RTLinux için Modül Geliştirme NASIL

Yazan:
Dinil Divakaran
<dinildivakaran (at) rediffmail.com>

Çeviren:
Aysun Özdemir
<aysun951 (at) yahoo.com>

23 Ocak 2007

Özet

Bu belgede RTLinux kurulumu ve gerçek zamanlı Linux programlarının NASIL yazılacağı anlatılmaktadır.

Konu Başlıkları

Bu çevirinin sürüm bilgileri:

1.0	Ocak 2006	aö
İlk çeviri		

Özgün belgenin sürüm bilgileri:

1.1	2002-08-29	dd
...		

Yasal Açıklamalar

1. Giriş

1.1. Amaç

Bu belge yeni başlayan kullanıcıların RTLinux ile olabildiğince sorunsuz bir şekilde çalışmalarını sağlamayı amaçlar.

1.2. Bu Belgeyi Kimler okumalı

Bu belge gerçek zamanlı çekirdeğin çalışmasını bilmek isteyen herkes içindir. Modül programlamaya alışkın olanlar için bu belge zor gelmeyecektir. Ama diğerlerinin üzülmesine de gerek yok; sadece modül programlamanın temel kavramları gerekli ve onlardan da gerektiği yerlerde bahsedeceğiz.

1.3. Teşekkür

İlk olarak danışmanım Pramode C. E.'e yardımları ve cesaretlendirmeleri için teşekkür ediyorum. Victor Yodaiken'e de teşekkür ediyorum, onun birçok makaleden biraraya getirdiği bilgiler olmasaydı belgenin bu halini alması mümkün olmazdı. Ayrıca "Bir Linux–tabanlı Gerçek–zamanlı İşletim Sistemi" başlıklı tezi için Michael Barabanov'a teşekkürler.

1.4. Geri Besleme

Belgeyle ilgili yorumlarınızı bana eposta ile göndermekten çekinmeyin. Belgede hatalar varsa bana bildirin ki bir sonraki sürümde onları düzeltebileyim. Teşekkürler.

2. RTLinux Kurulumu

RTLinux çekirdeğinin derlenmesindeki ilk adım 2.2.18[B3] (sadece x86 için) ya da 2.4.0–test1[B4] (x86, PowerPC, Alpha) sürümü önceden yamanmış çekirdeğin `/usr/src/` içine indirilmesi ve sıkıştırılmış dosyaların açılmasıdır. Ayrıca RTLinux çekirdeğinin yeni bir kopyasını (3.0 sürümünü) `http://www.rtlinux.org` adresinden `/usr/src/rtlinux/` içine koyun. (Komut satırı gösteriminde **$** kullanacağız).

1. Şimdi RTLinux çekirdeğini yapılandıralım:

```
$ cd /usr/src/linux
$ make config
        ya da
$ make menuconfig
        ya da
$ make xconfig
```

2. Çekirdeği derlemek için:

```
$ make dep
$ make bzImage
$ make modules
# make modules_install
# cp arch/i386/boot/bzImage /boot/rtzImage
```

3. Sonraki adım LILO yu biçimlendirmektir. `/etc/lilo.conf` dosyasına aşağıdaki satırları ekleyin:

```
image=/boot/rtzImage
label=rtl
read-only
```

```
root=/dev/hda1
```

Uyarı

Yukarıda ki /dev/hda1'i sizin kök dosya sisteminiz ile değiştirin. Hangisi olduğunu bulmanın en kolay yolu /etc/lilo.conf dosyasındaki "root=" girdisine bakmaktır.

4. Şimdi bilgisayarınızı yeniden başlatın ve LILO komut satırına 'rtl' yazarak RTLinux çekirdeğini yükleyin. Sonra /usr/src/rtlinux/ dizinine geçerek RTLinux'u yapılandırın.

```
$ make config
        ya da
$ make menuconfig
        ya da
$ make xconfig
```

5. Son olarak RTLinux'u derleyin ve kurun.

```
$ make
$ make devices
# make install
```

En son adımda kullanıcı programları oluşturmada ve derlemede gerekli olacak RTLinux için öntanımlı kurulum dizinini içeren (başlık dosyaları, yardımcı araçlar ve belgelendirmeyi içerir) dizin oluşturulacak:

/usr/rtlinux-xx (xx sürümü gösterir)

Buna da bir sembolik bağ oluşturulacaktır:

/usr/rtlinux

Lütfen ileriye yönelik uyumluluğu sağlayabilmek için kendi RTLinux programlarınızın hepsinin /usr/rtlinux'u öntanımlı yol olarak kullanmalarını sağlayın.

Bilgi

Eğer Linux çekirdek seçeneklerinde herhangi bir değişiklik yaparsanız lütfen bunları da yapmayı unutmayınız:

```
$ cd /usr/src/rtlinux
$ make clean
$ make
# make install
```

3. Neden RTLinux

RTLinux tasarımının nedenleri standart Linux çekirdeğinin çalışması incelenerek anlaşılabilir. Linux çekirdeği, donanımı kullanıcı seviyesindeki görevlerden ayırır. İyi bir ortalama başarım sağlamak veya iyi bir iş çıkarmak için çekirdek her göreve birer öncelik atar ve bunlar üzerinde bazı zamanlama algoritmaları kullanır. Bu görev, işlemci tarafından zaman dilimlerine bölüştürülerek dış ortamda çalıştırılır, böylece çekirdek her bir kullanıcı seviyeli görevi geçici olarak durdurabilir. Bu sıralama algoritmaları ile birlikte aygıt sürücüleri, kesintisiz sistem çağrıları, geçersiz durum kesme sinyali kullanımı ve sanal bellek işlemleri sonuçları önceden kestirilemeyen işlem kaynaklardır. Dolayısıyla bu kaynaklar bir görevin gerçek zamanlı başarımı için engel teşkil eder.

'mpg123' ya da başka bir çalıcı kullanarak müzik dinliyorsanız gerçek zamanlı olmayan başarım hakkında bir fikriniz vardır. Önceden belirlenen zaman dilimi için bu süreç çalıştırıldıktan sonra, standart Linux çekirdeği

görevin önceliğini hiçe sayıp işlemciyi başka bir göreve verebilir (örneğin; X sunucusunu veya Netscape'i açan bir göreve). Sonuç olarak müziğin sürekliliği kaybolur. Böylece çekirdek, işlemci zamanının bütün süreçler arasında doğru dağılımını sağlamayı denerken bir takım başka olayların meydana gelmesini engelleyebilir.

Bir gerçek zamanlı çekirdek, altındaki süreçlerin zamanlama gereksinimlerini garantileyebilmelidir. RTLinux çekirdeği, yukarıda bahsedilen sonuçları önceden kestirilemeyen işlem kaynaklarının kaldırılmasıyla gerçek zamanlı görevleri gerçekleştirebilir. RTLinux çekirdeğinin standart Linux çekirdeği ve donanım arasında oturduğunu varsayabiliriz. Bu durumda Linux çekirdeği gerçekzaman katmanını asıl donanım olarak görür. Artık, kullanıcı her bir görev için ayrı ayrı olmak üzere bütün görevler için önceliklikleri belirleyebilir ve tanıtabilir. Kullanıcı zamanlama algoritmaları, öncelikler, çalışma sıklığı vb. üstünde kararlar vererek süreçler için doğru zamanlamayı sağlayabilir. RTLinux çekirdeği, en düşük önceliği standart Linux çekirdeğine atar. Böylece kullanıcı görevleri gerçekzamanda çalışabilir.

Güncel gerçek zamanlı çalışma bütün donanım kesme sinyallerinin durdurulması ile elde edilir. Sadece RTLinux ile ilgili kesmeler için rutin kesme servisi çalışır. Bütün diğer kesme sinyalleri, RTLinux çekirdeğinin boşta kalması nedeniyle standart Linux çekirdeği çalışmaya başladığında yazılım kesmeleri olarak Linux çekirdeğine aktarılmak üzere tutulurlar. RTLinux yürütücüsü yürütme gücünü kendi üstüne almaz.

Gerçek zamanlı görevler ayrıcalıklıdır (yani donanıma doğrudan erişirler) ve sanal belleği kullanmazlar. Gerçek zamanlı görevler, belleğe özdevimli yüklenebilen özel Linux modülleri olarak yazılırlar. Bir gerçek zamanlı görev için ilklendirme kodu, gerçek zamanlı görev yapısını ilklendirir ve RTLinux çekirdeğine süre sonu, periyodu ve son anda ortaya çıkan kısıtlamalar hakkında bilgi verir.

RTLinux'un varlığı, Linux çekirdeğinin yapısına dokunmadığından Linux çekirdeği ile birlikte anlamlı olur. Linux çekirdeğinin gelecekteki gelişimine engel olmaksızın, nispeten basit değişikliklerle mevcut Linux çekirdeğini tam bir gerçek zamanlı ortama dönüştürmeyi başarır.

4. RTLinux Programları Yazmak

4.1. Modül Yazımına Giriş

Modül nedir, dersek? Bir Linux modülü **gcc**'nin `-c` seçeneği ile kullanılması sonucu oluşan nesne dosyasından başka birşey değildir. Bir modül, sıradan bir C dili dosyasının `main()` işlevi olmaksızın derlenmesiyle oluşur. `main()` işlevi yerine `init_module/cleanup_module` işlev çifti vardır:

- `init_module()` işlevi, modül çekirdeğe eklenirken çağrılır. Bu işlev başarı halinde 0, başarısızlık halinde negatif değer döndürmelidir.
- `cleanup_module()` işlevi, tam modül kaldırılırken çağrılır.

Genellikle, `init_module()` ya çekirdekle bir şeyler için bir eylemcinin varlığını bildirir ya da bir çekirdek işlevini kendi kodu ile değiştirir (genellikle kod bir şeyler yaptıktan sonra orjinal işlevi çağırır). `cleanup_module()` işlevinin `init_module()` işlevinin yaptığını geri aldığı kabul edilir, böylece modül emniyetle kaldırılabilir.

Örneğin, eğer siz `module.c` diye bir C dosyası yazarsanız (`main()` işlevi `init_module()` ve `cleanup_module()` işlevleriyle değiştirilerek), bu kod,

```
$ gcc -c {BAZI-SEÇENEKLER} my_module.c
```

ile bir modüle dönüştürülebilir. Bu komut;

```
$ insmod module.o
```

gibi bir '**insmod**' komutuyla çekirdeğe eklenebilen `module.o` isimli bir modül dosyası oluşturur. Benzer biçimde, bu modülü kaldırmak için '**rmmod**' komutunu kullanabilirsiniz:

```
$ rmmod module
```

4.2. RTLinux Evrelerinin Oluşturulması

Bir gerçek zamanlı uygulama genellikle çeşitli evrelerin çalışmalarının birleşimidir. Evreler bir ortak adres uzayını paylaşan hafif süreçlerdir. RTLinux'da, bütün evreler Linux çekirdeğinin adres uzayını paylaşır. Evreleri kullanmanın avantajı, evreler arasındaki geçişin bağlamsal geçişlere göre oldukça ucuz olmasıdır. Aşağıdaki örnekte görüleceği gibi farklı işlevler kullanarak bir evrenin çalışması üzerinde tam bir denetim kurabiliriz.

4.3. Örnek Bir Program

Bir evrenin çalışmasını anlamanın en iyi yolu, bir gerçek zamanlı programı izlemektir. Örneğin, aşağıda görünen program her saniye bir kez çalışacak ve her bir yineleme sırasında 'Merhaba Dunya' yazacak.

Örnek 1. hello.c dosyası

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

pthread_t evre;

void * evre_kodu(void)
{
        pthread_make_periodic_np(pthread_self(), gethrtime(), 1000000000);

        while (1)
        {
                pthread_wait_np ();
                rtl_printf("Merhaba Dunya\n");
        }

        return 0;
}

int init_module(void)
{
   return pthread_create(&evre, NULL, evre_kodu, NULL);
}

void cleanup_module(void)
{
   pthread_delete_np(evre);
}
```

Şöyle `init_module()` ile başlayalım. `init_module()`, **`pthread_create()`** işlevini çağırır. Bu, çağrılan evre ile aynı anda çalışan yeni bir evre oluşturmak içindir. *Bu işlev sadece Linux çekirdek evresinden çağrılabilir (`init_module()` kullanılarak).*

```
int  pthread_create(pthread_t      *evre,
                    pthread_attr_t *oznitelik,
                    void           * (*evre_kodu)(void *),
                    void           *arg);
```

Oluşturulan yeni evre `pthread.h` başlık dosyasında tanımlanan **`pthread_t`** türündedir. Bu evre, `evre_kodu()` işlevini, argümanını *arg* ile aktararak çalıştırır. *öznitelik* değişkeni yeni evreye uygulanacak evre özniteliklerini belirler. Eğer *öznitelik* NULL ise, öntanımlı öznitelikler kullanılır.

Bundan dolayı burada, `evre_kodu()` argümansız çağrılır. `evre_kodu` üç bileşenden (ilklendirme, çalışma ve sonlandırma) oluşur.

İlkllendirme aşamasında `pthread_make_periodic_np()` çağrısı yapılır.

```
int pthread_make_periodic_np(pthread_t evre,
                             hrtime_t  başlatma_anı,
                             hrtime_t  süre);
```

`pthread_make_periodic_np` *evre*'yi çalışmaya hazır olarak imler. Evre *başlatma_anı*'nda çalışmasına başlayacak ve nanosaniyelerle belirlenmiş bir *süre*'yle çalışacaktır.

`gethrtime` işlevi sistemin başlamasından beri geçen zamanı nanosaniyeler cinsinden döndürür.

```
hrtime_t gethrtime(void);
```

Bu zaman asla sıfırlanamaz ya da ayarlanamaz. `gethrtime`, daima monoton artan değerler verir. **`hrtime_t`** türü, 64 bitlik işaretli tamsayı belirtir.

`pthread_make_periodic_np()` çağrısıyla evre, işlemci zamanlayıcısına bu evreyi bir saniyede bir çalıştırmasını söyler. Bu, evre için ilklendirme bölümünün sonudur.

`while()` döngüsü, çalışmakta olan gerçek zamanlı evreyi sonraki *süre*'nin başlamasına kadar beklemeye alacak `pthread_wait_np()` işlevine bir çağrı ile başlar. Bu evre önceki bir `pthread_make_periodic_np` çağrısını ile devreye alınır. Evre tekrar çağrılır çağrılmaz, başka bir `pthread_wait_np()` çağrısıyla karşılaşana kadar `while` döngüsünde kalanlar çalıştırılır.

Bizim döngüden çıkmamızı sağlayacak bir yol olmadığından bu evre 1Hz'lik sıklıkla çalıştırılmaya devam edecektir. Programı sonlandırmanın tek yolu **`rmmod`** komutu ile onu çekirdekten ayırmaktır. Bu, evreye ayrılan özkaynakları serbest bırakarak evrenin iptal edilmesini sağlayan `pthread_delete_np()` işlevini çağıracak olan `cleanup_module()` işlevini çağıracaktır.

5. Derleme ve Çalıştırma

`hello.c` programını çalıştırmak için (tabii ki RTLinux açılışından sonra) aşağıdaki adımları sırası ile yapmalısınız:

1. GCC derleyicisini kullanarak kaynak kodunu derleyip bir modül oluşturun. Birşeyleri basitleştirmek adına bir Makefile oluşturmak daha iyidir. Böylece kodu derlemek için tek ihtiyacınız **`make`** yazmak olacaktır.

   Aşağıdaki satırları `Makefile` isimli bir dosyaya yazarak Makefile'ı oluşturabilirsiniz:

   ```
   include rtl.mk
   all: hello.o
   clean:
       rm -f *.o
   hello.o: hello.c
       $(CC) ${INCLUDE} ${CFLAGS} -c hello.c
   ```

2. `rtl.mk` dosyasını `hello.c` ve `Makefile` ile aynı dizinin içine kopyalayın. `rtl.mk` dosyası, kodu derlemek için gerekli bütün seçenekleri içeren bir başlık dosyasıdır. Bunu RTLinux kaynak ağacından kopyalayabilirsiniz ve `hello.c` dosyasının yanına koyabilirsiniz.

3. Kodu derlemek için '**`make`**' komutunu kullanın:

```
$ make
```

4. Oluşan ikilik nesne dosyası RTLinux tarafından çalıştırılacak çekirdeğe eklenmelidir. Bunun için '**rtlinux**' komutunu kullanmak gerekir (tabii önce 'root' olmak gerekir).

```
# rtlinux start hello
```

Artık **hello.o** programının iletisini saniyede bir yazdırması izleyebiliyor olmalısınız. Makinenizin yapılandırmasına bağlı olarak çıktıyı konsolda doğrudan ya da

```
$ dmesg
```

komutunu girerek görebilirsiniz. Programı sonlandırmak için modülü çekirdekten ayırmanız gerekir. Bunu yapmak için:

```
$ rtlinux stop hello
```

Modülü eklemek ve çıkarmak için diğer bir yol da sırayla **insmod** ve **rmmod** kullanmaktır.

Burada örnek programımızı çok basit yaptık. Gördüğümüzün aksine burada bir programda çok sayıda evre olabilir. Öncelik evre oluşturulurken belirtilebilir ve daha sonra değiştirilebilir. Ayrıca kullanılacak işlemci zamanlama algoritmasını da seçebiliriz. Aslında kendi zamanlama algoritmamızı kendimiz yazabiliriz!

Örneğimizde, evreye öncelik olarak 1 verebilir ve `evre_kodu()` işlevinin başlangıcına aşağıdaki satırları ekleyerek FIFO zamanlamasını seçebiliriz :

```
struct sched_param p;
p.sched_priority = 1;
pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);
```

6. Süreçlerarası İletişim

Önceki bölümlerde gördüğümüz örnek program bir gerçek zamanlı süreç olarak bilinen süreçtir. Bir uygulama programının her bir parçası gerçek zamanlı olmak zorunda değildir. Programın sadece hassas zaman kısıtlamaları gerektiren bölümleri bir gerçek zamanlı süreç gibi yazılmalıdır. Diğer bölümleri kullanıcı seviyesinde yazılıp çalıştırılabilir. Kullanıcı seviyesinde çalışan süreçleri yazmak, çalıştırmak ve hatalarını ayıklamak gerçek zamanlı evrelerden çoğunlukla daha kolaydır. Fakat tam bu noktada, kullanıcı seviyesindeki Linux süreçleri ile gerçek zamanlı evreler arasında iletişimi sağlayacak bir yöntem ihtiyacı ortaya çıkar.

Süreçlerarası iletişimin çeşitli yolları vardır. Biz en önemli ve en ortak iletişim yolundan, gerçek zamanlı FIFO'dan bahsedeceğiz.

6.1. Gerçek Zamanlı FIFO

Gerçek zamanlı FIFOlar tek yönlü kuyruklardır (First In First Out – ilk giren ilk çıkar). Yani bir süreç FIFOnun bir ucunda veri yazarken diğer süreçler FIFOnun diğer ucundan bilgileri okuyabilir. Genellikle bu süreçlerin bir tanesi gerçek zamanlı evre, diğeri kullanıcı seviyesindeki süreçtir.

Gerçek zamanlı FIFOlar aslında, ana numarası 150 olan karakter aygıtlarıdır (/dev/rtf*). Gerçek zamanlı evreler kullanacakları FIFO'ları belirtmek için bir tamsayı kullanır (örneğin, `/dev/rtf2` için 2). FIFOların numaraları için bir sınır vardır. FIFOlarla çalışmak için `rtf_create(), rtf_destroy(), rtf_get(), rtf_put()` vb işlevler vardır.

Diğer taraftan, Linux kullanıcı süreci gerçek zamanlı FIFOları normal karakter aygıtları gibi görür. Bu yüzden `open(), close(), read()` ve `write()` gibi işlevler bu aygıtlarda kullanılabilir.

6.2. FIFO Kullanan Bir Uygulama

İlk olarak, PC hoparlöründen müzik çalan (sadece iki notalı) basit bir C programını (ismi `pcaudio.c` olsun) göz önüne alalım. Şimdi, nota çalmak için sadece `/dev/rtf3` karakter aygıtına yazmamız gerektiğini varsayalım. (Sonra, bir gerçek zamanlı sürecin FIFOdan (`/dev/rtf3`) okumasını ve PC hoparlörüne göndermesini göreceğiz)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define DELAY 30000

void make_tone1(int fd)
{
  static char buf = 0;
  write (fd, &buf, 1);
}

void make_tone2(int fd)
{
  static char buf = 0xff;
  write (fd, &buf, 1);
}

main()
{
  int i, fd = open ("/dev/rtf3", O_WRONLY);
  while (1)
  {
    for (i=0;i<DELAY;i++);
    make_tone1(fd);
    for (i=0;i<DELAY;i++);
    make_tone2(fd);
  }
}
```

Şimdi, yukarda görünen program (`pcaudio.c`) derlenir ve çalıştırılırsa, bir kare dalgaya uygun düzenli pıtırtılar oluşturmalı. Fakat bundan önce '`/dev/rtf3`'ten okumak için bir modüle ve uygun veriyi PC hoparlörüne göndermemiz lazım. Bu gerçek zamanlı programı rtlinux kaynak ağacında (`/usr/src/rtlinux/examples/sound/`) bulabilirsiniz. `sound.o` modülünü '**`insmod`**' komutunu kullanarak çekirdeğe ekleyin.

Modülü ekledikten sonra aygıttan okumak için, programımızı artık çalıştırabiliriz ('**`gcc`**' kullanarak derleyin ve sonra oluşan '`a.out`'u çalıştırın). Bu şekilde süreç, sistemde başka süreç olmadığı zaman (zamanın tükenmesi) biraz daha düzenli tonlar üretir. Fakat diğer konsolda X sunucusu başlatıldığı zaman tonda sessizlik daha uzun süreli olmaya başlar. Sonunda bir 'find' komutu çalıştırıldığı zaman (`/usr` dizininde bir dosya için) örnek ses tamamen bozulur. Bunun ardındaki sebep, veriyi gerçek zamanlı olmayan FIFO üstünde yazmamızdır.

Seste herhangi bir karışıklık meydana gelmesin diye biz şimdi bu süreç gerçek zamanlı nasıl çalışır, onu göreceğiz. Önce yukarıdaki programı bir gerçek zamanlı programa dönüştürelim. (Dosya adı `rtaudio.c` olsun.)

```
#include <rtl.h>
#include <pthread.h>
#include <rtl_fifo.h>
```

```
#include <time.h>

#define FIFO_NO 3
#define DELAY 30000
pthread_t thread;

void * sound_thread(int fd)
{
  int i;
  static char buf = 0;
  while (1)
  {
    for(i=0; i<DELAY; i++);
    buf = 0xff;
    rtf_put(FIFO_NO, &buf, 1);

    for(i=0;i<DELAY;i++);
    buf = 0x0;
    rtf_put(FIFO_NO, &buf, 1);
  }
  return 0;
}

int init_module(void)
{
  return pthread_create(&thread, NULL, sound_thread, NULL);
}

void cleanup_module(void)
{
  pthread_delete_np(thread);
}
```

Eğer hala yapmadıysanız **sound.o** modülünü çekirdeğe ekleyin. Yukarıdaki programı bir Makefile yazarak (daha önce söylediğimiz gibi) derleyin, böylece **rtaudio.o** modülü üretilir. Bu modülü eklemeden önce birkaç şey daha lazım. Yukarıdaki programın sonsuz döngü içinde çalıştığına dikkat edelim. Evreyi uyutmak ya da sonlandırmak için kodumuz olmadığı için evre çalışmasını durdurmayacaktır. Bu yolla sizin PC hoparlörünüz bu tonu üretmeye devam edecek ve siz başka bir şeyler yapmak için bilgisayarınızı yeniden başlatmak zorunda kalacaksınız.

Bu yüzden evrenin tonlar arasında gecikme yapmayı kendi kendine soran küçük kod değişikliğini (sadece `sound_thread()` işlevinde) yapalım.

```
void * sound_thread(int fd)
{
  static char buf = 0;
  pthread_make_periodic_np (pthread_self(), gethrtime(), 500000000);

  while (1)
  {
    pthread_wait_np();
    buf = (int)buf^0xff;
    rtf_put(FIFO_NO, &buf, 1);
  }
  return 0;

}
```

Artık '**rmmod**' komutu ile modülü yalnızca kaldırarak süreci sonlandırabiliriz.

Burada süreçlerarası iletişim için gerçek zamanlı FIFOların nasıl kullanılacağını gördük. Ayrıca RTLinux'a olan gerçek ihtiyaç yukarıdaki örnekten anlaşılabilir.

7. Sonrası

Bu belge RTLinux'ta programlamanın temellerini içermeye ayrılmıştır. Bir kez temelini anlarsanız bilginizi kendi kendinize geliştirmeniz zor değil. Böylece rtlinux kaynağı ile birlikte mevcut diğer bütün örnekleri inceleyip anlayabilirsiniz. Sonra modüller yazabilir ve onları deneyebilirsiniz. Modül programlama hakkında daha fazla bilgi için Ori Pomerantz tarafından yazılan Linux Çekirdek Modülü Geliştirme Kılavuzu[B6]na bakabilirsiniz.

Notlar

Belge içinde dipnotlar ve dış bağlantılar varsa, bunlarla ilgili bilgiler bulundukları sayfanın sonunda dipnot olarak verilmeyip, hepsi toplu olarak burada listelenmiş olacaktır.

(B1) ../howto/gpl.pdf

(B3) http://ftp.kernel.org/pub/linux/kernel/v2.2/linux-«2.2.18.tar.gz

(B4) http://ftp.kernel.org/pub/linux/kernel/v2.4/linux-«2.4.0-«test1.tar.gz

(B6) http://www.faqs.org/docs/kernel/

Bu dosya (rtlinux-howto.pdf), belgenin XML biçiminin TEXLive ve belgeler-xsl paketlerindeki araçlar kullanılarak PDF biçimine dönüştürülmesiyle elde edilmiştir.

23 Ocak 2007