást érdemes az emulációs könyvtárakkal érdemes kezdeni, amelyek sok beágyazott ***Linux*** terjesztésnek részét képezik (például a ***MontaVista*** könyvtárai a ***Wind River VxWorks*** és ***pSOS*** rendszerekhez) avagy olyan harmadik féltől származó ***API***-átírat csomagokat használnak, mint amilyet például a ***MapuSoft*** kínál.

A legtöbb projekt hibrid megközelítést alkalmazza: az összes egyedi vagy könnyen megvalósítható API-t áthozza, újraszerkesztik azokat a részeket, ahol az nem okoz lassulást majd csapd-le-a-vakondokot módszerrel addig foltozzák a maradék kódot amíg le nem fordul és fut.

A rendszermag és a felhasználói tér elérhető API felületei

Akár a teljes újratervezést, akár a gyors de felületes ***API*** megközelítést választjuk, az ***RTOS*** alkalmazásunkat és az I/O kódot szét kell osztanunk, hogy az illeszkedjen a ***Linux*** rendszermag és felhasználói tér elképzeléshez. Az *1. táblázat* bemutatja a ***Linux*** mennyivel szigorúbb az előjogot igénylő utasítások terén mint az örökölt ***RTOS***. Segítségével könnyebben elvégezhetjük a felosztási rész.

Az 1. táblában két pontra különösen felhívnám a fegyelmet:

- az ***RTOS***-ok egyenlőség pártiak, az alkalmazásnak és az I/O kódnak is lehetővé teszi, hogy bármilyen címhez hozzáférjen és szinte bármilyen tevékenységet végezzen, a ***Linux*** ugyanakkor sokkal hierarchikusabb és kötöttebb.
- Az örökölt ***RTOS*** kód a rendszer valamennyi belépési pontját és szimbólumát láthatja, legalább linkeléskor, ugyanakkor a ***Linux*** felhasználói kód a rendszermag kódtól és annak névterétől teljesen elhatárolva készül.

A ***Linux*** előjoghoz kötött eléréshierarchiájának eredményeképpen általában csak a rendszermag kód (meghajtók) érhetik el a fizikai memóriát. Ha egy felhasználói kód ilyesmit szeretne, annak ***root*** jogon kell futnia. Általában véve a felhasználói tér kódja elhatárolódik a ***Linux*** rendszermag kódtól és csak azokat a szándékosan exportált szimbólumokat éri el, amelyek a ***/proc/ksyms*** alatt látszanak. Továbbá a rendszermag hívásai nem közvetlenül történnek, hanem a felhasználói könyvtár kódon keresztül. Ez a fajta szétválasztás szándékos, célja a ***Linux*** biztonságának és stabilitásának növelése. Amikor meghajtót írunk, minden éppen fordítva van. A statikusan meghajtók a teljes rendszermag névterét látják, nem csak az exportokat, ugyanakkor semmilyen rálátásuk nincs

a felhasználói tér folyamat alapú szimbólumaira és belépési pontjaira. Továbbá, ha a meghajtónkat futásidőben betölthető modulba csomagoljuk, programunk csak azokat a csatolófelületeket befolyásolhatja, amelyeket az `*EXPORT_SYMBOL*` makró segítségével kifejezetten exportáltunk a rendszermagban.

Hálózati meghajtók átírása

Mint fentebb említettük, karakteres és blokkos eszközök átírása ***Linux*** alá időigényes, de viszonylag magától értetődő folyamat. A hálózati meghajtók átírása viszont már ijesztőbbnek tűnik. Ne feledjük, hogy a ***Linux*** tulajdonképpen a ***TCP/IP***-vel együtt nőtt fel, sok 1990-es évek végén készült ***RTOS*** már tartalmazott hálózati kódot. Így aztán az örökölt hálózatkezelés sokszor csak a képességek vázát tartalmazza, azaz csak egyetlen folyamatot vagy példányt kezel egyetlen kapun avagy csak a fizikai csatolófelületet kezeli egyetlen hálózati médiumon. Néhány esetben a hálózati szerkezetet utólag csiszolgatták. Ez történt például a ***Wind River VxWorks MUX*** kódjával, hogy támogasson több csatolófelületet és fizikai kapcsolattípust. A rossz hír, hogy valószínűleg újra kell írnunk elég sok, ha nem az összes meglévő hálózati csatolófelületet. A jó hír, hogy a ***Linux*** alatti újra felosztás nem olyan nehéz és példának tucatnyi nyílt forrású hálózati kód áll rendelkezésünkre. Átalakítási munkánk feladata, hogy a *4. ábra* alján található részeket megfelelő csomag formázó és csatolófelület kóddal töltsük fel.
Hálózati meghajtók készítése nem kezdőknek való feladat. Mivel azonban sok ***RTOS*** hálózati meghajtó létező ***GPL Linux*** csatolófelületek forrásából készült, elképzelhető, hogy maga a kód segíti elő törekvéseinket. Továbbá, az integrátorok és tanácsadók elég nagy és egyre bővülő közössége odafigyel és üzletet lát a ***Linux*** átalakításokat végző beágyazott fejlesztők részére nyújtott segítségben. Az áraik is elfogadhatóak.

Összefoglalás

Cikkünk célja, hogy a beágyazott fejlesztőknek egy kis rálátást nyújtson, milyen kihívások és előnyök várhatóak amennyiben teljes programkészletüket ***Linux*** alá viszik át az eredeti ***RTOS*** rendszerről. Persze körülbelül 2.800 szó túl kevés rá, hogy igazán elmerüljünk a meghajtóátírás (meghajtó API-k és buszfelületek címfordítás és egyebek) részleteiben, de a seregnyi meglévő ***GPL*** meghajtó kód egyaránt szolgál kiváló dokumentációként és sablonként átültetési erőfeszítéseink során. Az itt bemutatott irányelvek csapatunkat kívánják segíteni az ***RTOS*** kód linuxos átírásában valamint ötletekkel támogatni az eredeti kód beágyazott ***Linux*** rendszerekhez leginkább illeszkedő újrafelosztását.

Linux Journal 2004. október, 126. szám

E cikk írásakor ***Bill Weinberg*** a Strategic Marketing and Technology Evangelist igazgatójaként a MontaVista és beágyazott Linux fejlesztése terén szerzett több mint 17 évnyi ipari tapasztalat tudását összegezte. Kiterjedt tapasztalatokkal rendelkezik a beágyazott valós idejű rendszerek terén és szakértőnek számít az operációs rendszerek, eszközök, program licencek és programkészítés témákban.