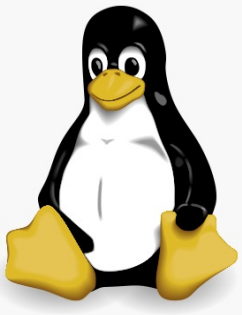
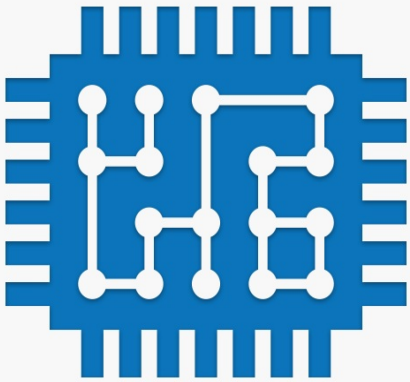


GÖMÜLÜ LINUX



Murat Demirten
Serkan Eser

İçindekiler

Giriş

Gömülü Linux kitabı, bu alanda gerçekleştirdiğimiz eğitim programlarımızda işlediğimiz konuların biraraya getirilmesi fikriyle oluştu.

Eğitim için hazırladığımız belgelerin kitap formatında düzenlenme ve güncelleme çalışmaları devam ediyor. Sürecin yarısına yaklaştığımızdan başkalarına da faydalı olabileceğini düşünerek, kitap içeriğini genel erişime açıyoruz.

Gömülü Linux kitabı bu alanda çalışmak isteyenlere başlangıç seviyesinde bilgiler sunmasının yanı sıra, uzun yıllardır çalışan kişiler için de öğretici unsurlar barındırmaktadır. Amacımız spesifik konulara girmek yerine öğrenme süreci açısından en fazla fayda üretecek başlıklara değinmek şeklinde olmakla birlikte, bu dengeyi tutturmanın kolay olmadığının da bilincindeyiz. Tam da bu noktada içerikle ilgili geri bildirimlerden memnuniyet duyacağımızı belirtiriz.

Sorularınız İçin

Kitapla ilgili öneri ve düşüncelerinizi *mdemirten* at *yh.com.tr* adresinden bizimle paylaşabilirsiniz.

Ek olarak teknik sorularınızı http://www.linux-tips.org sitesinde özellikle **Embedded** tartışma başlığı altında iletmeniz halinde, yanıtlamaya çalışacağımızı belirtmek isteriz. Katkı sağlayabilecek daha çok kişiye ulaşmak için site İngilizce olarak hazırlanmaktadır.

Yardımcı Kitap

Henüz genel erişime açılmamış olan Linux Sistem Programlama kitabımızın da okunmasını öneriyoruz. Mart ayı içerisinde ilk versiyonu açmış olmayı planlıyoruz.

Telif Hakkı

Teşekkür

İçeriğe katkılarından dolayı Serkan Eser'e teşekkürlerimi sunuyorum, katkılarının devamını bekliyoruz :)

Linux Çekirdeği

Linux, Unix felsefesi ve tasarım prensipleri doğrultusunda geliştirilmiş açık kaynak kodlu bir işletim sistemi çekirdeğidir. Çekirdeğin kaynak kodları GNU Genel Kamu Lisansı çerçevesinde özgürce dağıtılabilir, değiştirilebilir ve kullanılabilir.

Linux, 1991 yılında Finlandiyalı bir üniversite öğrencisi olan *Linus Benedict Torvalds* tarafından geliştirilmeye başlanmıştır. *Torvalds*, **25 Ağustos 1991**'de, *comp.os.minix* haber grubuna gönderdiği mesajda yeni bir işletim sistemi geliştirmekte olduğunu ve ilgilenen herkesin yardımını beklediğini yazdı. Daha sonra **17 Eylül 1991**'de Linux'un ilk sürümü olan **0.01**'i İnternet'te yayınladı. Kısa bir süre sonra, **5 EKim 1991**'de temel özellikleriyle beraber ilk resmi Linux sürümü olan 0.02'yi yayınladı. Linux ismi ilk olarak **0.02** versiyonunda geçmektedir.

Linus Torvalds, hâlen aktif olarak çekirdek geliştirme ekibinde olup halen en fazla kod gönderenler arasında ilk sıralarda yer almaktadır.

Kaynak Kod Boyutu

Linux çekirdeğinin prematüre hallerini es geçip **1994** yılında duyurulan **1.0** versiyonunu ele alacak olursak, proje yaklaşık **6 MB** yer kaplamaktaydı ve **34** dizin, **561** dosya içeriyordu. Kaynak kod satır sayısı ise tam olarak **165165** idi.

Mayıs **2015** tarihinde yayınlanan **4.0.4** versiyonunun kaynak kodları `tar.xz` arşivi haline getirildiğinde **78 MB** yer kaplamaktadır. Arşiv dosyası açıldığında ise **3159** dizin, **48948** dosya çıkmakta ve toplam boyut **641 MB** şeklinde olmaktadır. Proje yaklaşık **20 milyon** kod satırından oluşmaktadır.

Bu boyuttaki devasa bir yazılım projesinin belirli bir şirket çatısı olmaksızın binlerce insanın katkılarıyla yürütülebiliyor olması da başlı başına bir başarı noktası olarak değerlendirilmelidir. Sürecin yönetilebilir alt parçalara ayrılması ve sorumlulukların dağıtılması, karar alma mekanizmalarının işletimi, test, sürüm ve yol haritası planlama vb. gibi adreslenmesi gereken tüm başlıkların, bir an için *Enterprise* bakış açısıyla büyük bir şirket tarafından gerçekleştirildiğini ve halihazırda Linux'un desteklediği tüm platformları desteklediğini düşünelim:

- Acaba kaç yıl sürerdi?
- Nasıl bir maliyet ortaya çıkardı?
- Daha iyi olur muydu?

Desteklenen Mimariler

Güncel Linux çekirdeği, **29** farklı mimariyi (yazı ile yirmi dokuz) desteklemektedir.

Desteklenen mimariler Alpha, Analog Devices (Blackfin), ARM, Atmel AVR32, Axis Etrax CRIS, Texas TMS320 DSP Ailesi, Motorola 68K, Fujitsu FR-V, Qualcomm Hexagon, Hewlett-Packard PA-RISC, Renesas (Hitachi) H8, IBM 31bit System/390 ve 64bit Z Mainframe, Intel IA-64 Itanium, x86, Mitsubishi M32R, Xilinx Microblaze, MIPS, Panasonic MN103, Open RISC, Power (IBM), PowerPC, Sparc 32 Bit, Ultra Sparc 64 bit, SuperH, Synopsys, S+core, Tilera, Xtensa ve Unicore32 şeklindedir.

Gömülü Sistemlerdeki Kullanımı

Linux çekirdeği 2000'li yılların başından itibaren giderek artan hızda, gömülü sistemlerde kendine kullanım alanı bulmuştur.

Önceleri *Windows CE*, *VxWorks* vb. önemli rakipleri varken, ilerleyen yıllarda neredeyse standart haline gelmiş ve yoğun bir kullanım oranı yakalamıştır.

Günümüzde başta akıllı telefonlar olmak üzere, tüketici elektroniği dünyasında televizyon, setüstü-kutu, DSL modem, uydu alıcı, güvenlik duvarı, ağ depolama birimleri, el terminali, kontrol panel birimleri, navigasyon cihazları vb. pek çok farklı alanda kullanılmaktadır.

Geliştirme Süreci ve Versiyonlar

Linux çekirdeğinin geliştirme süreci tamamen gönüllü eforlara dayanmaktadır. Bu sebeple ticari dünyanın aksine sıkı tanımlanmış yol haritaları (roadmap) veya her X ayda bir çıkartılması beklenen sürüm hedefleri bulunmaz.

Bunun yerine teknik olarak izlenecek rehberler tariflenmiş olup bu çerçevede proje her türden katkıya sürekli açık durumdadır. Elbette bazı önemli geliştirmelerin geniş bir geliştirici kitlesiyle tartışılması, belli başlı kararlar alınması ve bazen bu kararlar neticesinde hatırı sayılır oranda yapısal değişikliklere gidildiği de olmaktadır.

Geliştirme Modeli

Yeni versiyonlar günümüzde halen bizzat Torvalds tarafından çıkartılmaktadır. Torvalds gönderilen yeni kodları ve çeşitli hatalara ilişkin yamaları bir araya getirerek sürümü oluşturur ve test süreçleri başlar. Bu noktadan sonra ortalama 10 hafta içerisinde yeni versiyon duyurulur. Bu şekildeki kernel versiyonları **vanilla** veya **mainline** şeklinde anılır.

Bazı Linux dağıtımları doğrudan **vanilla** kernel versiyonunu kullanırken, *Debian*, *RedHat* gibi bazı dağıtımlar ise çeşitli ek sürücüler, yamalar ve özellikleri içeren ayrı bir kernel versiyonu kullanırlar. Bu yöntemi izleyen dağıtımlarda kernel versiyonu çok sık güncellenmez, çoğu zaman dağıtımın bir sonraki versiyonu çıkana kadar aynı kernel versiyonu kullanılır (Örnek olarak Debian Wheezy versiyonunda kernel 3.2 kullanılırken Debian Jessie versiyonunda kernel 3.16 kullanılmaktadır). Bununla birlikte yeni kernel versiyonlarında düzeltilen önemli bir güvenlik açığı veya fonksiyonel düzenleme olursa, ilgili dağıtımların kernel paketleme ekipleri tarafından bu özellikler dağıtımla birlikte verilen kernel versiyonuna *backport* edilir.

Kaynak Kod Yönetim Sistemi

Linux projesi gibi büyük bir kaynak kod kümesi için kaynak kod yönetim sistemi de büyük önem taşımaktadır.

Linus Torvalds, *subversion*, *cvs* gibi merkezi yapıdaki çözümlere karşı hep mesafeli olmuştur. Merkezi yapıdaki çözümler birbiriyle yakın irtibat halinde çalışma imkanı olan ve az sayıda geliştiriciden oluşan projeler için kabul edilebilir olmakla birlikte, Linux projesindeki geliştirme modelinde çok dertler açmaktadır.

2002 yılında Torvalds önemli bir karar alarak, Linux kernel projesini teknik özellikleri açısından kendisini çok tatmin eden Bitkeeper platformu üzerinden yönetmeye karar verdi.

Bitkeeper zamanının ötesinde, dağıtık bir kaynak kod yönetim sistemi ve pek çok yeni özellik sunuyordu ancak ücretsiz bir yazılım değildi ve kaynak kodu kapalıydı. Bununla birlikte Linux projesine bedelsiz olarak hizmet veriyorlardı.

Linux gibi bir projenin kapalı kodlu bir kaynak kod yönetim sistemi üzerinden yönetilmesi zamanında epey tartışmalara yol açmıştır.

2005 yılında **Andrew Tridgell** Bitkeeper uygulama protokolünü *reverse-engineering* yoluyla çözmeye çalıştı. Buradaki amacı Bitkeeper ile entegre çalışabilecek araçlar üretmekti. Andrew Tridgell *reverse-engineering* konusunda çok başarılı bir araştırmacıydı ve özellikle Microsoft Windows ağlarındaki dosya paylaşımı için kullanılan **Server Message Block** protokolünü benzeri yöntemlerle çözmesiyle tanınıyordu. Aynı zamanda önemli Linux servislerinden **Samba**'nın yanı sıra **rsync**, **talloc** gibi projeleri geliştirmişti.

Andrew'in bu çalışması Bitkeeper'ın sahibi olan firmanın tepkisini çekti ve Linux projesi için verilen ücretsiz lisansı iptal ettiler.

Buna cevaben Linus Torvalds kolları sıvayıp dağıtık bir kaynak kod yönetim sisteminin geliştirmesine başladı. Aradan sadece 2 ay geçtikten sonra, bugün en çok kullanılan dağıtık kaynak kod yönetim sistemi olan Git ortaya çıkmış ve Linux kernel kaynak kodları için hizmet vermeye başlamıştı.

> Kişisel olarak **Git** projesini neredeyse Linux kadar önemli buluyorum. Bugün nasıl ki bir browser yazmak işletim sistemi yazmaktan çoğu zaman daha zor ise, Git'in sahip olduğu fonksiyonlar düşünüldüğünde Torvalds'ın yaptığı en önemli ve karmaşık işlerden biri olduğunu söyleyebiliriz.

Kod Sözdizim Rehberi

Linux çekirdek kaynak kodu üzerinde geliştirmeler yapmak istiyorsanız, `Documentation/CodingStyle` dosyasında yer alan yönergelerin takip edilmesinde fayda vardır. Milyonlarca satırdan oluşan bu büyük yazılım içerisinde başkalarıyla birlikte çalışma yapabilmek için genel bir yönergenin takip edilmesi, çekirdek geliştiricilerinin hayatını kolaylaştırır.

Bu noktada *Linus Torvalds*'a kulak verelim:

> Coding style is very personal, and I won't force my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.
>
> First off, I'd suggest printing out a copy of the GNU coding standards, and **NOT** read it. Burn them, it's a great symbolic gesture.

Linus özetle kod sözdizim tercihlerinin kişiye göre değişeceğini ama süreci yönetilebilir kılma adına bir şeyler yapılması gerekliliğinden bahsediyor. Devamında da bunu destekler biçimde, **GNU Sözdizim Rehberi** dokümanının çıktısını alıp okumadan yakmamızı salık veriyor.

Girinti ve Hizalama

TAB için **8** boşluk karakteri kullanılır. Geçmişte bunu 4'e hatta 2'ye indirmek üzerine çeşitli tartışmalar döndü. Ancak Torvalds bu tartışmaları **π** sayısını **3** olarak almak isteyenlerdekilere benzetiyor ve faydasız buluyor.

8 karakter gibi başlangıçta fazla gibi görünen boşluk sayısı, günde 20 saat çekirdek kodu inceleyen biri için bir bloğun nerede başlayıp nerede bittiğini kolay göstermesi açısından önemlidir.

8 karakterlik TAB kullanımı sonucudan kodun çok fazla sağ tarafa doğru ilerlediği için 80 karakterlik konsol ekranında okunmayı zorlaştırdığından şikayet edenlere Torvalds, *zaten 3 seviyeden fazla girinti yapmanız gerektiyse bir şeyleri yanlış yapıyorsunuz demektir, en iyisi oturup kodu düzeltin şeklinde* öneride bulunuyor.

Tahmin edebileceğiniz üzere çekirdek kodu içerisinde 3 seviyeden fazla girintiye sahip olan pek çok kod bulunmaktadır. Ancak bu boyuttaki bir yazılım projesi için girinti ve blok kullanımının görece minimum düzeyde tutulduğunu söyleyebiliriz. Torvalds ile ters düşmemek için özellikle yeni başlayanların söz dinlemesinde fayda var.

Kıvrık Parantez

Kıvrık parantezlerin kullanımında **Kernighan & Ritchie** metodu tercih edilmiştir.

Fonksiyon içerisinde kıvrık parantez koşullu ifadenin hemen bitiminde açılıp, blok sonunda ayrı bir satırda kapatılır:

```
if (x == condition) {
        // code here
}
```

Fonksiyon tanımlarında ise kıvrık parantez fonksiyon tanımını takip eden satırda açılıp, fonksiyon bitiminde ayrı bir satrıda kapatılır:

```
int my_super_function(int a, int b)
{
        // code here
}
```

do .. **while** döngülerinde aşağıdaki yapı kullanılır:

```
do {
        // code here
} while (x == y);
```

if .. **else** deyimlerinde aşağıdaki yapı kullanılır:

```
if (x < y) {
        ...
} else if (x > y) {
        ...
} else {
        ...
}
```

İsimlendirme Kuralları

C dili genel olarak *Spartan Programlama* mantığı etrafında bir kullanım kültürü oluşturmuştur. Minimalist, az yer kaplayan ama halen daha büyük oranda anlaşılabilir bir kod yazımı tercih edilir.

Örneğin bir **Pascal** programcısının `ThisVariableIsATemporaryCounter` şeklinde isimlendirdiği bir değişkenin **C** programcısı tarafından basitçe `tmp` şeklinde isimlendirileceğini tahmin edebiliriz: yazması kolay, okunurluğu da tamamen öldürmüyor, büyük oranda anlaşılıyor.

Bununla birlikte global değişkenlerde açıklayıcı bir isim kullanması şart koşulmudur. Genel olarak global değişkenlerin **sadece ve sadece zaruri olduğunda** kullanılması ve kullanıldığında derdini net bir şekilde anlatabilecek bir isimlendirme yapılması istenmektedir.

Fonksiyon isimlendirmelerinde de işlevini açıkça ifade eden bir yapı kullanılması istenir. Örneğin aktif olan kullanıcıların sayısını bulan bir fonksiyon için `cntuser()` gibi bir isim kabul edilmez, bunun yerine `count_active_users()` gibi bir isim kullanılmalıdır.

Değişken ve fonksiyon isimlendirmelerinde **Macar Notasyonu** kullanımı kesinlikle istenmez. Derleyici zaten değişken veya fonksiyonun tipini bildiğine göre, bu şekildeki kullanım kodu okuyan beyinlere hasar vermek dışında bir fayda vermeyecektir. Torvalds bu noktada *Microsoft* platformlarında bu kadar hata olmasına şaşmamalı şeklinde düşünüyor. Katılmak mümkün, bu notasyonu literatürden çıkarmak çok yerinde olurdu.

Yerel değişkenlerin de kısa ve tam yerinde kullanılması istenir. Örneğin bir döngü içerisinde sayaç tutmanız gerekiyorsa, bunun için `i` ismini kullanın. `loop_counter` şeklinde açıklayıcı bir isim fazladan pek bir şey kazandırmayacaktır.

Typedef Kullanımı

Bu konudaki kural basit: çok istisnai bir durum olmadıkça typedef **kullanmayın**.

```
vps_t a;
```

şeklindeki bir tanım açıklayıcı değil. Onun yerine

```
struct virtual_container a;
```

şeklindeki gibi, `struct` anahtar kelimesi ile birlikte daha açıklayıcı yapı ismini kullanın.

Genel bir kural olarak, elemanlarına erişmek için *accessor* metodları kullanmanız gereken opak veri tiplerinin dışında **typedef** kullanımından kaçının.

Fonksiyonlar

Fonksiyonlar sadece tek bir işi iyi yapacak şekilde ve kısa olmalıdır. Kısalıktan kasıt, 80x24'lük ANSI konsol ekranını baz aldığımızda bir fonksiyonun maksimum 2 ekran boyutu kadar yer kaplaması gerektiğidir.

Eğer fonksiyon içerisinde yapılan iş basit ve temiz olmasına rağmen, örneğin bir `switch..case` bloğu içerisinde çok sayıda değeri kontrol etmesi gerekiyorsa ve bu nedenle 2 ekran sınırını aşıyorsa, bu durum hoş karşılanır, sorun edilmez.

Fonksiyonların değerlendirilmesindeki bir diğer kriter de yerel değişken kullanımıdır. Maksimum 5-10 arasında yerel değişken kullanımı önerilmektedir. İnsan beyni genelde 7 farklı şeyi takip edebilir. Bu sayıların aşılması durumunda bir şeyleri yanlış yaptığınızı düşünmeniz, yeni bir tasarım yapmanız veya mevcut fonksiyonu, yardımcı ek fonksiyonlara bölmeniz beklenir.

Fonksiyonların bitiminde 1 adet boş satır bırakmanız istenmektedir. Böylece gözle daha kolay takip edilebilecektir. Ancak bu durumun istisnası olarak, eğer fonksiyonunuz kernel içerisinde başka yerlerden de kullanılacağı için *export* edilecekse, ilgili `EXPORT` makrosunu hemen fonksiyonun bitiminde tanımlamanız ve sonrasında boş satır bırakmanız beklenmektedir:

```
int system_is_up(void)
{
        return system_state == SYSTEM_RUNNING;
}
EXPORT_SYMBOL(system_is_up);
```

Ek olarak başlık dosyalarında fonksiyonlarınızı tanımlarken okunabilirliği artırmak için, parametre bloğunda sadece değişkene ait veri tipini değil, değişken ismini de belirtmeniz beklenir.

Goto Kullanımı

Kernel kodu ile uğraşırken yerinde `goto` kullanımı okunabilirliği ve verimliliği artırır.

Bununla birlikte etiket isimlendirmelerine özen göstermeniz beklenir. *GW-BASIC* alışkanlıklarından gelen `err1:` , `err2:` gibi etiketler kullanmamalısınız. Onun yerine `goto` sonrasında ne olacağını ve nereyi etkilediğini net bir şekilde belirtmeniz tavsiye edilmektedir. Örneğin `out_buffer:` şeklindeki bir etiket, `goto` işlemi sonrası fonksiyondan çıkılacağını ve `buffer` ile ilgil olduğunu göstermektedir.

```
int fun(int a)
{
        int result = 0;
        char *buffer;

        buffer = kmalloc(SIZE, GFP_KERNEL);
        if (!buffer)
                return -ENOMEM;

        if (condition1) {
                while (loop1) {
                    ...
                }
                result = 1;
                goto out_buffer;
        }
        ...
out_buffer:
        kfree(buffer);
        return result;
}
```

Yorum Satırları

Yorum satırlarını kullanmak güzeldir ama fazla kullanıldığında tehlikeli olabilir. Genel prensip olarak yorum satırlarında fonksiyonunuzun nasıl çalıştığını değil, basitçe ne yaptığını anlatmanız yeterlidir.

Yorum satırları fonksiyonun tanımının hemen üzerinde yer almalı, fonksiyon içerisinde yorumlara yer verilmemelidir.

Yorum satırları için Linux kernel kodunda **C89** `/* ... */` sitili kullanılır. Tek satırlık **C99** stili `// ...` yorumlara izin verilmez.

Fonksiyon başlangıcındaki örnek bir yorum bölümü şu şekilde olabilir:

```
    /*
     * This is the preferred style for multi-line
     * comments in the Linux kernel source code.
     * Please use it consistently.
     *
     * Description:  A column of asterisks on the left side,
     * with beginning and ending almost-blank lines.
     */
```

Kconfig Konfigürasyon Dosyaları

Kconfig konfigürasyon dosyalarındaki sözdizim kuralları biraz farklıdır.

`config` anahtar kelimesi altındaki satırlar **TAB** ile girinti yapılarak yazılır. `help` anahtar kelimesini takip eden yardım satırları ise buna ek olarak, **2** adet ek boşluk kullanılarak yazılmaktadır:

```
config NET_FC
    bool "Fibre Channel driver support"
    depends on SCSI && PCI
    help
      Fibre Channel is a high speed serial protocol mainly used to connect
      large storage devices to the computer; it is compatible with and
      intended to replace SCSI.
```

Konfigürasyon Süreci ve Kbuild/Kconfig Sistemi

Konfigürasyon sürecinin amacı, derleme işlemi sonrasında oluşacak olan kernel imajının sahip olacağı özelliklerin belirlenmesidir. Bu işlem tamamlandıktan sonra, belirlenen özellikler doğrultusunda derleme işlemi yürütülebilir.

Konfigürasyon

Konfigürasyon süreci oldukça zaman alıcı bir işlemdir. Derlenecek olan kernel imajının sahip olacağı özellikleri tek tek seçmeye çalışmak saatlerce sürebilir. Eğer karşımıza soru olarak çıkartılan seçeneklerin pek çoğu hakkında ön bilgimiz yoksa -ki çoğunlukla öyle olacaktır-, seçenekler hakkında ek bilgi alma ve karar verme sürecini de hesabımıza eklediğimizde bu süreç günlerce sürebilir.

Elbette konfigürasyon için bir kaç gün zaman ayırmayacağız ancak Gömülü Linux veya genel olarak Linux alanında bilginizi bir seviye daha ileriye taşımanın yollarından birinin, bu sürece geniş bir zaman ayırıp kernel içerisindeki teknolojileri incelemek ve ilgili dokümanları ek referanslarıyla birlikte okumaktan geçtiğini unutmayın.

Konfigürasyon işleminin sonucunda `.config` adında bir adet dosya üretilecektir. Dolayısıyla eğer kullanacağınız sistem için daha önceden üretilmiş bir konfigürasyon dosyası var ise onu baz alarak veya doğrudan kullanarak da işlem yapabilirsiniz.

Özellikle gömülü sistemler söz konusu olduğunda kullanacağınız board için üretici tarafından önceden hazırlanmış bir konfigürasyon dosyası genellikle bulunur. Bu konfigürasyon dosyasını kullanarak board üreticinizin sağladığı veya önerdiği kernel kaynak kodlarını derlediğinizde, kullanacağınız board ile uyumlu bir kernel imajı çıkmasını beklersiniz.

Hazır bir konfigürasyon dosyası gelmiş olsa dahi, bu dosyayı baz alarak konfigürasyon sürecinin üzerinden geçilmesi ve konfigürasyon dosyasının biraz daha iyileştirilmesi önerilir. Üreticiden gelen konfigürasyon dosyaları genellikle ürünü kullanacağınız senaryoda esasen ihtiyaç duymayacağınız pek çok bileşene dair desteği de içerir. Bu dosyayı iyileştirmeye çalışmak hem ürünü daha iyi tanımanızı hem de daha küçük bir kernel imajı elde etmenizi sağlayacaktır.

Kbuild / Kconfig Sistemi

700 MB'a yaklaşan kaynak kod büyüklüğü ve onbinlerce derleme seçeneği opsiyonu, Linux kernel derleme sürecinin yönetimine dair de ek bir sistem geliştirilmesi ihtiyacını doğurmuştur.

Kbuild sistemi ve beraberindeki **Kconfig** dili bu amaçla geliştirilmiştir. Konfigürasyon sürecine genel olarak baktığımızda, çok sayıda seçenek ve birbirlerine olan bağımlılık kurallarından oluşan bir veritabanı içerisinden, kendi içinde tutarlı bir alt küme oluşturma işlemi olarak görebiliriz.

Aşağıda bir kaç seçenek kümesine dair hiyerarşi gösterilmektedir:

```
+- Code maturity level options
|  +- Prompt for development and/or incomplete code/drivers
+- General setup
|  +- Networking support
|  +- System V IPC
|  +- BSD Process Accounting
|  +- Sysctl support
+- Loadable module support
|  +- Enable loadable module support
|     +- Set version information on all module symbols
|     +- Kernel module loader
+- ...
```

Örnekteki her bir eleman, opsiyonel olarak bağımlılık kuralları tanımlamış olabilir. Örneğin **PCI** veriyolu üzerinde çalışacak bir ethernet kartı doğal olarak **PCI Veriyolu Desteği**'ne bağımlı olduğundan, bu şekilde bir bağımlılık kuralının tanımlanması halinde, PCI desteği seçilmediğinde konfigürasyon sürecinin ilerleyen aşamalarında ilgili ethernet kartlarının karşımıza bir seçenek olarak gelmemesi gerekir. Eğer süreç bu şekilde işlemez ise, birbiriyle çelişen konfigürasyon sembollerini seçmemiz işten bile değildir.

Bağımlılık kurallarını doğru biçimde tüm sistem genelinde uyguladığımızda, bu kadar geniş bir seçim kümesi içerisinden her zaman anlamlı bir küme oluşturma imkanına sahip oluruz.

Kconfig dili oldukça basittir ve öncesinde çok fazla bilgi sahibi olmasanız dahi, yeterince örnek içeren bir Kconfig dosyasına bakıldığında kullanımı kendiliğinden anlaşılabilir.

> `Kbuild` sistemini anlatan dokümanlara, kernel kaynak kodu altında `Documentation/kbuild` dizininde erişebilirsiniz.

Örnek Kconfig Dosyası

Kısa bir örnekle başlamak için kernel kaynak kodu içerisindeki `drivers/android/Kconfig` dosyasını inceleyelim:

```
menu "Android"

config ANDROID
        bool "Android Drivers"
        ---help---
          Enable support for various drivers needed on the Android platform

if ANDROID

config ANDROID_BINDER_IPC
        bool "Android Binder IPC Driver"
        depends on MMU
        default n
        ---help---
          Binder is used in Android for both communication between processes,
          and remote method invocation.

          This means one Android process can call a method/routine in another
          Android process, using Binder to identify, invoke and pass arguments
          between said processes.

config ANDROID_BINDER_IPC_32BIT
        bool
        depends on !64BIT && ANDROID_BINDER_IPC
        default y
        ---help---
          The Binder API has been changed to support both 32 and 64bit
          applications in a mixed environment.

          Enable this to support an old 32-bit Android user-space (v4.4 and
          earlier).

          Note that enabling this will break newer Android user-space.

endif # if ANDROID

endmenu
```

Dosya öncelikle bir `menu` tanımıyla başlıyor. Bu sayede ileride karşımıza ismi "Android" olan bir menü çıkacağını anlıyoruz.

Sonraki adımda `config` anahtar kelimesi ile `ANDROID` sembolü tanımlanmış. Sembolün tipi `bool` olarak belirtildiğinde var/yok şeklinde ikili bir seçim içerdiğini anlıyoruz. Sembolün tipinden sonra gelen açıklama satırında ise ilgili sembolün hangi metinle karşımıza çıkacağını göstermektedir: Android Drivers

Hemen altında `---help---` ile başlayan bölümde tahmin edebileceğiniz üzere, bu sembolle ilgili yardım alınmak istendiğinde gösterilecek olan metin yer almaktadır.

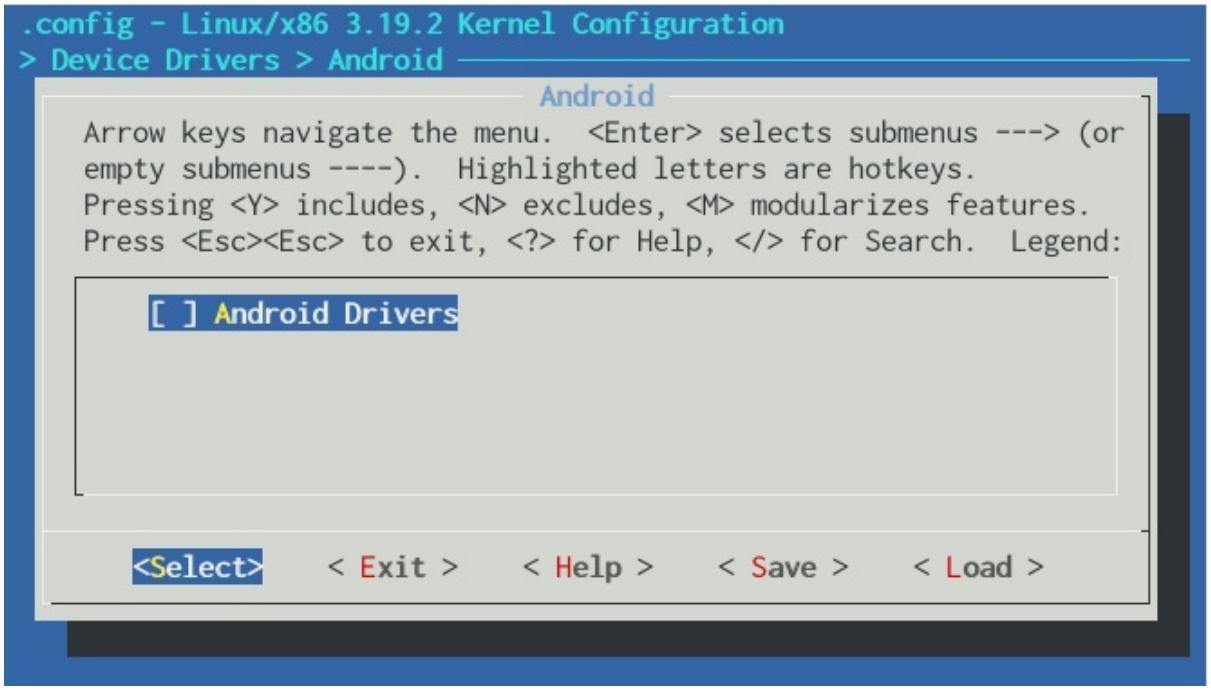
Sonrasında dosyanın geri kalanın bir `if ANDROID` bloğu içerisine alındığını görmekteyiz. Bunun anlamı eğer üst bölümde tanımlanan `bool` tipindeki `ANDROID` sembolü seçilirse bu konfigürasyon seçeneklerinin aktif olacağı ve kullanıcıya gösterileceği, aksi takdirde bu seçeneklerin hiç gösterilmeyeceğidir.

Bir sonraki sembolümüz gene `bool` veri tipinde `ANDROID_BINDER_IPC` şeklindedir. Bu sembolün `depends on` anahtar kelimesi kullanılarak `MMU` sembolüne bağımlı olduğu belirtilmiştir. Dolayısıyla konfigürasyon sürecinin diğer adımlarında `MMU` sembolü seçilmedi ise bu özellik de seçilemeyecektir. Ek olarak `default` anahtar kelimesiyle öntanımlı değerinin hayır anlamında `n` şeklinde olduğu belirtilmiştir.

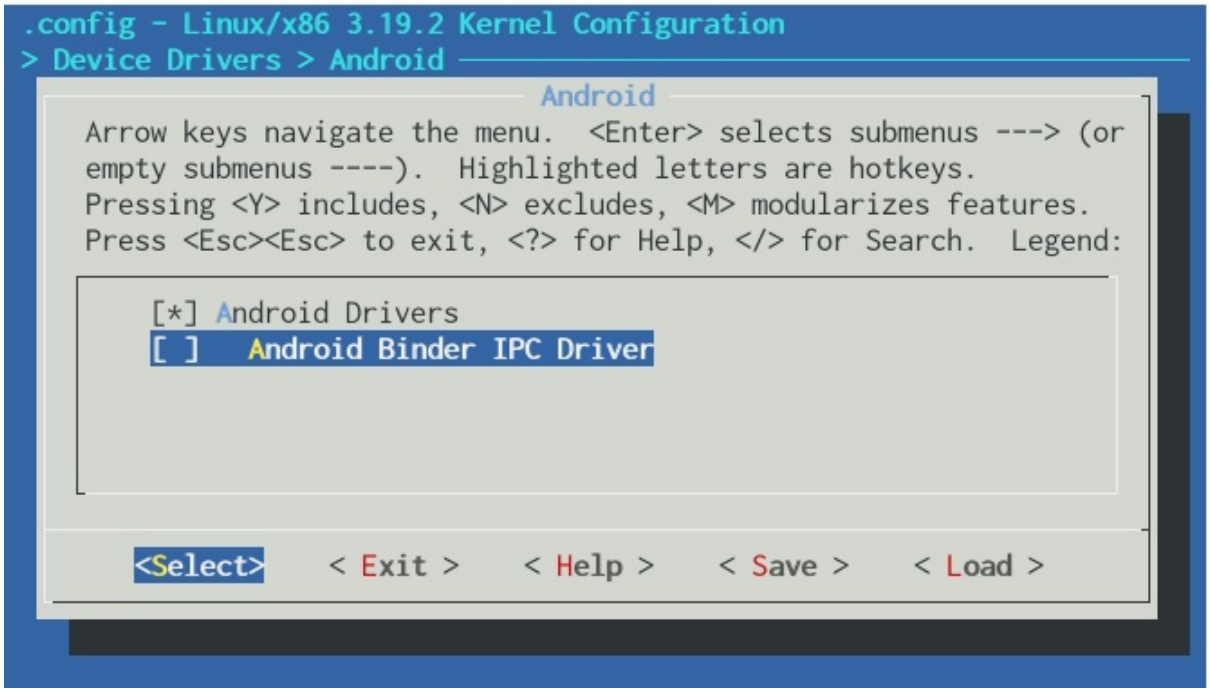
Son sembolümüz gene `bool` tipinde `ANDROID_BINDER_IPC_32BIT` 'dir. Bu sembol için öntanımlı değer evet anlamında `y` olarak verilmiştir. Bununla birlikte sembolün seçilebilmesi için verilen bağımlılık kurallarında, mimarinin 64 bit olamayacağı (`!64BIT`) ve `ANDROID_BINDER_IPC` sembolünün de seçilmiş olması şartlarının aynı anda sağlanması istenmiştir.

Örnek dosyamızın son kısımlarında açılan `if` bloğunun `endif` ile, en başta açılan `menu` bloğunun ise `endmenu` ile kapatıldığını görmekteyiz.

Aşağıdaki örneklerde bu menü için 64bit aktif iken `ANDROID` sembolünün değerleri için oluşan çıktıları görebilirsiniz:



`ANDROID` sembolü seçili değilken



`ANDROID` sembolü seçili durumdayken

Kconfig Sembol Tipleri

Yukarıda Kconfig sistematiği içerisinde kullanabildiğimiz `bool` tipini örnekle görmüştük. Kconfig içerisinde kullanılabilecek tiplerin tamamı ise aşağıdaki tabloda yer almaktadır.

Sembol_Tipi	Açıklama
bool	Evet/Hayır türünden boolean seçim imkanı verir
tristate	`bool` tipine 3. bir durumun daha eklenmesiyle bir özelliğin modül olarak derlenebilmesi için gereken seçimin de yapılabilmesine imkan verir. Bu sayede örneğin `FAT` dosya sistemi desteği için hayır (**n**), evet (**y**) ve modül (**m**) şeklinde 3 alternatiften birinin belirtilebilmesi mümkün olur. En çok kullanılan sembol tipi budur.
string	Değer olarak herhangi bir metin alır
hex	16'lık düzende nümerik veri alır
int	10'luk düzende nümerik veri alır

Nümerik sembol tipleri ayrıca opsiyonel olarak bir aralık belirtilmesine olanak veren **range** deyimine de sahiptir. Aşağıdaki örnekte `PM_WAKELOCKS_LIMIT` sembolünün `int` tipinde tanımlandığı, öntanımlı değerinin 100 ve geçerli değer aralığının 0 - 100000 arasında olduğu belirtilmektedir.

```
config PM_WAKELOCKS_LIMIT
        int "Maximum number of user space wakeup sources (0 = no limit)"
        range 0 100000
        default 100
        depends on PM_WAKELOCKS
```

`Kconfig` dosyalarında girinti seviyeleri (*indentation*) önemlidir, mevcut örneklere baktığınızda bunu kolaylıkla anlayabilirsiniz. Ana konumuzdan uzaklaşmamak adına diğer detaylara girmeyip burada sonlandırıyoruz.

> **Not**: Kbuild ve Kconfig sistemi özel olarak Linux kernel projesi için geliştirilmiştir. Ancak eğer sizin de birbirini etkileyen bileşenler arasında seçim yapmak suretiyle derleme yapmanız gereken büyük bir projeniz varsa, bu sistemi aynen kullanabilirsiniz. Keza `busybox` gibi çeşitli projelerde de Kbuild sistemi kullanılmaktadır.

Konfigürasyon Dosyası Formatı

Üretmeye çalışacağımız `.config` dosyasının formatı aşağıdaki gibidir:

```
...
# CONFIG_VIDEO_FIXED_MINOR_RANGES is not set
CONFIG_DVB_CORE=m
CONFIG_DVB_NET=y
CONFIG_TTPCI_EEPROM=m
CONFIG_DVB_MAX_ADAPTERS=8
CONFIG_DVB_DYNAMIC_MINORS=y
...
```

Yukarıda görülen sembol isimleri, `Kconfig` sistemi içerisindeki sembollerin önüne `CONFIG_` öneki getirilerek oluşturulur.

Dosya içerisinde 4 farklı türde satır olduğunu göreceğiz. Bunlar:

- **Yorum Satırları**: Diyez karakteri ile başlayan satırlar yorum satırı olarak nitelenir ve değerlendirmeye alınmazlar.
- **=y İle Biten Satırlar**: İlgili sembolün nitelediği bileşenlerin, oluşacak nihai kernel imajı içerisinde yer almasının istendiğini belirtir.
- **=m İle Biten Satırlar**: İlgili sembolün nitelediği bileşenlerin, oluşacak nihai kernel imajı içerisine konulmamasını ancak ayrı bir modül olarak derlenmesinin istendiğini belirtir.
- **..XXX=8, YYY=Z Şeklindeki Satırlar**: Bu şekildeki satırlar genel olarak derleme sürecinde bazı parametrik işlemlerin yapılabilmesine olanak sağlamak amacıyla kullanılır, örneğin yukarıdaki örneğimizde maksimum DVB adaptör kart sayısının 8 ile

sınırlandırıldığını görmekteyiz. Burada tanımlı değer Makefile sistematikleri üzerinden derleme sürecine doğru şekilde aktarıldığında, ilgili DVB kaynak kodları derlenirken bu limit dikkate alınacaktır.

Konfigürasyon Araçları

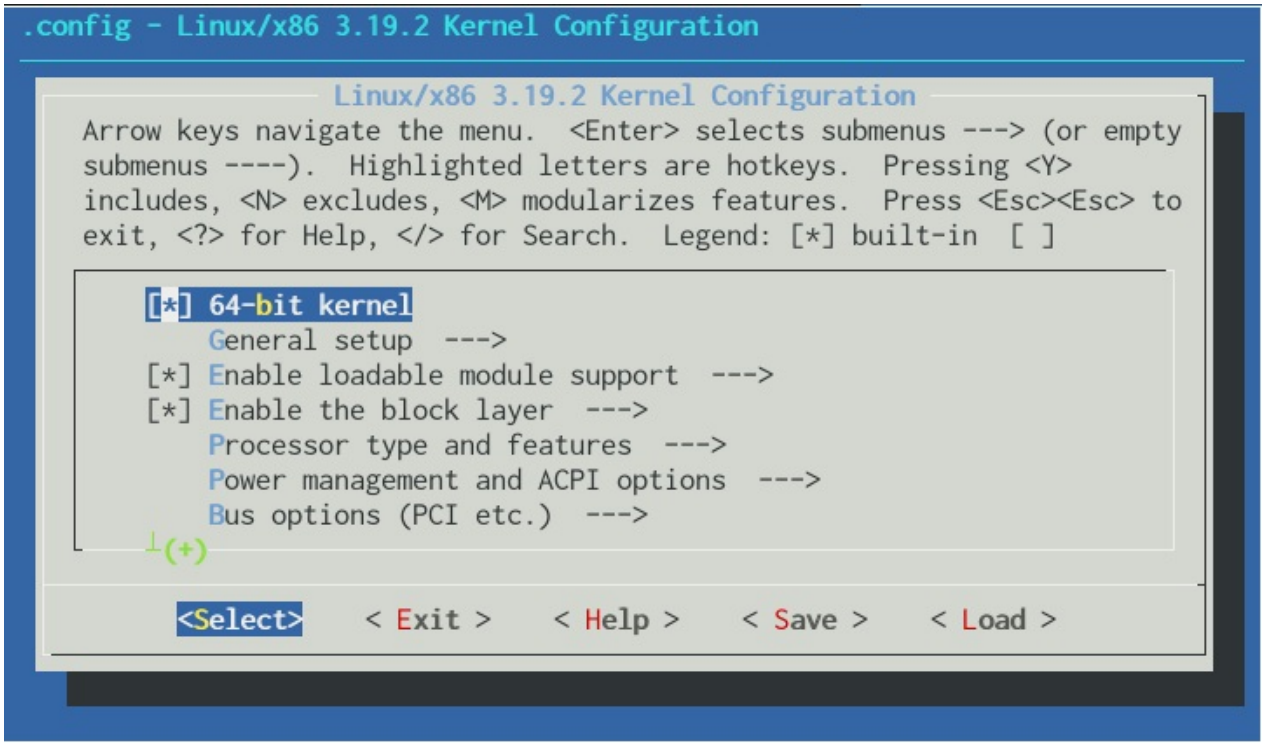
Konfigürasyon dosyasını elle oluşturmak pratikte imkansız ve zaten gereksizdir. `Kconfig` dosyalarını analiz edip bize seçim yapma noktasında yardımcı olacak çeşitli araçlar mevcuttur. Bu araçlar da kaynak kod şeklinde Linux kernel projesi içerisindeki `scripts/kconfig` dizininde yer alır. Kullanılabilmeleri için öncelikle derlenmeleri gereklidir.

Bu dizinde birden fazla yardımcı araç bulunur. Her biri aynı işi hedefleyen, farklı arayüzlere sahip araçlardır.

Yöntem	Açıklama	İhtiyaç Duyulan Kütüphaneler
xconfig	Qt-4 backend	qt4-dev
gconfig	Gtk-2 backend	gtk-2-dev, libglade-2-dev
nconfig	Ncurses backend	libncurses-dev
menuconfig	curses backend	libncurses-dev

Konfigürasyona başlamak için bu araçlardan birini kullanabilirsiniz. En sık kullanılan **menuconfig** aracını aşağıdaki şekilde başlatabilirsiniz:

```
$ make menuconfig
```



menuconfig aracı içerisinde yön tuşları ile gezebilir, **Help** bölümünden seçili başlık hakkında yardım alabilirsiniz. Belirli bir sembol için arama yapmak istediğinizde **/** kısayolunu kullanabilirsiniz (vi editöründeki ile aynı şekilde)

Konfigürasyon işlemi tamamlandıktan sonra tüm menülerden çıkış yapabilirsiniz. Uygulama sonlanmadan hemen önce yapılan konfigürasyonu kaydetmek isteyip istemediğiniz sorulacaktır.

Derleme ve Çapraz Derleme

Konfigürasyon süreci tamamlandıktan ve herhangi bir yolla `.config` dosyası kernel kaynak kodlarının bulunduğu ana dizinde oluşturulduktan sonra derleme aşamasına geçilir.

Derleme işlemi sonucunda işlemin yapıldığı sistem için çalışacak *native* kernel imajı elde edilebileceği gibi, çapraz derleme (*cross-compiling*) metoduyla farklı bir mimari için imaj üretimi de sağlanabilir.

Normal derleme işlemi için `make` komutunu vermeniz yeterlidir. Öncelikle ana dizinde yer alan `Makefile` dosyası okunacak ve `.config` dosyasında yapılan seçimler **export** edilerek derleme sürecinin ilerleyen aşamalarında kullanılabilir olması sağlanacaktır.

Kernel imajı sıkıştırılmadığında biraz büyükçe olduğundan her zaman sıkıştırılmış formda kullanılır. Bunun için kernel imajı içerisinde kendi kendini **decompress** edebilecek bir rutin eklenir ve kernel imajı sıkıştırılmış haliyle kullanılır.

Bu süreci otomatik hale getirmek için **PC** mimarisi için yapılan derlemeler için `bzImage` hedefi kullanılmaktadır:

```
$ make bzImage
```

Komut işletilirken build işleminin çıktıları aşağıdaki gibi olacaktır:

```
...
CC      arch/arm/kernel/topology.o
CC      arch/arm/kernel/io.o
AS      arch/arm/kernel/debug.o
CC      arch/arm/kernel/early_printk.o
LD      arch/arm/kernel/built-in.o
AS      arch/arm/kernel/head.o
LDS     arch/arm/kernel/vmlinux.lds
...
```

Bu özet çıktı yerine çalıştırılan derleyici, linker vb. tüm aldığı parametreleri görmek isterseniz, `make` komutunun sonuna `V=1` parametresini ekleyebilirsiniz (verbosity):

```
$ make bzImage V=1
...
gcc -Wp,-MD,arch/x86/crypto/.aes-x86_64-asm_64.o.d  -nostdinc -isystem /usr/lib/gcc/x8
6_64-linux-gnu/4.9/include
-I./arch/x86/include -Iarch/x86/include/generated/uapi -Iarch/x86/include/generated
-Iinclude -I./arch/x86/include/uapi -Iarch/x86/include/generated/uapi
-I./include/uapi -Iinclude/generated/uapi -include ./include/linux/kconfig.h
-D__KERNEL__  -D__ASSEMBLY__ -m64 -DCONFIG_X86_X32_ABI -DCONFIG_AS_CFI=1
-DCONFIG_AS_CFI_SIGNAL_FRAME=1 -DCONFIG_AS_CFI_SECTIONS=1 -DCONFIG_AS_FXSAVEQ=1
-DCONFIG_AS_CRC32=1 -DCONFIG_AS_AVX=1 -DCONFIG_AS_AVX2=1  -Wa,-gdwarf-2  -mfentry
-DCC_USING_FENTRY -DMODULE -c -o arch/x86/crypto/aes-x86_64-asm_64.o
arch/x86/crypto/aes-x86_64-asm_64.S
...
```

Derleme işlemi bittiğinde `bzImage` dosyası oluşur. Bu dosya `vmlinux` kernel imajının sıkıştırılmış halini ve decompress rutinini içerir.

Derleme işlemi çıktısının son bölümü aşağıdaki gibi bir çıktı üretecektir:

```
  LINK    vmlinux
  LD      vmlinux.o
  MODPOST vmlinux.o
  GEN     .version
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  CC      init/version.o
  LD      init/built-in.o
  KSYM    .tmp_kallsyms1.o
  KSYM    .tmp_kallsyms2.o
  LD      vmlinux
  SORTEX  vmlinux
  SYSMAP  System.map
  CC      arch/x86/boot/a20.o
  AS      arch/x86/boot/bioscall.o
  CC      arch/x86/boot/cmdline.o
  AS      arch/x86/boot/copy.o
  HOSTCC  arch/x86/boot/mkcpustr
  CC      arch/x86/boot/cpuflags.o
  CC      arch/x86/boot/cpucheck.o
  CC      arch/x86/boot/early_serial_console.o
  CC      arch/x86/boot/edd.o
  VOFFSET arch/x86/boot/voffset.h
  CC      arch/x86/boot/main.o
  CC      arch/x86/boot/mca.o
  LDS     arch/x86/boot/compressed/vmlinux.lds
  AS      arch/x86/boot/compressed/head_64.o
  CC      arch/x86/boot/compressed/misc.o
  CC      arch/x86/boot/memory.o
  CC      arch/x86/boot/compressed/string.o
  CC      arch/x86/boot/compressed/cmdline.o
```

```
  OBJCOPY arch/x86/boot/compressed/vmlinux.bin
  HOSTCC  arch/x86/boot/compressed/mkpiggy
  CC      arch/x86/boot/compressed/cpuflags.o
  CC      arch/x86/boot/compressed/early_serial_console.o
  CC      arch/x86/boot/pm.o
  AS      arch/x86/boot/pmjump.o
  CC      arch/x86/boot/printf.o
  CC      arch/x86/boot/compressed/eboot.o
  AS      arch/x86/boot/compressed/efi_stub_64.o
  CC      arch/x86/boot/regs.o
  CC      arch/x86/boot/string.o
  CC      arch/x86/boot/tty.o
  CC      arch/x86/boot/video.o
  CC      arch/x86/boot/video-mode.o
  CC      arch/x86/boot/version.o
  CC      arch/x86/boot/video-vga.o
  XZKERN  arch/x86/boot/compressed/vmlinux.bin.xz
  CC      arch/x86/boot/video-vesa.o
  CC      arch/x86/boot/video-bios.o
  HOSTCC  arch/x86/boot/tools/build
  CPUSTR  arch/x86/boot/cpustr.h
  CC      arch/x86/boot/cpu.o
  MKPIGGY arch/x86/boot/compressed/piggy.S
  AS      arch/x86/boot/compressed/piggy.o
  LD      arch/x86/boot/compressed/vmlinux
  ZOFFSET arch/x86/boot/zoffset.h
  OBJCOPY arch/x86/boot/vmlinux.bin
  AS      arch/x86/boot/header.o
  LD      arch/x86/boot/setup.elf
  OBJCOPY arch/x86/boot/setup.bin
  BUILD   arch/x86/boot/bzImage
Setup is 17388 bytes (padded to 17408 bytes).
System is 3131 kB
CRC 43b226da
Kernel: arch/x86/boot/bzImage is ready  (#1)
```

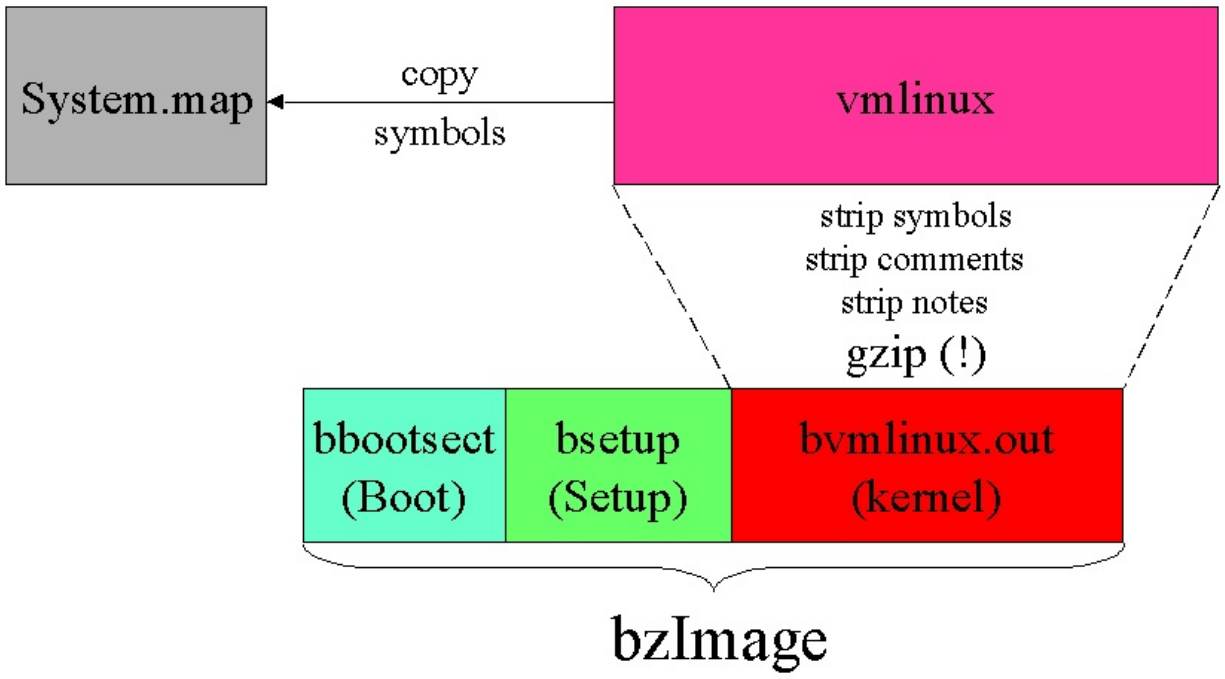
İşlem bitiminde kullanacağımız kernel imaj dosyası `arch/x86/boot` dizini altında oluşur.

> Diğer mimariler için derleme yaptığımızda `arch/<mimari>/boot` dizini altında oluşacaktır.

Derleme sürecini yönettiğimiz kernel ana dizininde ise `vmlinux` adında bir dosya oluştuğunda dikkat ediniz. Bu dosya kernel imajının sıkıştırılmamış halidir:

```
$ ls -lh vmlinux
-rwxr-xr-x 1 demirten demirten 120M May 16 07:58 vmlinux

$ file vmlinux
vmlinux: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
BuildID[sha1]=f6c3556c7b74f9b300b03275831a4b716d18ee9e, not stripped

$ ls -lh arch/x86/boot/bzImage
-rw-r--r-- 1 demirten demirten 3.1M May 16 07:58 arch/x86/boot/bzImage
```

Görüldüğü üzere sıkıştırılmamış hali **120 MB** iken, sıkıştırılmış hali **3.1 MB** seviyelerindedir.



Kernel imajı üretildikten sonra modüllerin derlenmesi ve uygun dizin yapısı ile sistemde oluşturulması gerekir. Bu işlemler için `modules` ve `modules_install` make hedefleri kullanılır:

```
$ make modules
$ make modules_install
```

`modules_install` hedefi, modüllere ait dizin yapısını öntanımlı olarak derlemenin yapıldığı sistemde, `/lib/modules/<KERNEL-VERSION>` dizini altında oluşturacaktır. Eğer derlediğimiz kernel imajını kendi sistemimizde kullanmak istemiyorsak (çapraz derleme vb.) modulleri ayrı bir dizine `install` etmemiz gerekir. Bunun için `INSTALL_MOD_PATH` değişkenini aşağıdaki gibi kullanmalısınız:

```
$ make modules_install INSTALL_MOD_PATH=/tmp/modules
```

İşlem sonucunda tüm modüller `/tmp/modules` dizini altındaki dizin yapıları içerisinde oluşturulacaktır.

Paralel Derleme

Derleme süreci oldukça cpu yoğun bir işlemdir. Ek bir parametre verilmemesi halinde sadece tek bir cpu core kullanılır. Oysa kernel içerisindeki Makefile dosyaları paralel derlemeye imkan verecek şekilde düzenlenmiştir. Birden fazla cpu core var ise, bunları maksimum düzeyde kullanmak ve derleme sürecini çok daha çabuk sonuçlandırmak için, paralel derleme yöntemini uygulayabilirsiniz. Bu yöntemde kaç adet paralel derleme süreci olacağını `-j N` parametresi ile verebilirsiniz.

Genel bir yaklaşım, sistemdeki tüm kapasiteyi kullanmak istiyorsanız cpu core sayısının 2 katı kadar bir değer kullanmak yönündedir. Bunun sebebi derleme sürecinde arada `gcc` dışında da betik uygulamaları vb. çalıştığından tam olarak cpu core sayısı verildiğinde atıl kalan bir kapasite olmasıdır. 2 katı gibi bir değer verildiğinde sistemin tüm performansından faydalanabilirsiniz:

```
$ make bzImage -j 8
```

Çapraz Derleme

Initramfs İmajının Eklenmesi

Hazırlanıyor...

U-boot İmajı Haline Getirilmesi

Hazırlanıyor...

Gömülü Sistemlerde Boot Yükleyiciler

U-Boot

U-Boot (Universal Bootloader), ARM, MIPS ve x86 olmak üzere birçok mimariyi destekleyen, açık kaynak kodlu bir önyükleyici uygulamasıdır. Genel olarak işletim sistemi çekirdeğini birincil belleğe yüklemekten ve uygun parametrelerle çalıştırmaktan sorumludur.

U-Boot genellikle, açılış sürecine kullanıcının dahil olmasına imkan vermek için, işletim sistemini başlatmadan önce, belli bir süre kullanıcının bir tuşa basmasını beklemektedir. Bu bekleme süresi, u-boot kaynak kodundan belirlenebildiği gibi sonrasında u-boot üzerinden de değiştirilebilmektedir. Bekleme süresi içinde bir tuşa basılarak u-boot komut satırına düşülebilir.

u-boot bir takım çevre değişkenleri kullanmaktadır. Bu çevre değişkenleri kullanılarak u-boot'un davranışı değiştirilebilmektedir. Önemli gördüğümüz bazı çevre değişkenleri ve görevleri aşağıdaki gibidir.

Çevre Değişkeni	İçeriği
bootcmd	Otomatik olarak çalıştırılacak komut kümesi
bootargs	Çekirdeğe geçirilecek komut satırı parametreleri
bootdelay	bootcmd içeriği işletilmeden önce bekleme süresi
ipaddr	Cihaz IP değeri
serverip	Sunucu IP değeri
baudrate	Baudrate değeri

u-boot ayrıca çok sayıda komut barındırmaktadır. Yine önemli gördüğümüz bazı komutlar ve görevleri aşağıdaki gibidir.

Komut	Görevi
setenv	Çevre değişkeni tanımlar veya değerini günceller
saveenv	Çevre değişkenlerinin değerlerini kalıcı hafızaya yazar
bootm	Bellekte verilen adresteki kodu işletir
fatload	DOS bölümünden belleğe dosya yükler
tftp	TFTP üzerinden dosya yükler
printenv	Çevre değişkenlerini listeler

Komut satırında *help* yazarak tüm komut listesini görebildiğiniz gibi *help [komut adı]* şeklinde bir komut ile ilgili daha detaylı bir bilgi de alabilirsiniz. *mmc* desteği olan bir u-boot üzerinden aldığımız yardım aşağıdaki gibidir.

```
# help mmc
mmc - MMC sub system

Usage:
mmc read addr blk# cnt
mmc write addr blk# cnt
mmc erase blk# cnt
mmc rescan
mmc part - lists available partition on current mmc device
mmc dev [dev] [part] - show or set current mmc device [partition]
mmc list - lists available devices
mmc setdsr - set DSR register value
```

> **Not:** u-boot'un barındırdığı komut kümesi, hedef karta göre değişmektedir. Örneğin ağ ve SD kart desteği olan bir kart için *tftpboot* ve *fatload* komutlarını barındırmasına karşın, bu özelliklerin olmaması durumunda bu komutlar da bulunmayabilir.
>
> **Not:** u-boot çevre değişkenleri, u-boot komut satırından değiştirilebildiği sonrasında Linux üzerinden de değiştirilebilmektedir.

Şimdi bir örnek üzerinden u-boot kaynak kodunun nasıl derlendiğine ve kullanıldığına bakalım.

Örnek platform olarak, *Board Spesifik Kılavuzlar* kısmında incelediğimiz, Olimex A20 kartını kullanacağız. İlk olarak bu kart ve ailesi için özelleştirilmiş u-boot kaynak kodunu indirelim.

```
git clone https://github.com/linux-sunxi/u-boot-sunxi.git
cd u-boot-sunxi
```

arch dizininde birçok mimari için yazılmış C ve sembolik makina kodları bulunmaktadır.

```
# ls arch/
arc  arm  avr32  blackfin  m68k  microblaze  mips  nds32  nios2  openrisc  powerpc  sa
ndbox  sh  sparc  x86
```

Projeyi derlemek için ana dizinde bir *Makefile* dosyası bulunmaktadır. Bu makefile dosyası ile önce projeyi doğru şekilde konfigüre etmeli ve ardından u-boot kodunu derlemeliyiz.

Ana dizinde desteklenen kartları gösteren *boards.cfg* adlı bir dosya bulunmaktadır. Bu dosya kartımıza ilişkin bir giriş bulundurmaktadır.

```
# grep sunxi boards.cfg | awk '{print $7}' | grep A20-OLinuXino-Micro
A20-OLinuXino-Micro
A20-OLinuXino-Micro_FEL
```

u-boot kodunu konfigür etmek için, makefile içeriğindeki *CROSS_COMPILE* değişkenine uygun değer vermeli ve hedef platformumuzu belirtemeliyiz. Genel formu ve örneğimiz için durum aşağıdaki gibidir.

```
make CROSS_COMPILE=<Çapraz Derleyici Önceki> <Öngörülen Ayar dosyası veya İfadesi>
make CROSS_COMPILE=arm-linux-gnueabihf- A20-OLinuXino-Micro_config
```

Çoğunlukla u-boot kodlarında *configs* dizini altında hedef platforma ilişkin öngörülen ayar dosyası bulunmasına karşın, burada incelediğimiz örnek için böyle bir kullanım söz konusu değildir.

Son aşamada ise u-boot kodu aşağıdaki gibi derlenebilir.

```
make CROSS_COMPILE=arm-linux-gnueabihf-
```

u-boot kaynak kodunda ayrıca, bazen MLO olarak da isimlendirilen, daha küçük bir önyükleyici olan SPL (Secondary Program Loader)'de bulunmaktadır. SPL, görev olarak, üretim aşamasında kodlanan ROM yükleyici ile u-boot arasında bulunmaktadır.

Yukarıda derlediğimiz u-boot için seri terminal üzerinden aldığımız açılış mesajları aşağıdaki gibidir.

```
U-Boot 2014.04-10733-gea1ac32 (Feb 02 2015 - 12:05:05) Allwinner Technology

CPU:   Allwinner A20 (SUN7I)
Board: A20-OLinuXino-Micro
I2C:   ready
DRAM:  1 GiB
MMC:   SUNXI SD/MMC: 0
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   emac
Hit any key to stop autoboot:  2
```

Şimdi kısaca bir u-boot kullanım senaryosuna bakalım. Bu aşamada, çekirdeğe geçirilecek komut satırı parametrelerini belirleyebilir, çekirdeği tftp veya SD kart üzerinden çekip cihazı açabiliriz. tftp üzerinden örnek bir kullanım aşağıdaki gibidir.

```
setenv bootargs console=ttyS0,115200 init=/bin/sh root=/dev/mmcblk0p2 rootwait panic=1
0
setenv autoload no
dhcp
setenv serverip 172.16.2.136
tftp 0x48000000 uImage
bootm 0x48000000
```

bootm komutundan sonra, u-boot mesajları aşağıdaki gibidir.

```
bootm 0x48000000
## Booting kernel from Legacy Image at 48000000 ...
   Image Name:   Linux-3.4.103-00033-g9a1cd03
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4555464 Bytes = 4.3 MiB
   Load Address: 40008000
   Entry Point:  40008000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK

Starting kernel ...
```

RedBoot

RedBoot (Red Hat Embedded Debug and Bootstrap firmware) **eCos** realtime işletim sisteminde ve **Intel XScale** gibi bazı **ARM** tabanlı platformlarda yoğun olarak kullanılan bir boot yükleyici uygulamasıdır.

RedBoot kendine özgü bir **partition table** yapısı kullanır. Tablo bilgisini flash üzerinde saklar. Linux çekirdeği içerisinde *RedBoot Partition Table Parsing* aktifleştirildiğinde RedBoot ile aynı şekilde mantıksal bölümlendirme yapılmış olur.

Son zamanlarda kullanım oranının azaldığı gözlemlenmektedir.

ARM Açılış Süreci

ARM mimarisinde açılış süreci aşağıdaki sıra ile gerçekleşir:

- Boot ROM üzerinden dahili kodun çalışması
- Birinci harici boot yükleyicinin çalışması (`X-loader`)
- Daha gelişmiş ikinci boot yükleyicinin çalışması (`U-boot`)
- Linux kernel'ın çalışmaya başlaması

Dahili Boot ROM adresi ilgili donanım için sabit olacaktır. Buradaki kodun temel amacı, birinci harici boot yükleyici kodunu yüklemektir

Bu noktada çoğu zaman donanım üzerinde, birinci harici boot yükleyicinin hangi ortamdan yükleneceğine dair (seri port, NAND flash, SD kart vb.) boot konfigürasyon pin'leri sağlanır.

Boot Yükleyicinin Görevleri: Bellek

Sistemdeki bellek alanlarının tespit edilmesi ve ilklendirilmesi Linux çekirdeğinden önce boot yükleyici uygulaması tarafından yapılmalıdır.

Bu işlem için bir takım otomatik algılama rutinleri, özel algoritmalar veya ilgili donanım için sabit değerler bulunabilir, tamamen boot yükleyici uygulamanın geliştirimine göre değişir.

Boot Yükleyicinin Görevleri: Kernel

Linux kernel imajı, boot yükleyici tarafından belleğe yüklenmelidir.

Boot yükleyici uygulama ne kadar yetenekli ise, o kadar farklı şekilde Linux kernel imajı yüklenebilir. Örneğin, boot yükleyici uygulamada **TCP** ve **HTTP** protokolü destekleniyorsa, herhangi bir web sunucuda barındırılan kernel imajı da download edilip belleğe aktarılabilir.

Öte yandan boot yükleyici uygulamanın olabildiğince küçük ve basit olması beklenir. TCP gibi kompleks bir protokolün dahil edilmesi iyi bir fikir değildir.

Pratik kullanımda boot yükleyiciler genellikle kernel imajını NAND flash bellekten, SD Kart üzerinden, seri porttan, network üzerindeki bir TFTP sunucudan veya USB disk üzerinden okurlar. Bu desteklerden bir veya birden fazlası kullandığınız boot yükleyici içerisinde aynı anda bulunabilir.

Boot Yükleyicinin Görevleri: Initial Ramdisk

Linux tabanlı sistemde, iki aşamalı `initrd` açılış yöntemi kullanılıyor ise, initrd imajını barındıran dosyanın da bulunduğu ortamdan belleğe yüklenmesi boot yükleyici uygulamanın görevleri arasındadır.

Tıpkı Linux imajını yüklerken olduğu gibi, boot yükleyici uygulama içerisindeki destekler, initrd imajının yüklenme opsiyonlarını da belirleyecektir

Oldukça benzer konseptler olmasına rağmen bu durum `initramfs` imajıyla karıştırılmamalıdır. Initramfs imajı doğası itibariyle zaten kernel imajının sonuna eklenmiş olduğundan, ayrıca yüklenmesine ihtiyaç bulunmamaktadır. `initrd` imajının ise boot yükleyici tarafından yüklenip, yüklendiği adresin kernel tarafına bildirimi şarttır.

Boot Yükleyicinin Görevleri: Kernel Parametreleri

Boot yükleyicinin bir diğer temel görevi, Linux çekirdeğine parametrelerin geçirilmesini sağlamaktır. Bu işlem için `ATAGS` sistematiği kullanılır

Fiziksel belleğin `0x100` offsetinden başlayarak **ATAGS** veriyapıları yerleştirilir. Başlangıç adresi aynı zamanda Linux imajını çalıştırmaya başlamadan önce `R2` yazmacına kaydedilecektir.

`ATAG_CORE` tipi ile başlayıp `ATAG_NONE` tipine kadar birbirini takip eden değerler kullanılır.

`ATAGS` Veri Tipleri

Tag	Değer	Açıklama
ATAG_NONE	0x00000000	Tag listesinin sonunu gösterir
ATAG_CORE	0x54410001	Liste başlangıcını gösterir
ATAG_MEM	0x54410002	Bellek üzerinde kullanılabilecek alanları gösterir
ATAG_VIDEOTEXT	0x54410003	VGA metin konsoluyla ilgili parametreleri gösterir
ATAG_RAMDISK	0x54410004	Ramdisk kullanımına ilişkin parametreleri gösterir
ATAG_INITRD2	0x54420005	Initrd imajının nerede bulunacağını gösterir
ATAG_SERIAL	0x54410006	64 bit board seri numarasını tutar
ATAG_REVISION	0x54410007	32 bit board versiyon numarasını tutar
ATAG_VIDEOLFB	0x54410008	Vesa frame buffer için değerleri tutar
ATAG_CMDLINE	0x54410009	Kernel tarafına aktarılacak komut satırı argümanlarını tutar

`ATAGS` Veri Tipleri

ATAGS veri tipleri aşağıdaki gibi başlık ve veri alanlarından oluşmaktadır

```
struct atag_header {
    u32 size; /* legth of tag in words including this header */
    u32 tag;  /* tag value */
};

struct atag {
    struct atag_header hdr;
    union {
        struct atag_core         core;
        struct atag_mem          mem;
        struct atag_videotext    videotext;
        struct atag_ramdisk      ramdisk;
        struct atag_initrd2      initrd2;
        struct atag_serialnr     serialnr;
        struct atag_revision     revision;
        struct atag_videolfb     videolfb;
        struct atag_cmdline      cmdline;
    } u;
};
```

Geleneksel Boot Yöntemi

Kernel **3.8** ve sonrasında kullanılmaya başlanan *Device Tree* modeliyle çalışanların dışında kalan tüm sistemler, geleneksel yöntemle boot sürecini gerçekleştirirler.

Bu yöntemde, her *board ailesi* için değil, her bir *board* için özgün bir `ID` alınması gereklidir. Bu **ID** değeri ARM mimarisi için Linux kernel tarafındaki ana sorumlu olan **Russell King** üzerinden kayıt ettirilmelidir. Güncel listeye http://www.arm.linux.org.uk/developer/machines adresinden ulaşabilirsiniz. 2014 sonu itibariyle **4981** adet board kaydı bulunmaktadır

Board Numarasının Çekirdeğe Aktarımı

Bootloader uygulamasının, özgün board numarasını `R1` yazmacına geçmesi zorunludur.

Linux çekirdeği, boot sürecinde bu yazmaca özgün board numarasının yazıldığını varsayar ve tüm akış bu şekilde ilerler.

Linux çekirdeğinin derlenmesi aşamasında da `R1` yazmacındaki özgün makine numarasıyla ilgili donanım spesifik kısımların seçilmiş ve çekirdek içerisine dahil edilmiş olması zorunludur. Aksi takdirde aşağıdaki gibi bir hata mesajı alınacaktır:

```
Uncompressing Linux... done, booting the kernel.
Error: unrecognized/unsupported machine ID (r1 = 0x000007d9).

Available machine support:

ID (hex)    NAME
00000af0    ti8168evm

Please check your kernel config and/or bootloader.
```

Özet: Boot Süreci

1. Boot ROM üzerinden dahili kod çalışır ve birinci boot yükleyici yüklenir (`X-loader`)
2. X-loader tarafından ek ilklendirmeler yapılır ve daha gelişkin ikincil boot yükleyici yüklenir (`U-boot`)
3. Boot yükleyici tarafından Linux çekirdeği bulunduğu ortamdan belleğe aktarılır
4. Eğer initrd imajı kullanılıyorsa, aynı şekilde initrd imajı da belleğe aktarılır
5. Sisteme ait özgün numara, `R1` yazmacına yazılır
6. Initrd imajının bellekteki yeri ve uzunluğu, kernel açılış parametreleri vb. parametreler için fiziksel belleğin başlangıcına yakın yerlerde `ATAGS` veri yapıları oluşturulur

7. `ATAGS` veri yapısının başlangıç adresi `R2` yazmacına yazılır (Genellikle 0x100)
8. Bellekteki kernel imajındaki ilk makine kodu çalıştırılır

`Device Tree` Yöntemi

Device Tree yönteminde (detaylar için ilgili konu başlığını inceleyiniz) açılış sürecinde bir miktar farklılık bulunmaktadır.

Bu modelde her bir board versionu için özgün bir numara belirtmek yerine, her bir board ailesi için özgün bir *Device Tree* numarası bulunur.

Geleneksel modele oranla çok daha az numara kullanılır zira bir **DT** modeli ile bir çok farklı board versiyonu adreslenebilir.

İlgili **DT** veri yapısı boot yükleyici tarafından belleğe aktarılır ve adresi `R2` yazmacına yazılır

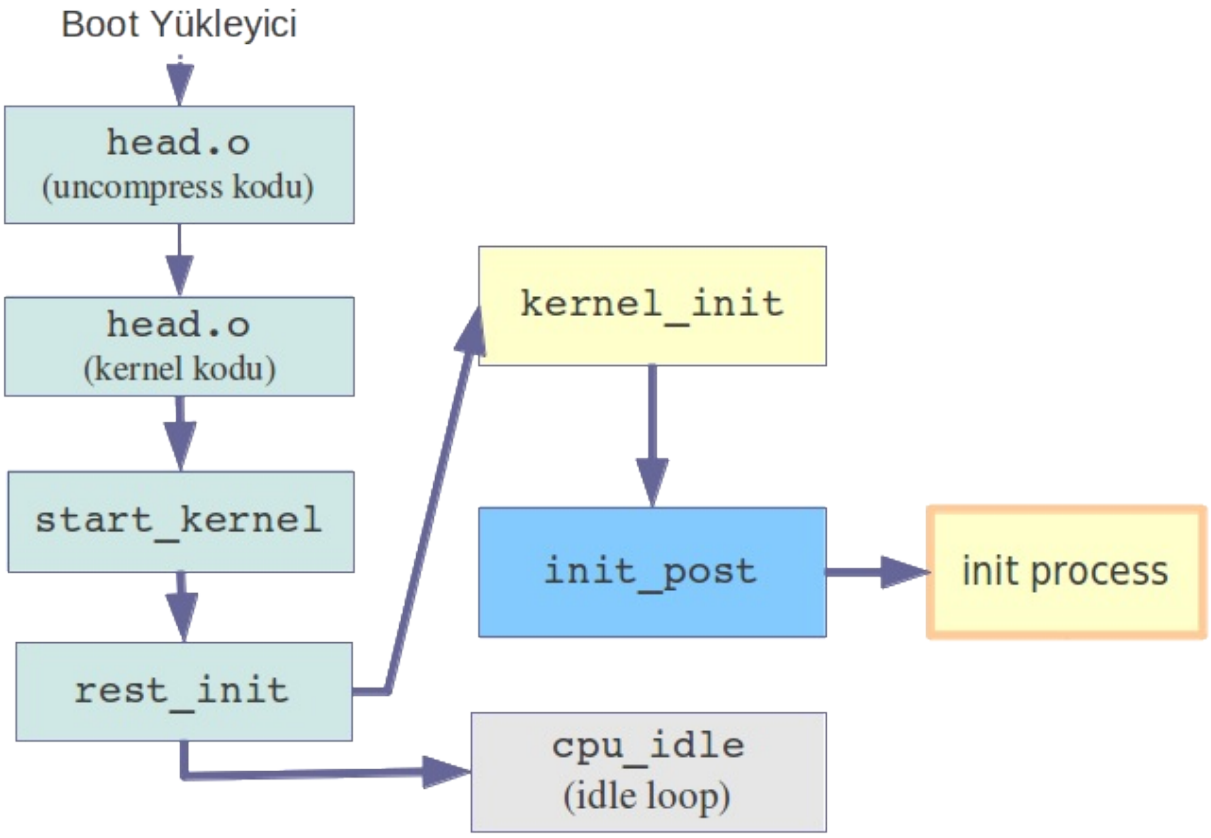
Bu yöntemi desteklemeyen eski boot yükleyicileri için Linux kernel içerisinde `CONFIG_ARM_APPENDED_DTB` özelliği kullanılarak **DT** veri yapısı kernel imajının sonuna eklenebilir

Linux Açılış Süreci

Kernel Açılış Süreci

Sistem Açılışı

Kernel Bootstrap Süreci



Kernel Derleme İşleminde Son Adımlar

```
...
LD      vmlinux
SYSMAP  System.map
SYSMAP  .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
AS      arch/arm/boot/compressed/head.o
GZIP    arch/arm/boot/compressed/piggy.gzip
AS      arch/arm/boot/compressed/piggy.gzip.o
CC      arch/arm/boot/compressed/misc.o
CC      arch/arm/boot/compressed/decompress.o
AS      arch/arm/boot/compressed/head_cpu.o
SHIPPED arch/arm/boot/compressed/lib1funcs.S
AS      arch/arm/boot/compressed/lib1funcs.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
..
```

Kernel Bootstrap Kodu

- head.o:

Bu bölümde mimariye özgün obje kodları bulunur. Boot yükleyici uygulama tarafından çalıştırılmasına yöneliktir.

- head-cpu.o:

İşlemciye özgü ilklendirme işlemlerine ait kodları barındırır.

- decompress.o:

Sıkıştırılmış formda bulunan kernel'i açma işlemlerini gerçekleştirir.

- lib1funcs.o:

ARM mimarisi için optimize edilmiş bölme işlemlerine dair kodları içerir. *VFP NEON*

`head.o` Tarafından Yapılan İşlemler

- Mimari, işlemci ve makine/sistem tipinin belirlenmesi
- **Memory Management Unit** konfigürasyonu ve **virtual memory** desteğinin etkinleştirilmesi
- *init/main.c* içerisindeki **start_kernel** fonksiyounun çağrılması

`start_kernel`

```
asmlinkage void __init start_kernel(void)
{
    char * command_line;
    extern const struct kernel_param __start___param[], __stop___param[];

    smp_setup_processor_id();

    /*
     * Need to run as early as possible, to initialize the
     * lockdep hash:
     */
    lockdep_init();
    debug_objects_early_init();

    /*
     * Set up the the initial canary ASAP:
     */
    boot_init_stack_canary();

    cgroup_init_early();
```

```
    local_irq_disable();
    early_boot_irqs_disabled = true;

/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    tick_init();
    boot_cpu_init();
    page_address_init();
    printk(KERN_NOTICE "%s", linux_banner);
    setup_arch(&command_line);
    mm_init_owner(&init_mm, &init_task);
    mm_init_cpumask(&init_mm);
    setup_command_line(command_line);
    setup_nr_cpu_ids();
    setup_per_cpu_areas();
    smp_prepare_boot_cpu();    /* arch-specific boot-cpu hooks */

    build_all_zonelists(NULL);
    page_alloc_init();

    printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
    parse_early_param();
    parse_args("Booting kernel", static_command_line, __start___param,
           __stop___param - __start___param,
           &unknown_bootoption);

    jump_label_init();

    /*
     * These use large bootmem allocations and must precede
     * kmem_cache_init()
     */
    setup_log_buf(0);
    pidhash_init();
    vfs_caches_init_early();
    sort_main_extable();
    trap_init();
    mm_init();

    /*
     * Set up the scheduler prior starting any interrupts (such as the
     * timer interrupt). Full topology setup happens at smp_init()
     * time - but meanwhile we still have a functioning scheduler.
     */
    sched_init();
    /*
     * Disable preemption - early bootup scheduling is extremely
     * fragile until we cpu_idle() for the first time.
     */
    preempt_disable();
```

```
    if (!irqs_disabled()) {
        printk(KERN_WARNING "start_kernel(): bug: interrupts were "
                "enabled *very* early, fixing it\n");
        local_irq_disable();
    }
    idr_init_cache();
    perf_event_init();
    rcu_init();
    radix_tree_init();
    /* init some links before init_ISA_irqs() */
    early_irq_init();
    init_IRQ();
    prio_tree_init();
    init_timers();
    hrtimers_init();
    softirq_init();
    timekeeping_init();
    time_init();
    profile_init();
    call_function_init();
    if (!irqs_disabled())
        printk(KERN_CRIT "start_kernel(): bug: interrupts were "
                 "enabled early\n");
    early_boot_irqs_disabled = false;
    local_irq_enable();

    /* Interrupts are enabled now so all GFP allocations are safe. */
    gfp_allowed_mask = __GFP_BITS_MASK;

    kmem_cache_init_late();

    /*
     * HACK ALERT! This is early. We're enabling the console before
     * we've done PCI setups etc, and console_init() must be aware of
     * this. But we do want output early, in case something goes wrong.
     */
    console_init();
    if (panic_later)
        panic(panic_later, panic_param);

    lockdep_info();

    /*
     * Need to run this when irqs are enabled, because it wants
     * to self-test [hard/soft]-irqs on/off lock inversion bugs
     * too:
     */
    locking_selftest();

#ifdef CONFIG_BLK_DEV_INITRD
    if (initrd_start && !initrd_below_start_ok &&
        page_to_pfn(virt_to_page((void *)initrd_start)) < min_low_pfn) {
        printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
```

```
            "disabling it.\n",
            page_to_pfn(virt_to_page((void *)initrd_start)),
            min_low_pfn);
        initrd_start = 0;
    }
#endif
    page_cgroup_init();
    enable_debug_pagealloc();
    debug_objects_mem_init();
    kmemleak_init();
    setup_per_cpu_pageset();
    numa_policy_init();
    if (late_time_init)
        late_time_init();
    sched_clock_init();
    calibrate_delay();
    pidmap_init();
    anon_vma_init();
#ifdef CONFIG_X86
    if (efi_enabled)
        efi_enter_virtual_mode();
#endif
    thread_info_cache_init();
    cred_init();
    fork_init(totalram_pages);
    proc_caches_init();
    buffer_init();
    key_init();
    security_init();
    dbg_late_init();
    vfs_caches_init(totalram_pages);
    signals_init();
    /* rootfs populating might need page-writeback */
    page_writeback_init();
#ifdef CONFIG_PROC_FS
    proc_root_init();
#endif
    cgroup_init();
    cpuset_init();
    taskstats_init_early();
    delayacct_init();

    check_bugs();

    acpi_early_init(); /* before LAPIC and SMP init */
    sfi_init_late();

    ftrace_init();

    /* Do the rest non-__init'ed, we're now alive */
    rest_init();
}
```

- `setup_arch(&command_line)` ile bootloader tarafından spesifik bir adrese konulmuş olan kernel boot parametrelerini işler
- Mesajları olabildiğince erken gösterebilmek için console aygıtının ilklendirilmesi
- security, buffers, high resolution timers gibi bir çok altsistemin ilklendirilmesi
- Son olarak `rest_init` 'in çağrılması

`rest_init`

`init` sürecinin her zaman **PID** değeri olarak **1** alması için erkenden bir thread oluşturuluyor ve `idle_loop` 'a geri dönüyor:

```
static noinline void __init_refok rest_init(void)
{
        int pid;

        rcu_scheduler_starting();
        /*
         * We need to spawn init first so that it obtains pid 1, however
         * the init task will end up wanting to create kthreads, which, if
         * we schedule it before we create kthreadd, will OOPS.
         */
        kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
        numa_default_policy();
        pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
        rcu_read_lock();
        kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
        rcu_read_unlock();
        complete(&kthreadd_done);

        /*
         * The boot idle thread must execute schedule()
         * at least once to get things moving:
         */
        init_idle_bootup_task(current);
        preempt_enable_no_resched();
        schedule();

        /* Call into cpu_idle with preempt disabled */
        preempt_disable();
        cpu_idle();
}
```

`kernel_init`

kernel_init temel olarak iki çağrıda bulunur:

- Bu aşamada temel kernel servisleri hazır olduğundan, *device init* işlemlerini başlatmak amacıyla `do_basic_setup` çağrılır

```
!c
static void __init do_basic_setup(void)
{
    cpuset_init_smp();
    usermodehelper_init();
    shmem_init();
    driver_init();
    init_irq_proc();
    do_ctors();
    usermodehelper_enable();
    do_initcalls();
}
```

- Sonrasında `init_post` çağrılır

`init_post`

- Boot işleminin son adımlarını gerçekleştirmekten sorumludur.
- Bir *console* açmayı dener (Initial Console)
- Başarısız olduğu takdirde: `Unable to open initial console` uyarı mesajı görüntülenir.
- Ardından `init` process'ini çalıştırmayı dener.
- Başarılı olması halinde *rest_init* sürecinde oluşturulan kernel thread'ini bir userspace process'e dönüştürür.

`init_post`

```
static void run_init_process(const char *init_filename)
{
        argv_init[0] = init_filename;
        kernel_execve(init_filename, argv_init, envp_init);
}

static noinline int init_post(void)
{
        /* need to finish all async __init code before freeing the memory */
        async_synchronize_full();
        free_initmem();
        mark_rodata_ro();
        system_state = SYSTEM_RUNNING;
        numa_default_policy();


        current->signal->flags |= SIGNAL_UNKILLABLE;

        if (ramdisk_execute_command) {
                run_init_process(ramdisk_execute_command);
                printk(KERN_WARNING "Failed to execute %s\n",
                                ramdisk_execute_command);
        }

        /*
         * We try each of these until one succeeds.
         *
         * The Bourne shell can be used instead of init if we are
         * trying to recover a really broken machine.
         */
        if (execute_command) {
                run_init_process(execute_command);
                printk(KERN_WARNING "Failed to execute %s.  Attempting "
                                        "defaults...\n", execute_command);
        }
        run_init_process("/sbin/init");
        run_init_process("/etc/init");
        run_init_process("/bin/init");
        run_init_process("/bin/sh");

        panic("No init found.  Try passing init= option to kernel. "
              "See Linux Documentation/init.txt for guidance.");
}
```

Özet

- Boot yükleyici uygulaması tarafından bootstrap kodu çalıştırılıyor
- Bootstrap kodu işlemci ve kartı ilklendiriyor

- Ardından Linux kernel'i RAM bellek üzerinde uncompress ediyor.
- *start_kernel* fonksiyonu çalıştırılıyor
- Boot yükleyici tarafından sağlanan command line bölümü kopyalanıyor
- Sistem ve konsol ilklendirmeleri yapılıyor
- Temel kernel servisleri ilklendiriliyor
- İleride `init` process'ine dönüşecek kernel thread'i oluşturuluyor
- Aygıtlar ilklendiriliyor ve `init` process'i userspace'ten çalıştırılıyor.

Kullanıcı Kipine Geçiş - Init Süreci

Linux çekirdeği açılış sürecinin kendisiyle ilgili son 2 adımında önce kök dosya sistemini bağlar, ardından kök dosya sisteminde yer alan bir adet uygulamayı başlatır.

Eğer çekirdek kendisiyle ilgili tüm açılış işlemlerini yapmasına rağmen, çalıştıracağı uygulamayı sistemde bulamaz veya uygulamayı çalıştıramaz ise, aşağıdaki gibi bir hata mesajı ile **panic** durumuna geçer:

```
Kernel panic - not syncing: No init found
```

Bu senaryo oluştuğunda henüz kullanıcı kipinde hiç bir uygulama çalıştırılamamış olduğundan, sistemi kullanmanız mümkün olmaz. Yapılacak yegane işlem, `panic=10` gibi bir değeri çekirdek açılış parametresi olarak kullanmaktır. Bu değer çekirdeğe, herhangi bir sebeple **panic** durumuna düşülürse, sistemin girilen saniye değeri sonrasında yeniden başlatılmasının istendiğini belirtir. Özellikle gömülü sistemler için **panic=xxx** parametresinin kullanılması önerilir. Herhangi bir beklenmedik sebeple çekirdek bu duruma düşerse, özellikle uzak lokasyonlarda yer alan cihazlarınız olduğunu varsayarsak, panic durumda kalmaktansa bir süre sonra otomatik yeniden başlama sürecinin denenmesini talep etmek ve sisteme tekrar erişim sağlamayı ummak daha iyi bir alternatiftir.

Init Uygulamasının Çalıştırılması

Çekirdek tarafından kullanıcı kipinde çalıştırılan bu uygulamanın Process ID (**PID**) değeri her zaman **1** olur. Eğer çalıştırılan bu process herhangi bir şekilde sonlanırsa, çekirdek tarafından yeniden bir process çalıştırmayı denemek şeklinde bir aksiyon alınmaz. Dolayısıyla kullanıcı kipinden baktığımızda sistem kilitlenmiş olur.

> **SORU:** Sistem açıldıktan sonra **root** kullanıcısı ile 1 nolu process'e `kill 1` veya `kill -9 1` komutu ile `SIGTERM` ve `SIGKILL` sinyali gönderirsek sistem kilitlenir mi?

1 nolu process'in çekirdek seviyesinde özel bir durumu vardır. Sistemdeki 1 nolu process'i **root** kullanıcısı ile dahi sinyal göndererek sonlandıramazsınız. 1 nolu process başlatılırken çekirdek tarafındaki process tablosunda **SIGNAL_UNKILLABLE** bayrağı ile işaretlenir. Bu sayede sonraki aşamalarda process'in sonlanmasına yol açabilecek türden hiç bir sinyal çekirdek tarafından bu process'e gönderilmez. Bu süreci detaylı incelemek isteyenler `kernel/fork.c` içerisindeki `is_child_reaper()` ve `kernel/signal.c` içerisindeki `SIGNAL_UNKILLABLE` kullanımlarına göz atabilir.

İlk çalıştırılan uygulamaya tehlikeli sinyallerin gönderilmesi çekirdek tarafından engelleniyor fakat uygulama içerisinde hata olması durumunda veya kontrollü bir `exit` yapılıyorsa, ilk çalışan uygulama sonlanmış olacağı için sistem kilitlenmiş olur.

Bu noktadan hareketle, sistemdeki ilk çalışacak uygulamayı kendimiz geliştirmek yerine, belki de toplam milyarlarca farklı sistem ve cihazda çalışan ve hiçbir hata içermediği böylece kanıtlanmış olan bir uygulamayı kullansak daha iyi olmaz mıydı?

Bu sorunun yanıtı **System V** ekolündeki Unix/Linux sistemleri için `/sbin/init` uygulamasıdır.

Eğer açılış sırasında çekirdek parametresi olarak `init=xxx` şeklinde bir parametre verilerek ilk çalıştırılacak uygulama özellikle belirtilmemişse, öntanımlı olarak `/sbin/init` çalıştırılır.

Aslında çoğu çekirdek versiyonunda öntanımlı uygulama olarak `/sbin/init` 'in çalıştırılması denenir ancak başarısız olursa sistemi açabilmek adına bir kaç farklı dosya daha sistemde aranır ve çalıştırılmaya çalışılır. Çekirdek içerisindeki `init/main.c` dosyasındaki aşağıdaki bölümü inceleyiniz:

```
...
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");

panic("No init found.  Try passing init= option to kernel. "
"See Linux Documentation/init.txt for guidance.");
```

> **SORU:** 1 nolu uygulamayı sonlandıramazsınız diyorsunuz. Peki 1 nolu **init** uygulaması sistem açıldıktan, tüm servisler (daemon'lar) ayağa kalktıktan sonra kendi isteğiyle `exit` ederse ne olur? Sistem açılmış, servisler çalışmaya başlamış olduğunda göre `init` 'in çalışmasına halen ihtiyaç var mıdır?

Böyle bir durumda sistemimiz intihar teşebbüsünde bulunmuş olacaktır. Bir uygulamanın servis (daemon) olması ne demektir? Uygulamaların onları başlatan uygulamalar (*parent process*) sonlansa dahi arkada çalışmaya devam eder hale getirilmesine şeytanlaştırma :) (***daemonize***) ismini veriyoruz. Aslında bu sürecin arkasında yatan basit mantık da, daemon haline getirmek istediğimiz uygulamanın *parent process id* değerini (**PPID**), **1** nolu process olarak değiştirmekten ibarettir. 1 nolu process hiç ölmeyeceği için daemon yaptığımız uygulamanın parent process'i hiç sonlanmayacak, bu sayede daemon da arka planda çalışmaya devam edecektir. Herhangi bir şekilde sadece 1 nolu uygulamayı sonlandırabiliyor olsaydık dahi, tüm daemon'ların anası 1 nolu process olduğundan, çekirdek tarafından tüm *child* process'leri de sonlandırılacaktı.

`/sbin/init` 'le Çalışmak

Buraya kadar okuduklarımızda, açılış sürecini yönetecek uygulamayı sıfırdan yazma riskini üstlenmektense, onun yerine `init` uygulamasını kullanmamız gerektiğini öğrendik. Peki, açılış sürecinin tümünü yönetmek istediğimize göre, bu görevi `init` 'e devredersek sonra nasıl tekrar süreci kontrol edeceğiz?

Init uygulama kodu oldukça sadedir. Temel bazı işlevler haricinde fazla bir iş yapmaz. Normal Linux dağıtımlarımızda kullandığımız `init` uygulaması ile busybox içerisinden çıkan kırpılmış `init` uygulaması birebir aynı değildir ancak temelde aynı işlevleri yerine getirirler. Gömülü sistemlerde kullanacağımız busybox init uygulaması orjinalinden farklı olarak çalışm seviyelerini (*runlevel*) desteklemez. Ancak çalışma seviyelerinin gömülü bir sistem için zaten pek işlevsel bir kullanımı bulunmamaktadır. Bu sebeple busybox init uygulamasının konfigürasyon dosyasında da minik farklar bulunur.

Init uygulaması ayağa kalktığında `/etc/inittab` dosyasını okur. Busybox versiyonunda `/etc/inittab` dosyasının okunması derleme sürecinde `CONFIG_FEATURE_USE_INITTAB` opsiyonuyla değiştirilebilmektedir. Bu özellik devre dışı bırakıldığında da init uygulaması çalışabilir bir sisteme ulaşmak adına öntanımlı olarak `/etc/init.d/rcS` betik uygulamasını çalıştırır ve sistemin öntanımlı konsolunda kabuk uygulamasını başlatır.

Süreci daha açık hale getirmek için `/etc/inittab` dosyasını kullanma yolunu tercih edeceğiz (önerilen çalışma şekli de budur).

Aşağıda busybox için örnek bir `inittab` dosyası yer almaktadır:

```
# /etc/inittab

::sysinit:/etc/init.d/rcS

# /bin/sh invocations on selected ttys
#
# Note below that we prefix the shell commands with a "-" to indicate to the
# shell that it is supposed to be a login shell.  Normally this is handled by
# login, but since we are bypassing login in this case, BusyBox lets you do
# this yourself...

# Start an "askfirst" shell on the console (whatever that may be)
::askfirst:-/bin/sh

# Start an "askfirst" shell on /dev/tty2-4
tty2::askfirst:-/bin/sh
tty3::askfirst:-/bin/sh
tty4::askfirst:-/bin/sh

# /sbin/getty invocations for selected ttys
tty5::respawn:/sbin/getty 38400 tty5
tty6::respawn:/sbin/getty 38400 tty6

# Example how to put a getty on a modem line.
#::respawn:/sbin/getty 115200 ttyS2

# Stuff to do when restarting the init process
::restart:/sbin/init

# Stuff to do before rebooting
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
::shutdown:/sbin/swapoff -a
```

Şimdi bu dosyadaki önemli satırları detaylandıralım.

`::sysinit:/etc/init.d/rcS`

Dosyada bizi en çok ilgilendiren bölüm, `sysinit` ile belirtilen kısımlardır. Bu anahtar kelime ile, `init` uygulamasına açılış sürecinde bizim adımıza bir başka uygulamayı başlatmasını söylüyoruz. Yani açılış sürecini init uygulamasına devretmekle kaybettiğimiz kontrolü, `sysinit` ile tanımladığımız uygulama üzerinden tekrar geri alıyoruz. Bu şekilde belirttiğimiz uygulamamızda sistemin açılışında yapılması gereken tüm işlemleri gerçekleştirip, gerekli servisleri de ayağa kaldırdığımızda tam anlamıyla çalışan bir sisteme kavuşmuş olacağız.

sysinit deyimi birden fazla satırda yer alabilir. Böyle bir kullanımda `init` uygulaması dosyada yer alan sıralama doğrultusunda her bir `sysinit` ile belirtilen uygulamanın çalışmasının sonlanmasını bekler ve bir sonraki uygulamaya geçer. Bu şekildeki bir kullanım

yerine `inittab` dosyasında tek bir `sysinit` içeren satırın yer alması ve çalıştırılacak diğer mekanizmaların ilgili uygulama içerisinden yönetilmesi tercih edilir.

`::askfirst:-/bin/sh`

Bu özel satır, çekirdeğin açılış sürecinde de kullandığımız ön tanımlı sistem konsolu üzerinde, `/bin/sh` kabuğunu başlatacaktır. `askfirst` anahtar kelimesi sayesinde karşımıza bir bilgi mesajı çıkacak `ENTER` tuşuna basmamız halinde ise kabuk uygulaması çalışacak ve kabuğu kullanmaya başlayabileceğiz. Herhangi bir güvenlik kontrolü içermeyen bu mekanizma, geliştirme sürecinin başlangıcında işimize yarayabilir ancak daha sonra devre dışı bırakılmalıdır. Aksi takdirde sistemimize bir şekilde seri konsol bağlantısı yapmayı başaran birisi, doğrudan sistem üzerinde bir kabuk çalıştırabilir.

`tty2::askfirst:-/bin/sh`

Konsolun haricinde benzer şekilde, **tty2**, **tty3** ve **tty4** sanal konsollarında güvenlik kontrolü olmadan kabuk uygulamasının başlatılması bu şekilde öntanımlı olarak örnek `inittab` dosyasında yer almaktadır. Eğer sisteme bağlı bir monitor var ve bu sanal konsol mekanizmasını kullanmak istiyorsak, aşağıda anlatacağımız şekilde `askfirst` ile değil, parola korumasıyla girişi sağlamalıyız. Ya da pek çok gömülü sistemde olduğu gibi, bu şekilde bir sanal konsol kullanımına ihtiyacımız yok ise, bu satırların `inittab` dosyasından çıkarılması yerinde olacaktır.

`tty5::respawn:/sbin/getty 38400 tty5`

Bu örnekte 5. sanal konsolda `getty` uygulamasının `38400 tty5` parametreleri ile başlatılması sağlanmaktadır. Eğer sanal konsol kullanacaksak bu yöntem izlenmelidir. `getty` uygulaması çalıştığı konsolda öncelikle `login` uygulamasını çalıştıracak ve kullanıcı adı, parola kontrolünü yapacak, başarılı olması durumunda kullanıcı adına kabuk uygulamasını başlayacaktır.

`respawn` özel deyimi ise, `init` uygulaması tarafından sağlanan ek bir özellik olup, ilgili kuralda çalştırılmak üzere belirtilen uygulama herhangi bir şekilde sonlanacak olursa, otomatik olarak aynı parametrelerle yeniden başlatılmasını (*respawning*) sağlar (init uygulaması bu gibi fonksiyonları sağlayabilmek adına da hiç ölmemelidir).

Respawn özelliği gömülü sistemimizde bizim için faydalı bir kullanım alanı bulabilir. Örnek olarak sisteminizde 3 adet önemli daemon yazdığınızı düşünelim. 4. olarak da bu daemon'ların çalışıp çalışmadığını periyodik olarak kontrol eden ve sürekli arka planda çalışan bir uygulamanız olduğunu düşünelim, adı **controller** olsun. Eğer controller uygulamanızda uygulamanın sonlanmasına neden olacak herhangi bir hata olursa, tüm

kontrol mekanizması devre dışı kalacaktır. **controller** uygulamanızı kontrol edecek ayrı bir uygulama yapsanız aynı sorun onun için de geçerli olacağından, hiç bir zaman ölmeyecek bir uygulamadan yardım almamız gerekecektir ki bu uygulama `init` olmaktadır. Teorik olarak bunun için çekirdekten de yardım talep edebilirdik ancak Linux çekirdeği böyle işlere bulaşmaz ve kullanıcı kipinde çalışan uygulamalardan ilk çalıştırdığı `init` haricindekilere özel bir muamele yapmaz. O yüzden biz de `init` uygulamasının yardımına başvurarak, sistemdeki diğer servisleri kontrol eden **controller** uygulamamızı, `respawn` deyimiyle `/etc/inittab` dosyasına yazıp, `init` tarafından sonlansa bile yeniden başlayacak hale getirebiliriz.

> Bu noktada şöyle de düşünebilirsiniz: zaten **controller** uygulama kodumuz çok basit olacağından hiç çökmeyecek bir şey yazabiliriz. O yüzden gene de `init` 'in yardımına ihtiyacımız var mı?

Doğru bir soru, gidiş yolundan puan verilebilir. Temel hedef tüm hataların kontrol edildiği ve çökmeyecek uygulamalar yazabilmek olmalıdır. Aksi takdirde bu iş tüm uygulamaların `init` üzerinden başlatılmasına kadar gider ki bu işlemin burada ayrıntısına girmeyeceğimiz başka zorlukları da vardır.

Fakat özellikle gömülü sistemler için, sistemin sağlıklı çalışması ve örneğin uzaktan bağlantı yapılabilmeye devam etmesi için gereken minimum servislerin her zaman çalışır durumda olduğunun kontrolünü yapan ve çalışmayanları yeniden başlatan, belki bundan biraz daha fazlasını da yapan genel bir **controller** yazılımının kendisini `init` üzerinden `respawn` ile çalıştırılacak şekilde başlatmak daha doğru bir davranış olacaktır.

controller uygulamanızı çok iyi yazmış olabilirsiniz, ancak sistemdeki başka bir uygulamanın belleği tükettiğini ve yeni bellek isteyen bir uygulama için çekirdek içerisindeki **O**ut **O**f **Memory K**iller mekanizmasının bellekte yer açmak için kullandığı algoritma sonucu sizin **controller** process'inizi seçtiğini ve onu sorgusuz sualsiz öldürdüğünü düşünelim (çekirdek gerçekten böyle işler de yapmaktadır). Bu durumda sizin uygulamanız çok iyi yazılmış olsa bile beklentiniz dışında bir sonlanma da yaşayabilirsiniz. Linux **OOM** Killer algoritmasının aday process seçim sürecinde **controller** uygulamanıza daha düşük değer vermesini sağlamak için yapabileceğiniz bazı işlemler bulunmaktadır. Ancak bunun gibi farkında olamayabileceğiniz detaylar nedeniyle, **controller** uygulaması için `init` uygulamasından yardım talep etmemiz yerinde olacaktır, ayıp karşılanmaz.

```
::respawn:/sbin/getty 115200 ttyS2
```

Bu satır `respawn` mekanizmasıyla **ttyS2** seri portunda **115200** baudrate ile `getty` uygulamasının başlatılmasını sağlamaktadır.

Gömülü sistemlerde seri port aygıt dosyası isimlendirmeleri kullanılan teknoloji ve board ailesine göre (*ttyO1* vb.) değişkenlik gösterebilmektedir. Bu bölümde sistemin standart konsolu veya diğer seri portları üzerinde kullanıcı kimlik denetimi yaparak girişe izin vermek istediğimiz aygıtlar için her biri ayrı bir satırda yer alacak şekilde ayarlamalarımızı yapabiliriz.

`::restart:/sbin/init`

Bu ayar bir miktar kafa karıştırıcı olabilir. Bazen `SIGHUP` sinyali göndererek `/etc/inittab` dosyasının yeniden okunmasını sağlamak yeterli olmaz ve `init` sürecini yeniden başlatmak gerekir. Init sürecini yeniden başlatmak istediğimizde (*restart*), hangi uygulamanın çalıştırılacağını söylüyor. Bazı özel senaryolarda, sistemin kök dizinini çalışma sırasında değiştirmeniz gerekebilir. Yeni kök dizinde farklı bir `/sbin/init` versiyonu da yer alabilir. Buradaki öntanımlı davranışla `init` 'e *restart* eyleminde gene `/sbin/init` uygulamasının çalıştırılacağını söylemiş oluyoruz.

> **SORU:** Peki bu şekilde bir *restart* işlemi sonucunda yeni oluşacak `init` process'in **PID** değeri **1**'den büyük olmayacak mıdır?

`::ctrlaltdel:/sbin/reboot`

Bu deyim özellikle **X86** tabanlı sistemlerde klavyeden `CTRL-ALT-DEL` tuşlarına basıldığında tetiklenen sistemi yeniden başlatma işleminde hangi uygulamanın çalışacağını göstermektedir.

`::shutdown:/bin/umount -a -r`

Shutdown anahtar kelimesi ile yer alan satırlar, sistem kapatılırken hangi uygulamaların çalışacağını belirtir. Birden fazla tekrar edildiğinde, kapanış sırasında `inittab` dosyasında yer aldığı sıralama ile ilgili komutlar çalıştırılacaktır.

Shutdown anahtar kelimesinin tekrarından oluşan satırlar ile kapanış sırasında birden fazla işlem yapmak yerine,

```
::shutdown:/etc/kapanis
```

örneğindeki gibi tüm süreci bir betik uygulamasına yönlendirip, betik içerisinde gereken diğer tüm işlemleri yapmanız daha doğru olacaktır.

Busybox ile Kök Dosya Sistemi Oluşturma

Gömülü sistemimizin oluşturulmasında kök dosya sisteminin üretimi önemli adımların başında gelmektedir.

Boot yükleyici ve Linux çekirdeği ile ilgili işlemler tamamlandıktan sonra, çekirdek tarafından **/** kök dizini altına bağlanacak dosya sistemimizi hazırlamalıyız. Dosya sistemimizin temellerini ise **Busybox** ile oluşturacağız.

Busybox projesi **1995** yılında, Özgür Yazılım dünyasının önemli figürlerinden Bruce Perens tarafından geliştirilmeye başlanmıştır. Projenin temel hedefi, 1.44 MB'lık tek bir disket içerisine temel bir Linux kök dosya sistemini ve Linux kurulum uygulamasını sığdırmak idi. O tarihlerde tipik bir Linux kurulum süreci, 2 disket ile başlıyordu. Birinci diskette Linux çekirdeği yer alıyordu. Kök dosya sistemi ise 2. disket üzerinden okunuyordu.

Linux dağıtımının kurulumu süresince bize yardımcı olan kurulum (*installer*) uygulaması da herhangi bir Linux uygulamasından farklı değildir. Dolayısıyla çalışabilmesi için alt tarafta başta standart C kütüphanesi olmak üzere, çalışan bir Linux dosya sistemine de ihtiyaç duymaktadır. Bunun yanı sıra kurulumun çeşitli aşamalarında **fdisk**, **mount**, **cp**, **mkdir**, .. vb. temel Linux araçlarının da sistemde yer alması gereklidir.

Busybox projesi derlendiğinde tek bir uygulama dosyası (**busybox**) oluşmaktadır. Bu tek uygulama içerisinde, pek çok temel Linux uygulamasına ait fonksiyonlar (kabuk, editör, dosya sistemi işlemler, disk bölümleme ve biçimlendirme araçları vb.) yer almaktadır. Busybox uygulaması içerisindeki bu fonksiyonları kullanmak için, ilgili fonksiyon ve fonksiyonun parametreleri busybox uygulamasına verilmelidir. Örnek olarak sistemde `mkdir` komutuyla `yeni` adında bir dizin yaratmak istersek, bunu `busybox` ile şu şekilde yapabiliriz:

```
$ busybox mkdir yeni
```

Eğer busybox uygulamasını derlerken `mkdir` fonksiyonlarını içeren `applet` 'i de dahil etmiş isek, işlem gerçekleşecektir.

Busybox uygulamamız içerisinde hangi fonksiyonlara destek olduğunu görmek için, uygulamayı parametre vermeden çalıştırıp çıktıyı inceleyebiliriz:

```
$ busybox
Usage: busybox [function [arguments]...]
or: busybox --list[-full]
or: busybox --install [-s] [DIR]
or: function [arguments]...

BusyBox is a multi-call binary that combines many common Unix
utilities into a single executable.  Most people will create a
link to busybox for each function they wish to use and BusyBox
will act like whatever it was invoked as.

Currently defined functions:
[, [[, acpid, add-shell, addgroup, adduser, adjtimex, arp, arping, ash, awk, base64, b
asename, beep, blkid, blockdev, bootchartd, brctl, bunzip2,
bzcat, bzip2, cal, cat, catv, chat, chattr, chgrp, chmod, chown, chpasswd, chpst, chro
ot, chrt, chvt, cksum, clear, cmp, comm, conspy, cp, cpio,
crond, crontab, cryptpw, cttyhack, cut, date, dc, dd, deallocvt, delgroup, deluser, de
pmod, devmem, df, dhcprelay, diff, dirname, dmesg, dnsd,
dnsdomainname, dos2unix, du, dumpkmap, dumpleases, echo, ed, egrep, eject, env, envdir
, envuidgid, ether-wake, expand, expr, fakeidentd, false,
fatattr, fbset, fbsplash, fdflush, fdformat, fdisk, fgconsole, fgrep, find, findfs, fl
ock, fold, free, freeramdisk, fsck, fsck.minix, fstrim, fsync,
ftpd, ftpget, ftpput, fuser, getopt, getty, grep, groups, gunzip, gzip, halt, hd, hdpa
rm, head, hexdump, hostid, hostname, httpd, hush, hwclock, id,
ifconfig, ifdown, ifenslave, ifplugd, ifup, inetd, init, insmod, install, ionice, iost
at, ip, ipaddr, ipcalc, ipcrm, ipcs, iplink, iproute, iprule,
iptunnel, kbd_mode, kill, killall, killall5, klogd, last, less, linux32, linux64, linu
xrc, ln, loadfont, loadkmap, logger, login, logname, logread,
losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lsof, lspci, lsusb, lzcat, lzma, lzop, lzop
cat, makedevs, makemime, man, md5sum, mdev, mesg, microcom,
mkdir, mkdosfs, mke2fs, mkfifo, mkfs.ext2, mkfs.minix, mkfs.vfat, mknod, mkpasswd, mks
wap, mktemp, modinfo, modprobe, more, mount, mountpoint,
mpstat, mt, mv, nameif, nanddump, nandwrite, nbd-client, nc, netstat, nice, nmeter, no
hup, nslookup, ntpd, od, openvt, passwd, patch, pgrep, pidof,
ping, ping6, pipe_progress, pivot_root, pkill, pmap, popmaildir, poweroff, powertop, p
rintenv, printf, ps, pscan, pstree, pwd, pwdx, raidautorun,
rdate, rdev, readahead, readlink, readprofile, realpath, reboot, reformime, remove-she
ll, renice, reset, resize, rev, rm, rmdir, rmmod, route, rpm,
rpm2cpio, rtcwake, run-parts, runlevel, runsv, runsvdir, rx, script, scriptreplay, sed
, sendmail, seq, setarch, setconsole, setfont, setkeycodes,
setlogcons, setserial, setsid, setuidgid, sh, sha1sum, sha256sum, sha3sum, sha512sum,
showkey, shuf, slattach, sleep, smemcap, softlimit, sort,
split, start-stop-daemon, stat, strings, stty, su, sulogin, sum, sv, svlogd, swapoff,
swapon, switch_root, sync, sysctl, syslogd, tac, tail, tar,
tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, timeout, top, touch, tr, tracer
oute, traceroute6, true, tty, ttysize, tunctl, ubiattach,
ubidetach, ubimkvol, ubirmvol, ubirsvol, ubiupdatevol, udhcpc, udhcpd, udpsvd, umount,
 uname, unexpand, uniq, unix2dos, unlink, unlzma, unlzop, unxz,
unzip, uptime, users, usleep, uudecode, uuencode, vconfig, vi, vlock, volname, wall, w
atch, watchdog, wc, wget, which, who, whoami, whois, xargs, xz,
xzcat, yes, zcat, zcip
```

Busybox içerisinde yer alan tüm fonksiyonları, yukarıdaki örnekte gösterdiğimiz şekliyle busybox uygulamasına parametre olarak geçirmek suretiyle kullanmaya kalksaydık, özellikle kabuk betiklerimiz gereksiz tekrarlar nedeniyle oldukça çirkin görünecek, betiklerin okunabilirliği de önemli oranda azalacaktı.

Busybox bu problemi çok basit ama zekice bir yöntemle çözmektedir. Tek bir uygulamadan ibaret olmasına rağmen, `make install` komutu verildiğinde, busybox içerisinde desteği eklenmiş tüm fonksiyonlar (uygulamalar) için kendisini gösterecek şekilde sembolik linkler oluşturmaktadır. Örnek olarak, `/bin/mkdir` linki `/bin/busybox`'ı göstermekte, `/bin/cp` linki de aynı şekilde `/bin/busybox`'ı işaret etmektedir. Bu şekilde tüm gerekli linkler üretildiğinde artık uygulamaları `busybox mkdir` şeklinde değil sadece `mkdir` şeklinde çağırmamız mümkün olmaktadır. Sonuçta tüm bu linkler yüzünden aynı uygulama çalışmakta, ancak her defasında farklı isimle çağırmış olduğumuzdan `argv[0]` adresinde çağrıldığı programın ismi (örneğin mkdir) yer almaktadır. Busybox içerisindeki komut satırını parse eden bölüm, çağrılma biçimi `busybox` dışındaki bir değer ise ve bu değere ilişkin destekler uygulama içerisinde derlenmişse, kendi içinde ilgili fonksiyonu çağırmak suretiyle istenen amacı gerçekleştirmektedir.

Konfigürasyon ve Derleme

Busybox içerisinde pek çok bileşenle geldiğinden, derleme süreci öncesinde hangi bileşenlerden oluşan bir busybox uygulaması üretmek istediğimizi belirlememiz, bunun için bir konfigürasyon dosyası üretmemiz gerekiyor.

Seçenek / kombinasyon sayısı bir kernel derleme süreci kadar devasa olmasa da, hatırı sayılır miktarda seçenek kümesi olduğunu söyleyebiliriz. Tıpkı kernel derleme konfigürasyonu sürecinde olduğu gibi, busybox konfigürasyon ve derleme işlemleri için de **Kbuild** sistemi kullanılmaktadır.

Menuconfig arayüzü üzerinden konfigürasyon sürecimizin hedefi olan `.config` dosyasını üretebiliriz. Bunun için `make menuconfig` veya aşağıdaki **make** hedeflerinden biriyle `.config` dosyamızı oluşturabiliriz:

Make Hedefi	Açıklama
menuconfig	İnteraktif menü tabanlı seçi arabirimini çalıştırır
defconfig	Hemen her seçeneğin aktifleştirildiği genel bir konfigürasyon sağlar
allyesconfig	Tüm olası seçeneklerin aktifleştirildiği bir konfigürasyon sağlar
allnoconfig	Tüm seçeneklerin inaktif edildiği, minimum bir konfigürasyon sağlar
android_defconfig	Android platformu için derleme yapılacaksa genel bir seçenek kümesi sağlar

Seçim işlemi tamamlandıktan sonra

```
$ make
```

komutuyla derleme işlemi yapılabilir. Paralel derleme özelliğini `-j` parametresi ile devreye alıp (genelde işlemci çekirdek sayısının 2 katına kadar değer girebilirsiniz) derleme sürecini hızlandırabilirsiniz:

```
$ make -j 8
```

> **NOT:** Bu noktadan sonraki çapraz derleme örneklerinde, `/home/training/toolchains` dizini altına `gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux` toolchain'inin açıldığı ve PATH ortam değişkeninin uygun şekilde ayarlandığı varsayılmıştır. Cross-Compiler prefix'i `arm-linux-gnueabihf-` şeklinde olup **EABIHF** yani **Hard-Float** türünde bir toolchain'dir.

Busybox uygulamasını hedef platform için çapraz derlemek istediğimizde, kernel derleme sürecindeki benzer şekilde `ARCH` ve `CROSS_COMPILE` değişkenlerine atama yapmamız gerekecektir:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j 8
```

Bu şekilde çapraz derleme süreci tamamlanacaktır.

İşlem bitiminde busybox uygulamasını tüm gerekli linkleriyle birlikte bir dizin yapısı içerisinde (aslında kök dosya sistemimizin ilk şablonu da diyebiliriz) `make install` komutuyla oluşturabiliriz. Bu komut eğer `CONFIG_PREFIX` değişkenine `.config` dosyası üretimi sürecinde farklı bir değer atanmadı ise bulunulan dizin altında, `_install` şeklinde bir alt dizin açacak ve tüm linkleri ve uygulama dosyasını bu alt dizinlerde oluşturacaktır.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- CONFIG_PREFIX=/opt/cross/rootfs ins
tall
```

Yukarıdaki örnekte ARM mimarisi için çapraz derleme işlemi sonrası busybox uygulaması ve gerekli linkleri, `/opt/cross/rootfs` dizini altında oluşturulacaktır.

NFS Üzerinden Sistemi Açarak İlerleme

> Not: Bu noktada NFS Sunucu Kurulumu ve NfsRoot bölümlerini henüz okumadı iseniz incelemeniz önerilir. Kök dosya sistemi ile ilgili burada verilen örnekler **BeagleBoneBlack** cihazı üzerinde çalıştırılmıştır.

Örneklerimizde öntanımlı `beaglebone_defconfig` ile derlenmiş çekirdek, aşağıdaki gibi **U-boot** komutları ile açılmaktadır:

```
setenv serverip 192.168.100.1
setenv ipaddr 192.168.100.5
setenv console ttyO0,115200n8
setenv rootpath /opt/cross/rootfs
setenv bootargs console=${console} root=/dev/nfs nfsroot=${serverip}:${rootpath},vers=
3 rw ip=${ipaddr}
tftp 0x80200000 uImage-dtb.am335x-boneblack
bootm
```

Yukarıdaki gibi minik dosya sistemimizi `/opt/cross/rootfs` dizini altında hazır ettikten sonra, bu dizini NFS sunucumuz üzerinden paylaşıma açıp, cihazımız üzerinden yeni oluşturduğumuz bu dosya sistemiyle açılış yapmayı deneyelim.

```
[    5.320738] IP-Config: Guessing netmask 255.255.255.0
[    5.326212] IP-Config: Complete:
[    5.329625]      device=eth0, hwaddr=d0:5f:b8:ef:61:07, ipaddr=192.168.100.5, mask=
255.255.255.0, gw=255.255.255.255
[    5.340641]      host=192.168.100.5, domain=, nis-domain=(none)
[    5.346837]      bootserver=255.255.255.255, rootserver=192.168.100.1, rootpath=
[    5.354377] ALSA device list:
[    5.357663]   #0: TI BeagleBone Black
[    5.369033] VFS: Mounted root (nfs filesystem) on device 0:12.
[    5.375638] devtmpfs: error mounting -2
[    5.379967] Freeing init memory: 216K
[    5.394642] Kernel panic - not syncing: No init found.  Try passing init= option to
 kernel. See Linux Documentation/init.txt for guidance.
```

Açılış sürecimizin yukarıda belirtilen şekliyle sonlandığını gördük.

Daha önceki aşamalarda, Linux çekirdeğinin açılıştaki tüm işlemleri başarılı olarak tamamlaması halinde son 2 işlem olarak, kök dosya sistemini mount edip, kök dosya sistemi üzerinden bir adet uygulamayı başlatacağını ifade etmiştik. Yukarıdaki çıktıya baktığımızda kök dosya sisteminin NFS üzerinden başarılı bir şekilde mount edildiğini görmekteyiz:

```
[    5.369033] VFS: Mounted root (nfs filesystem) on device 0:12.
```

Fakat daha sonra `init` uygulamasının çalıştırılamadığını ve kernel panic durumu oluştuğunu görmekteyiz:

```
[    5.394642] Kernel panic - not syncing: No init found.
```

Neden böyle bir problem oluştu?

Problemin nedeni üretmiş olduğumuz dosya sistemimizde `/bin/busybox`'ı gösteren bir çok link olmasına rağmen, toplamda sadece **1** adet binary uygulama bulunmakta, paylaşımlı kütüphane dosyası ise hiç bulunmamaktadır. Hatta paylaşımlı kütüphaneleri tutacağımız `lib` dizini dahi henüz sistemimizde mevcut değildir. Yapmamız gereken, hedef platform için çapraz derleme yaptığımız `busybox` uygulamamızın bağımlı olduğu kütüphaneleri bularak, kök dosya sistemimizdeki `lib` dizini altına kopyalamaktır.

Öncelikle derlemiş olduğumuz `busybox` binary dosyasının kütüphane bağımlılıklarını bulalım. Bunun için `readelf` uygulamasını `-d` parametresi ile çağırabiliriz:

```
$ readelf -d busybox | grep Shared
  0x00000001 (NEEDED)       Shared library: [libm.so.6]
  0x00000001 (NEEDED)       Shared library: [libc.so.6]
```

Görüldüğü üzere `libc.so.6` ve `libm.so.6` kütüphanelerine ihtiyaç duymaktadır. Öncelikle kök dosya sistemimizde henüz mevcut olmayan `lib` dizinini oluşturalım:

```
$ mkdir /opt/cross/rootfs/lib
```

Sonrasında ihtiyaç duyulan 2 kütüphaneyi kopyalayalım. Peki bu kütüphaneleri nereden kopyalayacağız? Kendi geliştirme bilgisayarımızın `/lib` dizini altındakileri kopyalayamayız.

Oluşturacağımız kök dosya sistemine zaman zaman bu şekilde gereken kütüphaneleri kopyalamamız gerekecektir. Bu işlemleri çoğunlukla öncelikle ilgili kütüphaneyi ayrı bir yerde hedef platform için derleyerek, sonrasında kütüphane dosyalarını kök dosya sistemimiz altına kopyalayarak gerçekleştireceğiz.

Ancak `libc.so.6`, `libm.so.6` gibi temel kütüphaneler zaten kullandığımız toolchain içerisinde yer almaktadır. Bu kütüphaneleri yeniden derlemek, bir tür toolchain üretim süreci de gerektirdiğinden, çapraz derlemeyle ilgili bölümde de anlattığımız üzere yeniden üretmek yerine toolchain içerisinden çıkan versiyonlarını kullanmak daha sağlıklıdır.

Kullandığımız ARM eabihf toolchain'i için bu dosyalar, toolchain ana dizinini referans aldığımızda, `./arm-linux-gnueabihf/libc/lib/arm-linux-gnueabihf` dizini altında yer alır. Bulmakta sorun yaşarsanız toolchain ana dizininde iken:

```
$ find -name libc.so.6
```

gibi bir komutla da arama yapıp bulabilirsiniz.

Şimdi her iki kütüphaneyi kök dosya sistemimize kopyalayalım:

```
$ cp ./arm-linux-gnueabihf/libc/lib/arm-linux-gnueabihf/libc.so.6 /opt/cross/rootfs/li
b/
$ cp /arm-linux-gnueabihf/libc/lib/arm-linux-gnueabihf/libm.so.6 /opt/cross/rootfs/lib
/
```

Sistemimizi tekrar NFS üzerinden açtığımızda problemin devam ettiğini göreceğiz. **Busybox** uygulamamızın ihtiyaç duyduğu kütüphaneleri kök dosya sistemimize attık ancak paylaşımlı kütüphane bağımlılığı olan herhangi bir uygulamayı çalıştırabilmek için aynı zamanda `ld.so` kütüphanesine de ihtiyaç bulunmaktadır. Bu kütüphane toolchain versiyonlarına göre `ld-linux.so.2`, `ld-linux.so.3`, `ld-linux-armhf.so.3` gibi isimlerde bulunabilir. Bizim kullandığımız ARM eabihf toolchaini içerisinde bu dosya, `ld-linux-armhf.so.3` şeklindedir ve bunun da kök dosya sistemine kopyalanması gereklidir:

```
$ cp ./arm-linux-gnueabihf/libc/lib/ld-linux-armhf.so.3 /opt/cross/rootfs/lib/
```

ARM eabihf toolchain ile çalışan sistemimiz için son bir işlem daha yapmamız gerekiyor. `ld-linux-armhf.so.3` loader uygulaması, aynı anda eabihf ve eabi sistemleri destekleyebilmek için, yükleyeceği diğer kütüphaneleri `/lib/arm-linux-gnueabihf` dizini altında arar. Aynı anda her iki ARM ABI ile çalışmak pek karşılaşılan bir durum değildir. Bu nedenle genellikle `/lib/arm-linux-gnueabihf` dizini `/lib` dizinini gösteren bir sembolik link olarak oluşturulur:

```
$ cd /opt/cross/rootfs/lib && ln -s . arm-linux-gnueabihf
```

Artık tüm bileşenlerimiz hazır görünüyor. Cihazımızı tekrar NFS üzerinden açmayı denediğimizde, bu defa açılış sürecinin gerçekleştiğini görmekteyiz.

`/etc/inittab` İyileştirmeleri

Artık açılan bir sistemimiz var. Fakat göreceğiniz üzere seri konsol ekranımızda aşağıdaki mesajlar sürekli olarak çıkmakta:

```
can't open /dev/tty2: No such file or directory
can't open /dev/tty3: No such file or directory
can't open /dev/tty4: No such file or directory
can't open /dev/tty2: No such file or directory
can't open /dev/tty3: No such file or directory
can't open /dev/tty4: No such file or directory
...
```

Sistemimiz açılmış ve çalışıyor durumda olmasına rağmen bu hata mesajları yüzünden konsolu efektif kullanamıyoruz. Buradaki problem sistemimizde henüz bir `/etc/inittab` dosyası olmadığı için busybox init uygulamasının öntanımlı bir `inittab` dosya içeriği varmış gibi çalışmasından kaynaklanıyor. `/dev/ttyXX` aygıtları sistemimizde yer almadığından uygulamayı çalıştıramamakta ve tekrar tekrar yaptığı denemelerden kaynaklanan hata mesajları görüntülenmektedir.

Örnek bir `inittab` dosyasını busybox kaynak kodlarını açtığımız dizinde, `examples/inittab` yolunda bulabiliriz. Bu dosyanın içeriği **Init Süreci**'nin anlatıldığı bölümde detaylandırılmıştır.

Sistemimiz için aşağıdaki `inittab` dosyasını kullanacağız:

```
::sysinit:/etc/acilis
::askfirst:-/bin/sh
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/etc/kapanis
```

Yukarıdaki içeriğe sahip dosyayı `/opt/cross/rootfs/etc/inittab` şeklinde oluşturup ardından sitemimizi tekrar açmayı denediğimizde, daha önce aldığımız hata mesajlarının ortadan kalktığını ama yenilerinin geldiğini görmekteyiz:

```
can't run '/etc/acilis': No such file or directory

Please press Enter to activate this console.
getty: can't open '/dev/null': No such file or directory
getty: can't open '/dev/null': No such file or directory
getty: can't open '/dev/null': No such file or directory
getty: can't open '/dev/null': No such file or directory
getty: can't open '/dev/null': No such file or directory
...
```

`/etc/acilis` dosyamızı henüz hazırlamadığımız için onunla ilgili aldığımız hata normaldir. Ancak sonrasında sürekli `/dev/null` dosyasının yer almadığına dair hatalar almaktayız ve bu hata mesajları yüzünden gene konsolumuzu kullanamıyoruz. Kullandığımız çekirdek içerisinde **DEVTMPFS** ve **DEVTMPFS_MOUNT** desteği açıktır dolayısıyla `/dev` dizini altındaki aygıt dosyalarının çekirdek tarafından otomatik üretilmesi gerekirdi (detaylar için **Devtmps Dosya Sistemi** bölümüne bakınız). Buna rağmen `/dev` dizini altına bu özel dosya sisteminin çekirdek tarafından otomatik olarak bağlanamadığını görüyoruz. Bunun çok basit bir nedeni var: minik dosya sistemimizde henüz `/dev` dizini mevcut değil.

Mount işleminin başarılı olabilmesi için gerekli şartlardan biri, mount edilecek dizinin de öncesinde sistemde bulunmasıdır (örneğimizde `/dev` dizini).

Bu sorunu çözmek için kök dosya sistemimizde aşağıdaki komutla `/dev` dizinini oluşturalım:

```
$ mkdir /opt/cross/rootfs/dev
```

Sonrasında sistemi tekrar NFS üzerinden açalım:

```
[    5.368528] VFS: Mounted root (nfs filesystem) on device 0:12.
[    5.375468] devtmpfs: mounted
[    5.378895] Freeing init memory: 216K
can't run '/etc/acilis': No such file or directory

Please press Enter to activate this console.
/ #
```

Nihayet sistemimizi görece uygun bir şekilde açmayı başardık. Mesajları incelediğimizde `devtmpfs` mount işleminin gerçekleştiğini görüyoruz. `/etc/acilis` betiğimiz henüz hazır olmadığından hata alıyoruz ancak konsola düşmeyi başardık. `ls /dev` komutuyla `/dev` dizini altındaki dosyaları görebiliriz.

Şimdi çalışan uygulamaların listesini `ps` komutuyla almayı deneyelim:

```
/ # ps
PID   USER     TIME   COMMAND
ps: can't open '/proc': No such file or directory
```

Görüldüğü üzere `/proc` dizininin sistemde olmadığına dair bir hata alıyoruz ve process listesini de göremiyoruz.

Sistemimizi yeniden başlatmayı deneyelim:

```
/ # reboot
reboot: can't open '/proc': No such file or directory
```

Bu işlem için de `/proc` dizinine ihtiyaç olduğu görülüyor. Çalışan bir Linux sisteminde `proc` dosya sisteminin `/proc` altına mount edilmiş durumda olması önemlidir. Bu dizin altındaki pek çok dosya ve dizin üzerinden, kullanıcı kipindeki uygulamalar çekirdek ile haberleşebilir ve bir takım işlemleri gerçekleştirebilirler.

Şu ana kadar `proc` dosya sistemini mount etmiş olmamamıza rağmen sistemimiz -yeterince işlevsel olmasa da- çalışıyordu, dolayısıyla bu tarz işlemler için çekirdek tarafından bir yardım beklememeliyiz. Bu süreci kullanıcı kipinde gerçekleştirmemiz lazım. Sistem her açıldığında `mount` komutu ile `proc` dosya sistemini `/proc` dizini altına bağlamalıyız.

`/dev` dizini için yaptığımız örnekte olduğu gibi, öncesinde `/proc` dizinini kök dosya sistemimizde oluşturalım:

```
$ mkdir /opt/cross/rootfs/proc
```

Şimdi `/etc/acilis` betiğini `/opt/cross/rootfs/etc/acilis` ismiyle oluşturmaya başlayalım:

```
#! /bin/sh
echo "Sistem acilisi basladi"
echo "proc dosya sistemi baglaniyor"
mount -t proc none /proc
```

Dosyayı oluşturduktan sonra çalıştırma haklarını vermemiz gerekiyor:

```
$ chmod 755 /opt/cross/rootfs/etc/acilis
```

Tekrar sistemi NFS üzerinden açtığımızda hiç hata almadan sistemin açıldığını görebiliriz:

```
[    5.374822] devtmpfs: mounted
[    5.378254] Freeing init memory: 216K
Sistem acilisi basladi
proc dosya sistemi baglaniyor

Please press Enter to activate this console.
/ #
```

Artık `ps`, `top`, `ifconfig`, `reboot` vb. pek çok uygulama olması gerektiği gibi çalışmaya başlayacaktır. `reboot` komutunu tekrar deneyelim:

```
/ # reboot
can't run '/etc/kapanis': No such file or directory
The system is going down NOW!
Sent SIGTERM to all processes
```

Reboot gerçekleşti ancak `/etc/kapanis` uygulaması henüz hazır olmadığından bir hata mesajı aldık. Bu dosyayı da oluşturalım. Temel olarak kapanış sürecinde, tüm mount edilmiş dosya sistemlerinin unmount edilmesini sağlamamız yerinde olacaktır. Unmount işlemi Linux *Page Cache* tablolarının depolama ortamlarına yazılmasını tetikler, böylece kapanış sırasında veri kaybı sorunu yaşamazsınız. Aksi takdirde **Write-Back** mekanizması nedeniyle veri kayıplarıyla karşılaşabilirsiniz.

```
#! /bin/sh

echo "Sistem kapatiliyor"
umount -a -r
```

Aynı şekilde çalıştırma haklarını vermemiz gerekiyor:

```
$ chmod 755 /opt/cross/rootfs/etc/kapanis
```

Initramfs İle Erken Kullanıcı Kipi

Önceki bölümlerde Linux sistemlerinin genel açılış sürecini inceledik. Linux çekirdeği açılış sürecinin son önemli adımı olarak, kullanıcı kipinde bir adet uygulamayı çalıştırmak zorundadır. Doğal olarak bu uygulamayı çalıştırmadan önce, uygulamanın bulunduğu dosya sistemi (kök dosya sistemi) *mount* edilmiş olmalıdır.

Kök dosya sisteminin fiziksel olarak nerede bulunduğu ise kernel tarafından önceden biliniyor olmalıdır. Bu bilgi kernel derleme sürecinde statik olarak kernel obje kodları içerisine gömülebileceği gibi boot yükleyici uygulama üzerinden kernel açılış parametresi olarak da belirtilebilir. Her iki durumda da kernel, kök dosya sisteminin hangi aygıt (NFS dahil) üzerinde bulunduğuna dair tek bir değeri biliyor olacaktır.

Kernel kök dosya sistemi olarak tek bildiği yeri ya başarılı olarak mount edecek ya da *VFS: unable to mount root fs* şeklindeki bir hata mesajı ile açılış sürecini sonlandıracaktır.

Şimdi bu bilgiler ışığında aşağıdaki sorulara yanıtlar aramaya çalışalım:

- Sistemde kullanılan kök dosya sisteminin türüne ait dosya sistemi desteği ve ilgili fiziksel aygıta erişmek için gereken destekler, kök dosya sistemi *mount* edilmeden önce kernel tarafında hazır olmalıdır. Yani kök dosya sistemi olarak *Ext4* kullanılan bir sistemde *Ext4* desteği, *XFS* kullanılan bir sistemde de *XFS* desteği hazır olmalı, aynı şekilde dosya sistemi fiziksel olarak *MMC* aygıtı üzerinde ise *MMC* katmanı, *NFS* üzerinde ise *IP ve NFS* katmanı kernel tarafında hazır edilmelidir. Bunlardan herhangi biri eksik olduğunda, kök dosya sistemi mount edilemez ve sistem açılışına devam edemez. Tüm bu destekleri hazır hale getirmenin bir yolu, ilgili destekleri kernel kodu içerisine statik olarak eklemekten (modül yapmamak) geçer. Ancak yüzlerce farklı kombinasyonda çalışmasını beklediğiniz bir Linux dağıtımı için öntanımlı kernel üretmeye çalışıyorsanız, tüm bu destekleri kernel imajı içerisine atmak gibi bir zorunlulukla karşılaşırsınız. Bu da kernel imajını gereksiz bir şekilde büyütecektir. Bu sorunu nasıl çözebiliriz?
- Gömülü sistemimizde yedek amaçlı ikinci bir kök dosya sistemi tuttuğunuzu düşünelim. Açılış sürecinde kernel birinci dosya sistemini mount edemez (dosya sistemi bütünlüğü bozulmuş olabilir) veya dosya sistemi içerisinde bir takım anormallikler tespit ederse otomatik olarak yedek olarak tuttuğunuz kök dosya sisteminden açılışa devam etsin istiyorsunuz. Bunu nasıl yapabilirsiniz?
- Sistemimizde açılış sırasında özel bir USB stick takmak ve özel bir takım güvenlik kontrolleri yapmak suretiyle, tüm sistem yazılımlarının (kök dosya sistemi ve diğer yazılımlar) güncellenmesi nasıl mümkün olabilir?

- Kök dosya sistemini `dm-crypt` veya benzeri bir *encrypted* dosya sistemi ile birlikte nasıl kullanabiliriz? Bu tarz bir dosya sisteminin kullanılabilmesi için öncesinde gereken `cryptsetup` vb. uygulamalarını kök dosya sistemi mount edilmeden nasıl çalıştıracağız?
- Asıl veya yedek kök dosya sistemlerimizin her ikisinde birden sorun olması veya sistemin bir şekilde kök dosya sistemine erişememesi durumunda kernel panic durumuna düşmek yerine, otomatik olarak recovery amaçlı kullanılabilecek bir ortam sağlanabilir mi? Hatta bu recovery ortamına SSH vb. bir protokolle uzaktan bağlanmak mümkün olur mu?

Soruları daha da artırabiliriz. Dikkat edilirse bu sorulara olumlu yanıtlar verebilmek için, Linux kernel tarafında kök dosya sistemini mount etme işleminden önce, sorumuza bağlı olarak özel bir takım ek işlemler yapılması gerektiği görünmektedir. Sözgelimi `dm-crypt` ile encrypted bir kök dosya sistemi kullanıyorsak, kernel tarafından ön hazırlıkların yapılması gerekiyor. Eğer `dm-crypt` yerine `encfs` kullanmak istersek, bu defa kernel tarafında başka işlemlerin yapılması gerekecekti. Yukarıdaki soruların her biri ve burada listelemediğimiz diğer pek çok soru için Linux kernel tarafında ayrı ayrı ek destek gerekiyor. Böyle bir işlem kernel tarafında mümkün müdür?

Şimdiye kadar öğrendiklerimizi bir kenara atmamıza gerek yok, Linux kernel açılış sürecinin son kısımları oldukça basittir: kök dosya sistemini mount et ve içerisinden bir adet uygulamayı çalıştır.

Yukarıdaki sorunlara çözüm üretebilmek için Linux kernel tarafında karmaşık işlemler yapmak yerine, daha basit bir yaklaşım geliştirilmiştir: 2 aşamalı açılış yöntemi

Bu yöntem başlangıçta genelde sadece Linux dağıtımı geliştiricileri için önemliydi ancak günümüzde pek çok gömülü sistemde de önemli kullanım alanları bulmaktadır.

Initrd - Initial Ramdisk

2 aşamalı açılış sisteminin öncülü `initrd` sistemidir. Çekirdeğin çok eski versiyonlarında dahi bu destek bulunmaktadır.

Initrd imajı esasen çalışabilen küçük bir kök dosya sistemidir. Ayrıca hazırlanır ve genellikle *ext2* ya da *cramfs* gibi bir dosya sisteminin hazırlanması ve sıkıştırılmasıyla `cpio` arşivi şeklinde oluşturulur.

Örnek olarak kendi kullandığımız Linux bilgisayarımızdaki *initrd* imajının içeriğine bakmayı deneyelim. Örneğimizde Debian Jessie 64bit versiyonundaki `/boot/initrd.img-3.16-2-amd64` dosyasını kullanacağız. Dosyanın boyutunun **15 MB** olduğunu görmekteyiz. `file` komutu ile dosya hakkında bilgi edinmeye çalışalım:

```
$ file /boot/initrd.img-3.16-2-amd64
  /boot/initrd.img-3.16-2-amd64: gzip compressed data,
  last modified: Wed Dec 10 00:23:17 2014, from Unix
```

`.gz` gibi bir uzantı verilmiş olmamakla birlikte dosyanın `gzip` ile sıkıştırılmış olduğu görülmektedir. Öncelikle dosyamızı açmalıyız:

```
$ zcat /boot/initrd.img-3.16-2-amd64 > /tmp/initrd
```

Dosyayı `/tmp/initrd` şeklinde sistemimize açtık ve boyutunun **45 MB**'a çıktığını gördük. `file` komutuyla bu dosyaya baktığımızda:

```
$ file /tmp/initrd
  initrd: ASCII cpio archive (SVR4 with no CRC)
```

şeklinde dosyanın bir `cpio` arşivi olduğunu öğreniyoruz. Boş bir dizin oluşturup, `cpio -id` parametresiyle bu arşivi yeni oluşturduğumuz dizin içerisine açabiliriz:

```
$ mkdir /tmp/image
$ cd /tmp/image
$ cpio -id < /tmp/initrd
90871 blocks
$ ls
bin  conf  etc  init  lib  lib64  run  sbin  scripts
```

Initrd imajının içerisine görebilmiş olduk. Burada yer alan `init` uygulaması initrd açılış yönteminde kullanılan ilk uygulamadır. Çoğunlukla bir kabuk uygulaması olduğundan herhangi bir metin editörü ile içeriğini açıp inceleyebiliriz.

Initrd imajı bu örnekte olduğu gibi kernel imajından ayrı bir dosyada tutulur. Boot yükleyici uygulama initrd açılış sürecinde öncelikle bulunduğu ortamdan kernel imajını belleğe yükler. Sonrasında benzer şekilde initrd imajını da bulunduğu yerden belleğe yükler ve kernel açılışında initrd imajının bellekte bulunduğu adresi kernel tarafına bildirir. Kernel bu şekilde başlatıldığında kök dosya sistemini geleneksel biçimde mount etmeyi denemek yerine, initrd imajının içeriğini `RAM Disk` yöntemiyle belleğe açar ve açtığı yeri kök dosya sistemi olarak kullanır. Geleneksel açılışta kök dosya sistemi mount edildikten sonra öntanımlı olarak `/sbin/init` çalıştırılıyorken, initrd sistemiyle açılış gerçekleştirildiğinde kök dizindeki `/init` uygulaması çalıştırılır.

Özetle Linux kernel tarafında daha önce öğrenmiş olduğumuz açılış sistematiği korunmakta, gene sadece bir adet kök dosya sistemi mount edilmekte ve içerisinden bir adet uygulama çalıştırılmaktadır.

Ancak root dosya sistemi fiziksel bir medya üzerinden değil, boot yükleyici tarafından öncesinde belleğe aktarılmış bir imaj üzerinden gerçekleşmekte ve ilk çalışan uygulama `/init` olmaktadır.

Bu yöntem **erken kullanıcı kipi** (early user-space) olarak da bilinmektedir.

Çift aşamalı açılış sürecinde kernel tarafından hazır edilen dosya sistemi ve `/init` uygulamasının çeşitli kontrol vb. işlemleri yaptıktan sonra asıl kök dosya sistemini öncelikle örneğin `/mnt` gibi bir dizine mount etmesi, sonra da kök dosya sistemini (`/`) ilgili yere (`/mnt`) kaydırması beklenmektedir.

Initramfs - Initial Ram FileSystem

Initramfs yöntemi küçük detaylar haricinde initrd süreciyle aynı şekilde gerçekleşir. Initrd için yazdıklarımız genel olarak burada da geçerlidir.

Initramfs sürecindeki temel farklılık, initrd yaklaşımındaki ayrı bir initrd imaj dosyası kullanmak yerine, imajın da kernel içerisine eklemlenmiş olmasıdır.

Bu sayede kernel imajını ayrı initrd imajını ayrı yönetmek yerine, ya hep ya hiç şeklinde boot yükleyici tarafından kernel imajının belleğe yüklenmesi başarıldı ise, her durumda erken kullanıcı kipine ulaşmak mümkün olmakta; initrd imajının bozulmuş olması veya boot yükleyici tarafından diskten yüklenememesi gibi sorunlar ortadan kalkmaktadır.

Linux kernel **2.6** ve sonraki serilerde *initrd* yerine *initramfs* sisteminin kullanımı tercih edilmektedir.

Her iki imaj türü de, *Kök Dosya Sistemi Oluşturma* başlıklı bölümde anlatılan yöntemlerle üretilebilir. Initramfs imajı için bu şekilde küçük bir dosya sistemi hazırladığımızda, derleyeceğimiz kernel içerisine bu imajı eklememiz gereklidir.

Bunun için ya imajımızı tek dosyalık `cpio` arşivi haline getirmeli ya da hazırladığımız ana dizini kernel derleme sürecinde belirtmeliyiz. Bu işlem için kernel tarafında `CONFIG_INITRAMFS_SOURCE` değişkeni kullanılır.

Açılış Süreci

Her iki yöntemde kernel tarafından çalıştırılan ilk uygulama `/init` şeklindedir.

`/init` uygulamasındaki yapabileceklerimizin sınırı, initramfs kök dosya sistemine dahil ettiğimiz araçlarla sınırlıdır. Çok küçük bir imaj yapılabileceği gibi çok daha gelişmiş bir dosya sistemi ve uygulamalardan oluşan bir imak da üretilebilir.

Bu noktada konuya başlarken sorduğumuz soruların tümüne cevaplar üretebiliriz zira artık kernel katmanında değil, kullanıcı kipinde çalışmaktayız ve istediğimiz her türlü uygulamayı sistemimize dahil edebilir veya geliştirip kullanabiliriz.

Örnek olarak encrypted kök dosya sistemi kullanacaksak, gerekli araçları initramfs imajımızın içerisine dahil edip, asıl dosya sistemini mount etmeyi denemeden önce sistemimizi uygun şekilde hazırlayabiliriz.

Yedek dosya sistemi senaryosunda, asıl dosya sistemini mount etme işlemimiz başarısız olursa veya mount etmemize rağmen içerisindeki belirli dosyaların *md5sum* değerleri beklediğimizden farklı ise, *unmount* işlemiyle vazgeçebilir ve yedek dosya sistemini mount edebiliriz.

USB veriyolunda belirli özelliklere sahip bir disk var ise, içerisinde güvenlik/imza kontrolü de yapıp sistemi içerisindeki dosyalarla güncelleyebilir veya sistemin bir yedeğini ilgili disk üzerine geri kopyalayabiliriz.

Bu şekilde sınırsız sayıda senaryo üretilebilir. Artık kullanıcı kipinde çalıştığımız için senaryo sayısının bizim için (ve de kernel için) bir önemi yoktur. Herhangi bir işlem artık yapılabilir durumdadır.

Bu şekildeki 2 aşamalı açılış süreçlerinde, `/init` uygulamasından son olarak asıl kök dosya sistemini belirlemesi ve sistemin initramfs imajını gösteren kök dizinini, asıl kök dosya sistemini gösterecek şekilde değiştirmesi beklenir.

Aşağıda örnek bir `/init` betiği görülmektedir:

```
#!/bin/sh

# get_opt("init=/sbin/init") will return "/sbin/init"
get_opt() {
    echo "$@" | cut -d "=" -f 2
}

export PATH=/bin:/sbin:/usr/bin:/usr/sbin

[ -d /dev ]  || mkdir -m 0755 /dev
[ -d /root ] || mkdir -m 0700 /root
[ -d /sys ]  || mkdir /sys
[ -d /proc ] || mkdir /proc

# devtmpfs does not get automounted for initramfs
mount -t devtmpfs devtmpfs /dev

mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t tmpfs tmpfs /tmp
```

```
# Sistem spesifik
# USB
modprobe musb_hdrc
modprobe ti81xx
modprobe sd_mod
modprobe usb-storage

modprobe omap_hsmmc
modprobe mmc_block
modprobe ext4

echo "# Checking usb startup disk"
sleep 3

mkdir -p /mnt/usb

mount -t vfat /dev/sda /mnt/usb 2> /dev/null || \
  mount -t vfat /dev/sda1 /mnt/usb 2> /dev/null

if [ -e "/mnt/usb/upgrade/run.sh" ]; then
    echo "SCRIPT CALISTIRILIYOR..."
    sh /mnt/usb/upgrade/run.sh
    umount /mnt/usb
fi

# Defaults
init="/sbin/init"
root="/dev/mmcblk0p2"
mnt_point="/mnt/rootfs"

# Process command line options
for i in $(cat /proc/cmdline); do
    case $i in
        root\=*)
            root=$(get_opt $i)
        ;;
        init\=*)
            init=$(get_opt $i)
        ;;
    esac
done

# Mount the root device
mount "${root}" $mnt_point

#Check if $init exists and is executable
if [[ -x "$mnt_point/${init}" ]] ; then
    mount --move /sys $mnt_point/sys
    mount --move /dev $mnt_point/dev
    mount --move /tmp $mnt_point/tmp

    #Switch to the new root and execute init
    exec switch_root $mnt_point "${init}"
```

```
fi

#This will only be run if the exec above failed
echo "Failed to switch_root, dropping to a shell"
exec sh
```

`/init` betiğimiz çok daha karmaşık işlemler yapıyor olabilir. Burada fikir vermesi açısından ufak bir implementasyon yapılmış ve özellikle bir kaç ufak iyileştirme yapılabilecek nokta bırakılmıştır. Bunların neler olabileceğini bulmaya çalışmak öğretici olabilir.

Initramfs süreciyle ilgili önemli bir not eklemekte fayda görüyoruz. Bu şekilde açılış gerçekleştirildiğinde, ilerleyen aşamalarda değineceğimiz `devtmpfs` dosya sistemi henüz mount edilmiş durumda değildir. Dahası bu mount işlemini bizim yapmamız gerekmektedir. `/init` betiğinin ilk satırlarında bu bölümü görmekteyiz. Ancak bir sorun daha var ki, `/init` uygulaması çalıştırılırken `/dev/console` aygıt dosyasına erişim de yapılmakta ve henüz devtmpfs üzerinden `/dev` dizini hazır hale getirilmemiş olduğundan bu dosyaya ulaşılamamakta ve `/init` uygulaması çalıştırılamamaktadır. Bu senaryo ile karşılaştığınızda kernel açılış mesajlarında aşağıdaki gibi bir hata görürsünüz:

```
Warning: unable to open an initial console.
```

Bu sorunun üstesinden gelebilmek için, initramfs imajını ürettiğiniz kök dosya sistemindeki `/dev` dizini altına `console` özel aygıt dosyasını aşağıdaki gibi oluşturabilir:

```
$ sudo mknod -m 622 /path/to/initramfs/dev/console c 5 1
```

veya `cp -a` komutuyla kendi sisteminizdeki `/dev/console` dosyasını da kopyalayabilirsiniz.

Örnek `/init` uygulamamız ilerleyen zamanda daha detaylı açıklanacaktır. Kod üzerinden takip edip ne yapmaya çalıştığımızı anlayabilirsiniz. Bu süreçteki en önemli komutlar `switch_root` ve `mount --move` ile başlayan satırlarda yer almaktadır.

Devtmpfs Dosya Sistemi

Çalışan bir Linux dosya sisteminin en temel bileşenlerinden biri `/dev` dizini altında yer alan aygıt dosyalarıdır. Dosya sisteminde normal bir dosya gibi görünen bu dosyalar, diğerlerinden farklı olarak bir adet **MAJOR** bir adet de **MINOR** numarası içerirler. Linux çekirdeği tarafından dosyaların isimlerinin bir anlamı yoktur, önemli olan ilişkili oldukları major:minor numaralarıdır.

Örnek olarak `/dev/ttyS0` özel dosyasının major numarası **4**, minor numarası **64**'tür. Aynı major:minor numaralarına sahip `/dev/seriport` isminde bir aygıt dosyası üretilirse, hangi dosya üzerinden erişim yaparsak yapalım, gerçekte hep aynı aygıt üzerinde çalışmış oluruz (örneğimizde birinci seri port).

Sistemde çalışan uygulamalar `/dev` dizini altındaki fiziksel ve soyut aygıt dosyalarına ihtiyaç duyarlar. Kendisine yazılan tüm içeriği yok eden `/dev/null`, istendiği kadar rastgele veri üreten `/dev/random`, istendiği kadar NULL veri üreten `/dev/zero` vb. gibi dosyaların da kendilerine özgü bir major ve minor numaraları bulunmaktadır.

Özetle major:minor ikilisi, ilgili aygıt dosyasının çekirdek içerisindeki hangi katmanla ilişkili olduğunu ve hangi driver/fonksiyonların kullanılacağını da belirtmiş olur.

Linux sistemlerinde `/dev` dizini altındaki aygıt dosyalarının oluşturulması için 3 temel yöntem bulunur:

- `mknod` uygulamasıyla statik olarak önceden gerekli tüm aygıt dosyaları hazırlanabilir. Zahmetli bir süreçtir.
- `udev` servisi üzerinden çekirdek tarafından gönderilen mesajlar dinlenir ve yeni bir aygıt ilklendirildiğinde veya ortadan kalktığında üretilmesi/silinmesi gereken aygıt dosyaları `udev` tarafından düzenlenir.
- `devtmpfs` sistemiyle Linux çekirdeği tarafından, `tmpfs` dosya sisteminde mount edilmiş bir alan üzerinde otomatik olarak üretilebilir. Gömülü sistemler için oldukça uygundur, Linux dağıtımlarında da kullanılmaya başlandığı görülmektedir.

`devtmpfs` özelliği, Linux çekirdeğinin **2.6.32** versiyonuyla birlikte sunulmaya başlanmıştır. Bu özelliği destekleyen bir çekirdeğe sahipseniz diğer opsiyonlara bakmanıza hiç gerek yoktur. Bu yöntemin hem kullanımı kolay (neredeyse hiç bir işlem yapmak zorunda kalmıyoruz) hem de çalışma zamanında özellikle açılış sürecinde getirdiği ek yavaşlık sıfıra yakındır (*udev* modelinde ilk açılışta *sysfs* üzerinden parse işlemi yapıp tüm aygıtları bulması ve gerekli aygıt dosyalarını oluşturması 5-20 saniye arası alabilmektedir).

Bazı gömülü sistemlerle birlikte gelen öntanımlı çekirdek konfigürasyon dosyalarında, çoğu zaman üretici `devtmpfs` desteğinden habersiz olduğundan veya bir şekilde eski konfigürasyon dosyaları yeni özellikler dikkate alınmadan dağıtılmaya devam ettiğinden, `devtmpfs` desteğinin kapalı olarak geldiği görülmektedir. Bu gibi senaryolarda, konfigürasyon dosyasında `CONFIG_DEVTMPFS` ve `CONFIG_DEVTMPFS_MOUNT` seçeneklerini aktif hale getiriniz.

`CONFIG_DEVTMPFS` desteği, çekirdeğin `devtmpfs` özelliğiyle üretilmesini sağlar. Ancak sadece bu seçenek aktif ise, açılış sırasında `devtmpfs` özel dosya sistemini aşağıdaki gibi bir komutla sizin mount etmeniz gerekir:

```
mount -t devtmpfs none /dev
```

Ancak bu mount işleminden sonra sistemi kullanabilir olursunuz.

`CONFIG_DEVTMPFS_MOUNT` seçeneğini de aktifleştirdiğinizde, çekirdek kök dosya sistemini mount ettiğinde, otomatik olarak `devtmpfs` dosya sistemini de `/dev` dizini altına mount eder. Bu senaryoda açılış sürecinden bizim yukarıdaki gibi mount komutuyla ek bir işlem yapmamıza gerek kalmaz. Önerilen yöntem her iki seçeneğin de aktif edilmesidir.

Bu iki özellik sayesinde, `/dev` dizini altında sadece o an sistemde mevcut olan aygıtlara ilişkin dosyaların barındırılması garantilenmiş olur. İlgili aygıt çıkartıldığında veya çekirdek modülü kaldırıldığında, ilişkili aygıt dosyaları da çekirdek tarafından kaldırılır (örneğin usb seri port çeviricinin çıkartılması durumu).

NFS Root Mekanizması

NFS desteği Linux'ta çok uzun yıllardır bulunmaktadır. Linux sisteminizden ağ üzerindeki herhangi bir veya birden çok NFS paylaşımını, çeşitli dizinlere mount etmek suretiyle kullanabilirsiniz.

Ancak kök dosya sistemi NFS üzerinden kullanmak, biraz daha farklı bir senaryodur. Sistem açıldıktan sonra, NFS yardımcı araçlarını kullanmak suretiyle paylaşımların mount edilmesi senaryosuna göre önemli farklar içerir.

Bu süreç NFS Root mekanizması olarak adlandırılır ve özellikle gömülü sistemlerde sıklıkla kullanılır. Çıkış amacı başlangıçta disksiz istemcileri ağdaki bir NFS paylaşımı üzerinden başlatmak olsa da, gömülü Linux sistemlerinin yaygınlaşmasıyla kullanım alanı da genişlemiştir.

Kök dosya sistemini NFS üzerinden alabilmek için çekirdek içerisinde aşağıdaki destekler aktif olmalıdır:

1. TCP/IP katmanı
2. Çekirdek seviyesinde statik veya dinamikIP yapılandırması desteği
3. NFS istemci desteği
4. NFS Root desteği

Bunları açacak olursak, TCP/IP protokolü gereken iletişim protokollerinin temelini sağlama noktasında mecburen gereklidir. Statik veya dinamik IP yapılandırmasından kasıt, henüz kök dosya sistemi mount edilmeden IP atama işlevlerini gerçekleştirdiğimiz `ifconfig` gibi uygulamaları kullanamayacağımız için, Linux çekirdeğinin bize IP atama işlemleri için sunduğu olanaklar kümesidir. Her bir seçenek çekirdek derleme süreci içerisinde ayrı ayrı aktifleştirilebilmektedir. Statik IP ataması yapılabildiği gibi çekirdek seviyesinde basit bir DHCP istemci desteği de verilebilmektedir.

NFS istemci desteği, herhangi bir NFS paylaşımının mount edilebilmesi için gereklidir. NFS Root desteği ise, Linux çekirdeğinin normalde bir major:minor ile tanımladığı **root** dosya sisteminin ne olacağı bilgisini (`root=/dev/sda1` , `root=1f03` vb.) `/dev/nfs` gibi *pseudo* aygıt dosyası üzerinden alabilmesi için gereklidir.

Tipik bir nfs root senaryosunda çekirdek parametreleri aşağıdaki gibi olur:

```
root=/dev/nfs nfsroot=SERVER_IP:NFS_SHARE_PATH,vers=3 ip=CLIENT_IP
```

Çapraz Derleme ve Gerekli Ekipmanlar

Sistemimizde kullandığımız standart derleme araçları (gcc, g++ vb.) normal olarak aynı sistemde çalışacak obje kodları üretir. Derlenen uygulamada kullanılan kütüphaneler ve başlık dosyalarını öncesinde sistemde yer alır, derleme ve linkleme işlemi bu doğrultuda gerçekleşir. Sonrasında uygulama ilgili sistemde çalıştırılır.

Çapraz Derleme (Cross Compiling) sürecinde ise derleme işlemi sonucunda üretilecek obje kodlarının, işlemin yapıldığı sistemden bağımsız olarak başka bir hedef sistemde çalışması beklenir.

Derleme işleminin yapıldığı sistem **host**, derleme sonucunda oluşturulacak uygulamaların çalışacağı sistem ise **target** olarak isimlendirilir.

Örnek olarak Linux çalışan 64 bitlik X86 ailesi kişisel bilgisayarınızda ARM veya Windows platformu için derleme yaptığınızda, bilgisayarınız **host**, derleme sürecinin hedefi olan platform (örneğimizde ARM veya Windows) **target** olacaktır.

Bir hedef platform için çapraz derleme işlemini yapabilmek, sisteminizde ilgili hedef platform için obje kodları üretecek derleyici (gcc, g++), obje kodları üzerinde işlem yapabilecek çeşitli *binary* araçları (objdump, objcopy, readelf, strip vb.), temel C kütüphanesi ve ilgili başlık dosyalarının kurulu olmasını gerektirir. İşte tüm bu topluluk **Toolchain** olarak adlandırılır.

Toolchain Türleri ve İsimlendirme

Toolchain'ler genel olarak, ilgili hedef platformu tanımlayan bir ön ek ile isimlendirilirler. Örneğin ARM platformu için **arm-linux** ön eki kullanılabilir. Toolchain içerisinden çıkan tüm araçlarda bu ön ek bulunur: arm-linux-gcc, arm-linux-g++, arm-linux-as, arm-linux-objcopy vb.

Derleyicinin mutlaka gcc ailesi olması da gerekmez. Örneğin Linux bilgisayarınızda 32 bit Windows hedefli uygulama derlemek istiyorsanız, kullanacağınız C derleyicisi **i686-w64-mingw32** olacaktır.

Çeşitli projelerde çalışırken toolchain isimlendirmelerinde kafa karıştırıcı durumlarla karşılaşabilirsiniz. Örnek olarak aşağıdaki toolchain isimlerini ele alalım:

- arm-linux
- arm-none-linux-gnueabi
- arm-fsl-linux-gnueabi
- arm-none-eabi

- arm-linux-gnueabihf

Toolchain isimlendirmelerinde genellikle `arch [-vendor] [-os] - eabi` sistematiği takip edilir. Örneğin arm-none-linux-gnueabi toolchain'i için hedef mimari **arm**, üretici belirtilmemiş o yüzden **none**, işletim sistemi **linux** ve ABI versiyonu **gnueabi** olarak belirtilmek istenmiştir. Bu toolchain ile ARM Linux EABI binary formatında dosyaları üretebileceğimizi düşünebiliriz.

Diğer bir örneğe baktığımızda üretici alanında **fsl** ibaresini görmekteyiz. Bu da bize ilgili toolchain'in **FreeScale** firması tarafından üretildiğini gösteriyor.

Arm-none-eabi örneği ise diğerlerinden biraz daha farklı bir yerde duruyor, içerisinde hiç linux ibaresi geçmiyor yani işletim sistemi kısmı yok. Bu toolchain işletim sistemi olmayan bir ortamda doğrudan ARM işlemci üzerinde koşacak kodların üretimi için kullanılacaktır. Dolayısıyla bu toolchain ile kod üretirken harici kütüphaneler kullanmak ve linklemek (dinamik veya statik) mümkün olmayacaktır.

Arm-linux-gnueabihf toolchain'indeki **hf** ibaresi, ABI tipini göstermekte olup aşağıda bu konuya değinilmiştir.

Toolchain ABI (OABI, EABI)

Toolchain isimlendirmelerinde karşılaşabileceğiniz, oabi, eabi ve eabihf ibareleri, özellikle günümüzde popüler olan ARM platformları için **A**pplication **B**inary **I**nterface'in ne olduğunu göstermektedir.

OABI, ARM platformu için kullanılan ilk ABI versiyonudur. OABI, ilgili sistemin kayar noktalı işlemleri hızlı yapabilmek için bir **F**loating **P**oint **U**nit'e sahip olduğunu varsayar. Dolayısıyla böyle bir toolchain ile üretilecek olan obje kodlarında FPU Instruction'ları yer alacaktır.

Ancak özellikle ilk üretilen ARM platformlarında herhangi bir **FPU** birimi bulunmamaktaydı. Bu nedenle ilgili toolchain'ler üzerinden elde edilmiş olan uygulamalar, işlemcinin desteklemediği FPU Instruction'larını ürettiğinde CPU Exception durumu oluşmaktaydı.

İşlemci mimarilerinde **exception** ve **interrupt** mekanizmaları, beklenmedik bir olay ile normal akışı kesintiye uğratıp farklı bir şekilde ilerlenmesini sağlar.

Genel olarak **exception** durumu CPU içerisinde beklenmedik bir durumun oluşmasını, **interrupt** ise CPU dışında beklenmedik bir olay nedeniyle ortaya çıkar.

İlgili CPU mimarisi, **Exception Handling** için bir mekanizma tanımlamış ise, oluşan exception durumlarında sistemin çakılmasını engellemek ve bu durumu yönetmek mümkündür.

Exception durumları aşağıdaki olaylarda oluşur:

1. Aritmetik işlemler sonucunda elde edilen değerin register'a sığmaması ve taşma durumu
2. İşletim sistemi tarafından tetiklenen sistem çağrıları
3. CPU tarafından bilinmeyen bir instruction gelmesi

OABI toolchain ile derlenmiş uygulamalar ARM mimarisinde bu üçüncü durumu sıklıkla oluştururlar. CPU tarafından **exception** üretilir ve bu exception sistemin çakılmasını engellemek için Linux çekirdeği içerisinde, FPU instruction'ları yazılım yoluyla emüle edilir. Böylelikle sistemin çakılması engellenmiş olur.

Ancak bu süreç beraberinde önemli bir performans maliyeti de getirmektedir. Özellikle kayan nokta işlemlerinin sık kullanıldığı uygulamalarda sürekli bu şekilde CPU Exception üretimi ve bunun çekirdek tarafından emülasyonunun yapılması, *context-switch* nedeniyle performansı ciddi oranda düşürmektedir.

EABI, ARM platformu için yeni ABI versiyonudur. Yukarıda bahsedilen *context-switch* kaynaklı performans sorununu, emülasyon işlemini kullanıcı kipinde yapmak suretiyle çözmektedir. EABI için hazırlanmış toolchain'lerin ürettiği obje kodları içerisinde asla FPU Instruction'ları yer almaz. Bunun yerine ilgili Instruction'lar, derleme sürecinde ayrı fonksiyonlarla emüle edilirler. Bu şekilde üretilmiş bir uygulama hiç bir FPU Instruction üretmediğinden, CPU Exception durumu oluşmayacak, *context-switch* gerçekleşmeyecek ve çekirdek seviyesinde bu nedenle bir *exception handling* ve emülasyon yapılmak zorunda kalmayacaktır.

EABI toolchain'leri içerisinde bu destek ön tanımlı olarak aktive edilmiş olabileceği gibi, `-mfloat-abi=soft` şeklindeki derleyici parametresi ile de seçilebilmektedir. Yapılan testlerde FPU Instruction'ları yoğun kullanan uygulamaların, çekirdek seviyesinde emülasyon yerine kullanıcı kipinde bu şekilde emüle edilmesi halinde, 5 - 20 kat arası hızlanma sağladığı görülmüştür.

Yeni nesil ARM işlemcilerinde ise bu durum değişmeye başlamıştır. Çeşitli kayan nokta işlemlerini hızlandırmak için farklı bir instruction kümesine sahip olan **V**ector **F**loating **P**oint birimi ve sonrasında da özellikle matris ve vektör işlemlerini hızlandıran **NEON** birimi işlemcilerde kullanılmaya başlanmıştır. Donanım tarafında sağlanan bu destekler, `-mfloat-abi=soft` şeklinde derlenmiş olan uygulamalar için herhangi bir ek fayda sağlamayacaktır. Bu nedenle EABI için *hard floating point* destekli, **eabihf** toolchain'leri ortaya çıkmıştır.

Donanımınızın desteklemesi halinde sisteminizde yer alan tüm bileşenlerin (kütüphaneler ve uygulamalar) **eabihf** toolchain'i tarafından derlenmesi halinde, önemli oranda performans kazanımı elde edilebilmektedir. **eabihf** toolchain'lerinde `-mfpu=vfp` veya `-mfpu=neon` derleyici parametreleri ile donanımıza özgü hızlandırmaları aktive edebilirsiniz.

> NOT: **ABI** versiyonları için verdiğimiz örnekler ARM mimarisi için geçerlidir. Farklı mimarilerde farklı ABI'ler bulunmaktadır. Örneğin **MIPS** için **o32**, **o64**, **n32** ve **n64** olmak üzere 4 farklı ABI bulunmaktadır. Çok daha eski bir mimari olmasına rağmen, bunun gibi pek çok dokümanda örnek olarak dahi yer alamıyor olması, ARM mimarisinin gömülü Linux sistemler pazarındaki hakimiyetine dair fikir verebilir. Her geçen yıl bu fark açılmaktadır.

Toolchain Edinme

Toolchain'ler hazır biçimde indirilip kullanılabileceği gibi, sıfırdan bir toolchain üretmek de mümkündür.

Toolchain üretimi oldukça zahmetli ve uzun zaman alan bir süreçtir. Bu süreci kolaylaştırmayı amaçlayan, `crosstool` ve `crosstool-ng` projeleri bulunmakla birlikte çoğu durumda hazır bir toolchain ile projelere devam edilmesi önerilir. Kendi ürettiğiniz toolchain üzerinden ilgili mimari için üretilen kodlarda herhangi bir sıkıntı var ise, bunun algılanması ve çözülmesi çok ciddi zaman kayıpları getirebilir. Daha da kötüsü, geliştirme ortamında her şey yolundaymış gibi görünürken, proje sahaya çıktıktan sonra kaotik durumlar ile karşılaşabilirsiniz.

Hazır bir toolchain kullandığınızda böyle bir sorunla karşılaşma riskiniz çok daha düşüktür. Zira aynı toolchain tüm dünyada pek çok projede kullanıldığından, toolchain'in kendisinde bir problem varsa bunun tespit edilmesi ve düzeltilmesi kolay olacaktır. Ayrıca donanımın üreticisi tarafından uzun zaman test edilmiş ve müşterilere önerilen bir toolchain versiyonu varsa, elbette onun kullanılması da uygun olacaktır.

Peki ne zaman sıfırdan bir toolchain üretmek anlamlı olabilir?

Öncelikle söylemek gerekir ki, yukarıda özel toolchain üretiminin dezavantajları adına söylediklerimiz donanım üreticileri için de geçerlidir. Onlar da tüm bu ekosistem tarafından üretilen bilgi birikimini kullanmak ve kendi toolchain bazında kendilerine özgü bir problemi satın almamamak isterler.

Fakat bazen donanım üreticisinden gelen toolchain versiyonu çok eski olabilir. İçerisinden çıkan `glibc` standart C kitaplığının versiyonu çok düşük ise, alışageldiğiniz bazı fonksiyonları bulamayacağınızdan geliştirme yaparken bir miktar konfor kaybına uğrayabilirsiniz.

Örneğin **glibc** 2.5 versiyonu kullanılıyorsa, `asprinf()` fonksiyonu C kitaplığında yer almayacaktır. Veya **glibc** 2.12 öncesinde bir versiyon kullanılıyorsa, `getsubopt()` fonksiyonu yer almayacaktır. Bu sorunları çözmek adına versiyonu yükseltmek için gene hazır bir toolchain kullanabilirsiniz. Ancak böyle bir durumla karşılaştığınızda muhtemelen

kullandığınız donanım eskidir ya da üretici toolchain noktasında fazla destek sunmuyordur. Bu noktada donanımıza göre özelleştirilmiş bir toolchain üretimi, son çare olarak düşünülebilir.

Toolchain Sağlayıcıları

Günümüzde ARM platformu gömülü Linux sistemleri dünyasını domine ettiği için, daha çok bu platforma özgü opsiyonları incelediğimizde, aşağıdaki toolchain sağlayıcılarının öne çıktığını görmekteyiz:

- **Mentor Graphics / Sourcery CodeBench**: Destek ile birlikte satılan ücretli versiyonlarının yanı sıra, kayıt formunu doldurup ücretsiz olarak kullanılabilecek *Lite* versiyonları da bulunmaktadır. Uzun yıllardır bu alanda çalışan deneyimli bir firma olup gömülü Linux dünyasında yoğun bir kullanımı mevcuttur. **ARM**, **MIPS**, **PowerPC**, **SuperH** mimarileri için toolchain üretimi yapmaktadırlar.
- **Linaro**: Özellikle **ARM** platformu ve *hard-float* toolchain'ler üzerinde çalışmaktadırlar. Her geçen gün kullanımı artmaktadır.
- **Linux Dağıtımları**: Debian GNU/Linux, Ubuntu gibi dağıtımlarda **ARM** mimarisi için `gcc-arm-linux-gnueabi` ve `gcc-arm-linux-gnueabihf` paketlerini kurmak suretiyle paket geliştiricileri tarafından üretilmiş toolchain'ler de kullanılabilir

Kullanım

Toolchain arşivini sisteminizdeki bir dizine açtığınızda, `arm-none-linux-gnueabi-gcc` örneğindeki gibi derleyici araçlarına ait dosyalar oluşacaktır. Toolchain ön ekleri çoğunlukla buradakine benzer olmakla birlikte önceki bölümlerde değindiğimiz şekilde, değişkenlik de gösterebilir.

Eğer toolchain içerisinde kullanılan ön ek, `arm-none-linux-gnueabi-` şeklinde uzun olduğunda alışkanlık, kolaylık vb. nedenlerle, `arm-linux-` ön ekini kullanmak isterseniz, toolchain **bin** dizinine gidip, aşağıdaki gibi bir komutla sembolik linkler oluşturmak suretiyle, hem `arm-linux-gcc` şeklinde hem de `arm-none-linux-gnueabi-gcc` şeklinde derleyici çalıştırmanız mümkün olacaktır:

```
$ cd /path/to/toolchain/bin && \
   for i in `ls`; do s=`echo $i | cut -b 24-`; ln -s $i arm-linux-$s; done
```

Toolchain içerisinden çıkan derleyici çalıştırmak için ya bulunduğu yerin mutlak yolunu (*absolute path*) vermeli, ya da toolchain **bin** dizinini `PATH` ortam değişkeninize ekleyip, sadece derleyici adını vererek çalıştırmalısınız: değişkeninimize toolchain `bin` dizinini

ekleyebiliriz:

```
$ export PATH=$PATH:/path/to/toolchain/bin
$ arm-linux-gcc -v
```

> `PATH` ortam değişkenini bu şekilde her bir kabuk oturumu için ayarlayabilir veya kullandığınız kabuğun her girişte okuduğu temel konfigürasyon dosyalarından birine yazarak sonraki oturumlar için de kalıcı hale getirebilirsiniz.

Çalışılan Toolchain Hakkında Bilgi Edinme

Toolchain içerisinden çıkan gcc derleyicisine `-v` parametresini vermek suretiyle, toolchain'in oluşturulması sırasında yapılan tercihleri öğrenebiliriz:

```
$ arm-linux-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gcc
COLLECT_LTO_WRAPPER=/home/demirten/work/downloads/arm-2011.03/bin/../libexec/gcc/arm-n
one-linux-gnueabi/4.5.2/lto-wrapper
Target: arm-none-linux-gnueabi
Configured with: /scratch/janisjo/arm-linux-lite/src/gcc-4.5-2011.03/configure \
--build=i686-pc-linux-gnu --host=i686-pc-linux-gnu --target=arm-none-linux-gnueabi \
--enable-threads --disable-libmudflap --disable-libssp --disable-libstdcxx-pch \
--enable-extra-sgxxlite-multilibs --with-arch=armv5te --with-gnu-as --with-gnu-ld \
--with-specs='%{save-temps: -fverbose-asm} %{funwind-tables|fno-unwind-tables|mabi=*|f
freestanding|nostdlib:;:-funwind-tables} \
-D__CS_SOURCERYGXX_MAJ__=2011 -D__CS_SOURCERYGXX_MIN__=3 -D__CS_SOURCERYGXX_REV__=41 %
{O2:%{!fno-remove-local-statics: -fremove-local-statics}}\
 %{O*:%{O|O0|O1|O2|Os:;:%{!fno-remove-local-statics: -fremove-local-statics}}}' --enab
le-languages=c,c++ --enable-shared \
--enable-lto --enable-symvers=gnu --enable-__cxa_atexit --with-pkgversion='Sourcery G+
+ Lite 2011.03-41' \
--with-bugurl=https://support.codesourcery.com/GNUToolchain/ --disable-nls --prefix=/o
pt/codesourcery -\
-with-sysroot=/opt/codesourcery/arm-none-linux-gnueabi/libc --with-build-sysroot=/scra
tch/janisjo/arm-linux-lite/install/arm-none-linux-gnueabi/libc \
--with-gmp=/scratch/janisjo/arm-linux-lite/obj/host-libs-2011.03-41-arm-none-linux-gnu
eabi-i686-pc-linux-gnu/usr --with-mpfr=/scratch/janisjo/arm-linux-lite/obj/host-libs-2
011.03-41-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --with-mpc=/scratch/janisjo/arm
-linux-lite/obj/host-libs-2011.03-41-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr --wi
th-ppl=/scratch/janisjo/arm-linux-lite/obj/host-libs-2011.03-41-arm-none-linux-gnueabi
-i686-pc-linux-gnu/usr \
--with-host-libstdcxx='-static-libgcc -Wl,-Bstatic,-lstdc++,-Bdynamic -lm' \
--with-cloog=/scratch/janisjo/arm-linux-lite/obj/host-libs-2011.03-41-arm-none-linux-g
nueabi-i686-pc-linux-gnu/usr --with-libelf=/scratch/janisjo/arm-linux-lite/obj/host-li
bs-2011.03-41-arm-none-linux-gnueabi-i686-pc-linux-gnu/usr \
--disable-libgomp --enable-poison-system-directories --with-build-time-tools=/scratch/
janisjo/arm-linux-lite/install/arm-none-linux-gnueabi/bin --with-build-time-tools=/scr
atch/janisjo/arm-linux-lite/install/arm-none-linux-gnueabi/bin
Thread model: posix
gcc version 4.5.2 (Sourcery G++ Lite 2011.03-41)
```

gcc `--sysroot` parametresi

Çapraz derleme süreci boyunca sadece **glibc** standart C kitaplığına ihtiyaç duyan uygulamaların derlenmesi oldukça kolaydır. Ancak spesifik bir uygulama için derleme sürecinde ihtiyaç duyduğumuz diğer kütüphanelerin, çapraz derleme hedef sistemi için de önceden derlenmiş olması, başlık ve kütüphane dosyalarının çapraz derleme için kullanılan derleyicinin baktığı dizinlerden birinde yer alması gereklidir.

Örnek olarak, eğer **Sqlite C** arayüzünü kullanan bir uygulama geliştiriyorsanız, uygulamayı çapraz derlemeye başlamadan önce, sqlite kütüphanesini hedef platform için derlemeli, başlık dosyalarını ve kütüphanelerini belirli bir dizinde oluşturmalısınız. Bu adımdan sonra yazmış olduğunuz uygulamayı da hedef platform için derleyebilirsiniz.

Örnekten yola çıkarak şöyle bir soru sorabiliriz: **Sqlite** kütüphanesini bu şekilde derlemek zorunda kalmak yerine toolchain üreticisi aynı derleme işlemini yapıp, başlık ve kütüphaneleri toolchain içerisine dahil etse daha iyi olmaz mıydı?

Bu mümkündür hatta bazı toolchain'lerde gömülü sistemlerde sık kullanılan kütüphanelerin önceden derlenip hazır halde sunulduğunu görmekteyiz. Toolchain üretiminde verilen bu türden ek kararlar, oluşan toolchain'lerin boyutlarının birbirinden farklı olmasına yol açmaktadır.

Kullanılan toolchain içerisinde ihtiyaç duyulan tüm kütüphane ve başlık dosyalarının bulunması olanaksızdır. Dolayısıyla uygulamamızın ihtiyaç duyduğu diğer kütüphanelerin derlenmesi işlemi kaçınılmaz olarak ihtiyaç duyuldukça yapılması gereken bir süreçtir.

İhtiyaç duyulan tüm dosyaları belirli bir dizin yapısı altında birleştirebilmemiz halinde çapraz derleme işlemlerini çok daha kolay hale getirebiliriz. Örnek olarak `/opt/cross` şeklinde bir dizin açıp, başlık dosyalarını `/opt/cross/usr/include`, kütüphane dosyalarını ise `/opt/cross/usr/lib` dizini altında topladığımızı varsayalım. Yeni nesil **gcc** derleyicilerinde yer alan `--sysroot` parametresi aracılığıyla, derleyicimizin sadece `/opt/cross` dizini altında hazırladığımız başlık ve kütüphane dosyalarına bakmasını sağlayabiliriz. Bunun için derleyiciyi aşağıdaki şekilde çağırmamız yeterli olacaktır:

```
$ gcc --sysroot=/opt/cross test.c
```

Sysroot parametresiyle sağlanan bu kolaylık, **Buildroot** gibi hedef mimari için derleme ve dosya sistemi üretimi süreçlerini modellemeyi amaçlayan projelerde yoğun olarak kullanılmaktadır. Buildroot ile yeni bir projeye başlandığında, **staging** adında bir dizin oluşturulmakta ve öncelikle toolchain içerisinden çıkan başlık dosyaları `staging/usr/include` dizinine, kütüphane dosyaları ise `staging/usr/lib` dizinine kopyalanmaktadır. Sonrasında Buildroot içerisinde seçilmiş tüm kütüphaneler için derleme yapıldıkça ortaya çıkan yeni kütüphane ve başlık dosyaları da bu dizinler altında oluşturulmaya devam etmektedir.

Böylelikle host bilgisayarımızdaki `/usr/include` ve `/usr/lib` dizin yapısına benzer şekilde `staging` dizini altında bileşenler hazır hale getirilmektedir. Yapılan tüm çapraz derleme işlemlerinde de `--sysroot=staging/` **gcc** parametresi kullanıldığından derleme işlemleri daha kolay şekilde yapılabilmektedir.

pkg-config

`pkg-config` uygulaması, bir kütüphaneyi kullanmak istediğinizde vermeniz gereken derleme zamanı parametrelerini belirlemenize yardımcı olan bir araçtır. Örnek olarak **xt** kütüphanesiyle uygulamamızı linklemek istediğimizde gerekecek **LDFLAGS** ve **CFLAGS** parametrelerini pkg-config ile şu şekilde öğrenebiliriz:

```
$ pkg-config --libs xt
-lXt -lX11

$ pkg-config --cflags xt
-I/usr/include/X11
```

Ancak **pkg-config** uygulamasının bu bilgileri verebilmesi için, ilgili kütüphane sisteme kurulurken, pkg-config veritabanı dizinine gereken bilgileri içeren `.pc` uzantılı bilgi dosyasının da konulmuş olması gerekir. Her kütüphane bu şekilde bir .pc dosyası sağlamıyor olsa da büyük çoğunlukta .pc dosyaları sağlanmaktadır.

Bu sistemi destekleyen bir kütüphaneyi derlediğinizde, `make install` işlemi sonrası, `.pc` uzantılı tanım dosyası da sisteme kopyalanmaktadır. Bu dosyaların da başlık ve kütüphane dosyalarında olduğu gibi, hedef mimari için ortak bir dizin yapısı altında birleştirilmesinde fayda vardır. Normalde `.pc` dosyaları host bilgisayarda `/usr/lib/pkgconfig` dizini altına kurulur. Çapraz derleme işlemlerinde `./configure` aşamasında `--prefix=/usr` şeklinde verilmiş ise, `make install` aşamasında `DESTDIR` ortam değişkenini düzenleyerek **sysroot** için hazırladığımız dizin yapısı altında hem .pc dosyalarını hem de diğer başlık ve kütüphane dosyalarını olması gereken yerlere kopyalayabiliriz:

```
$ make DESTDIR=/opt/cross install
```

Bu komut sonrasında (configure aşamasında verilen prefix değerine bağlı olarak) başlık dosyaları `/opt/cross/usr/include`, kütüphane dosyaları `/opt/cross/usr/lib` ve .pc dosyaları `/opt/cross/usr/lib/pkgconfig` dizini altına kopyalanacaktır.

> `DESTDIR` ortam değişkenini kullanmaz isek kopyalama işlemi hedef platform yerine, host bilgisayardaki /usr/lib/pkgconfig dizinine kopyalanmaya çalışır ki bu istediğimiz bir durum değildir.

`pkg-config` uygulaması öntanımlı olarak `/usr/lib/pkgconfig` dizinine baktığı için çapraz derleme senaryolarında bu duruma müdahale etmez isek, host sisteminde yer alan kütüphaneler için yanıtlar üreyecektir. `PKG_CONFIG_PATH` ortam değişkenini kullanarak, uygulamanın veritabanı olarak farklı bir dizine bakmasını sağlayabiliriz:

```
$ PKG_CONFIG_PATH=/opt/cross/usr/lib/pkgconfig ./configure --host=arm-linux
```

Tipik Bir Çapraz Derleme Senaryosu

Autoconf Uygulamaları

GNU autoconf sistemiyle birlikte gelen kütüphane veya uygulamaların çapraz derlenmesi süreci oldukça kolaydır. Toolchain ön ekinin `--host` parametresiyle belirtilmesi yeterlidir. Toolchain'imizdeki derleyici `arm-linux-gcc` ise:

```
$ ./configure --host=arm-linux
```

şeklinde, `arm-none-linux-gnueabi-gcc` şeklinde ise:

```
$ ./configure --host=arm-none-linux-gnueabi
```

şeklinde **configure** sürecini tamamladığımızda, tüm **Makefile** dosyaları doğru derleyici araçlarını gösterecek biçimde üretilecektir.

> **Autoconf** destekli uygulamaların çapraz derleme işlemleriyle bir süre uğraştıktan sonra, `--host` parametresi ile hedef platformunu belirtiyor olmamız size biraz tuhaf veya yanlış gelebilir. Hatta `./configure -h` ile **configure** sistemi hakkında yardım aldığınızda, orada `--target` ve `--build` şeklinde parametreler olduğunu daha görürsünüz. Çapraz derleme süreci bir hedef (target) sistem için kod üretmeyi amaçladığına ve genel olarak da **host** kavramı işlemin yapıldığı geliştirme bilgisayarını gösterdiğine göre, `--target` parametresinin daha anlamlı olduğunu düşünmek doğaldır. Ancak GNU autoconf sisteminde **target** parametresi, sadece toolchain üretiminde anlamlı olan özel bir parametredir. Aşağıdaki örnekte **gcc** uygulamasının kaynak kodundan çapraz derlemede kullanılmak üzere **configure** edildiğini görüyoruz:

```
$ ./configure --build=i686-pc-linux-gnu --host=arm-linux --target=i686-pc-linux-gnu
```

> Bu örnekte
>
> - Kaynak koddan derleme sürecinde x86 Linux geliştirme bilgisayarındaki derleme araçlarının kullanılacağı [**--build**]
> - İşlem sonucunda oluşacak **gcc** uygulamasının ARM üzerinde çalışacağı [**--host**]
> - Derleme sonrası oluşacak **gcc** uygulamasının x86 mimarisi için kod üreteceği [**--target**]
>
> tanımlanmış oluyor. Derleme bittikten sonra oluşacak **gcc** uygulaması **ARM** üzerinde çalışacak, ancak ürettiği kod **x86** mimarisi için olacaktır (ARM üzerinden çapraz derlemeyle x86 kodu üretilecektir)

Derleme işlemi yapacağımız kütüphane ve uygulamaların konfigürasyon adımlarında farklılıklar görülebilir. İlgili yazılımın ihtiyaç duyduğu diğer kütüphaneler ve bu kütüphanelerin çapraz derlenmiş versiyonlarının nerede bulunabileceğini gösterebileceğimiz onlarca parametre bulunmaktadır.

```
$ ./configure --help
```

ile bunları görebiliriz.

Bu noktada yardımımıza `pkg-config` uygulaması koşar. `PKG_CONFIG_PATH` ortam değişkeni düzgün şekilde ayarlanmışsa, ek bir işleme gerek kalmaksızın ihtiyaç duyulan diğer kütüphanelerin çapraz derlenmiş versiyonlarına ait bilgiler **configure** uygulaması tarafından bulunabilir.

Ancak bunun mümkün olmadığı senaryolarda (pkg-config sisteminde tanımı olmayan kütüphaneler, standart dışı dizinlerde kurulu kütüphaneler vb.), ortam değişkenlerini kullanarak bu dizinleri gösterebiliriz:

```
$ CFLAGS="-I/opt/other_build/xxx/include" \
LDFLAGS="-L/opt/other_build/xxx/lib -lxxx" \
./configure --host=arm-linux
```

Başarılı bir **configure** sonrasında gerekli tüm `Makefile` dosyaları üretilmiş olduğundan sonrasında:

```
$ make
```

komutu ile derleme işlemleri yapılır.

Uygulama derlendikten sonra, `DESTDIR` yöntemiyle çapraz derlenmiş yazılımları topladığımız mini root filesystem altına kurulum yapabiliriz:

```
$ make DESTDIR=/opt/custom_build install
```

Bazı eski programlar `DESTDIR` özel değişkenini dikkate almayabilirler, bu durumda el yordamı ile ilgili dosyaların mini root filesystem altında yüklenmesi gerekecektir.

Autoconf Dışındaki Uygulamaların Derlenmesi

Bazı eski uygulamalarda, autoconf içermeyen küçük uygulamalarda veya Linux kernel kodu, busybox gibi bazı özel derleme süreçleri olan uygulamalarda çapraz derleme işleminin nasıl olacağı, ilgili sistemle birlikte gelen **Makefile** dosyaları incelenerek anlaşılabilir.

Linux kernel kodu, busybox gibi uygulamaların Makefile dosyalarında aşağıdaki gibi bir tanım bulunur:

```
CC = $(CROSS_COMPILE)gcc
```

Bu şekildeki bir Makefile dosyasında `CROSS_COMPILE` değişkeni verilmediği müddetçe standart `gcc` derleyicisi çalışacaktır. Ancak `CROSS_COMPILE` değişkenine `arm-linux-` şeklinde bir değer atandığında yukarıdaki satır doğrultusunda Makefile içerisinde birleştirme yapıldığında C derleyicisini gösteren `CC` değişkeninin değeri çapraz derleme için istediğimiz biçimde `arm-linux-gcc` halini alacaktır. Bu tarz uygulamaları aşağıdaki gibi derleyebiliriz:

```
$ make CROSS_COMPILE=arm-linux-
```

> **Not**: Autoconf sistemindeki **host** parametremizden farklı olarak sondaki **-** karakterine dikkat ediniz. Bu gereklilik tamamen **Makefile** dosyalarındaki kuralların tanımıyla ilgilidir.

Autoconf desteği dışındaki bu tarz uygulamalarda çoğunlukla **CROSS_COMPILE** ve **CROSS_COMPILE_PREFIX** değişkenleri kullanılır, bu değişkenlere doğru toolchain ön ekleri atandığında çapraz derleme işlemi gerçekleştirilir.

Nadiren bazı uygulamaların Makefile dosyalarında

```
CC = gcc
```

doğrudan tanımlar da bulunabilmektedir. Bu şekildeki bir uygulamayı çapraz derlenebilir hale getirmek için Makefile dosyalarında düzenleme yapmaktan başka yöntem bulunmamaktadır.

NOR, NAND, eMMC ve Flash Tabanlı Depolama

Flash tabanlı depolama ortamları uzun yıllardır kullanılıyor. Mekanik disklere oranla hareket eden parçaların olmaması hem gürültü oranını düşürmekte hem de daha verimli bir çalışma imkanı sağlamaktadır. Ayrıca hareket halindeki mobil sistemler için çok daha dayanıklı bir kullanım sağlamaktadır. Hareket eden mekanik bileşenler olmadığı için rastgele erişim performansı da yüksek olmaktadır. Tüm bu saydığımız avantajlara ek olarak, mekanik bir diske oranla güç tüketimi çok daha düşük olmaktadır.

Elbette yukarıda sayılan tüm avantajlarına rağmen, flash tabanlı depolama aygıtlarının önemli handikapları da bulunmaktadır.

Flash tabanlı bir aygıt üzerinde, aynı bloğa herhangi bir şey yazmak için, tüm blok içeriğinin silinmesi gerekmektedir. Ancak her bir blok için yazma ömrü sınırlıdır. Ürünlere göre değişkenlik göstermekle birlikte genelde üreticiler blok başına 100.000 - 1.000.000 arasında değişen silme işlemi adedine kadar garanti verebilmektedir.

Blok başına silme ömrü kısıtı nedeniyle, flash tabanlı depolama aygıtı kullanılırken, aynı blok üzerinde işlem yapmak yerine donanım veya yazılım katmanında bu işlemlerin yönetilmesi ve sistemdeki tüm blokların eşit sayıda kullanılması sağlanmaya çalışılır. Depolama aygıtının ömrünü uzatmaya yönelik bu yönteme **wear leveling** adı verilmektedir.

NOR Flash

NOR Flash tipi depolama aygıtına işlemci tarafından doğrudan rastgele erişim yapılabilmekte ve okunan kodlar doğrudan çalıştırılabilmektedir (XIP - eXecute-In-Place). Geleneksel yöntemlerin aksine, NOR Flash'tan okunan bir makine kodunun çalıştırılmadan önce RAM'e kopyalanmasına gerek yoktur.

Bu özellik NOR flash tipini özellikle *bootloader*, *BIOS* gibi uygulamalar için daha iyi bir seçenek haline getirmektedir.

Okuma performansı iyi olmasına rağmen, yazma performansı NAND flash tipine göre düşüktür. Bu problem NOR flash'ları genel bir depolama birimi olarak kullanma noktasında dezavantajlı konuma getirmektedir. Ayrıca zaman içerisinde NAND flash kapasitelerinin artması ve fiyatlarının düşmesi, NOR flash kullanımını önemli oranda düşürmüş ve kullanımı belirli alanlarla sınırlanmıştır.

NAND Flash

NAND Flash depolama aygıtı son yıllarda geniş bir kullanım alanı bulmuş, birim maliyeti düşmüş ve kapasitesi artmıştır. Ancak NOR Flash tipinden farklı olarak, CPU tarafından doğrudan erişilip kod çalıştırılamaz. NAND controller üzerinden erişilip içerisinde yer alan kodlar RAM'e aktarıldıktan sonra işlemci tarafından işletilebilir.

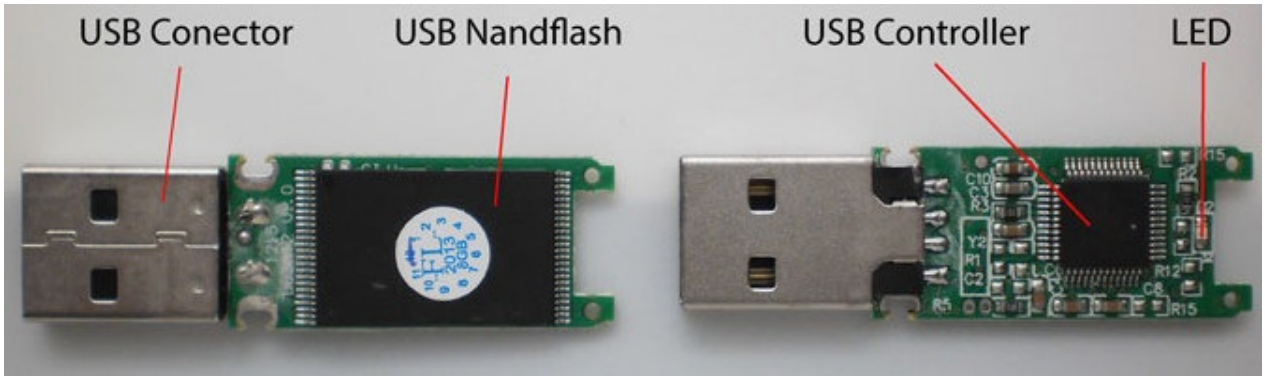
NAND Flash'ın bir diğer problemi, üretim sürecinden belirli bir oranda bozuk blokla (badblocks) çıkıyor olmasıdır. Üreticiler, **%1**'e kadar olan bozuk blok oranını kabul edilebilir bulmakta ve yaptıkları testlerde bulunan bozuk blokları, ilgili bloğun **OOB** (Out-Of-Band) alanına **MBBM** (Manufacturer's Bad Block Marker) özel işareti ile yazmaktadır.

Fabrika çıkışı işaretlenmiş bozuk blokların yanı sıra, kullanım sırasında da yeni bozuk bloklar oluşacaktır. Bunların algılanması ve benzer şekilde bir daha kullanılmaması için işaretlenmesi gereklidir.

Yukarıdaki maddeler ışığında, NAND Flash tabanlı bir depolama aygıtını verimli kullanabilmek için, donanım üzerinde veya kernel seviyesindeki yazılımla NAND Flash üzerinde yapılacak işlemlerin yönetilmesi gerekmektedir.

Günümüzde 2 tür NAND Flash depolama aygıtı bulunmaktadır. Birinci tür, içerdiği **F**lash **T**ranslation **L**ayer sayesinde üst katmanda standart bir blok aygıtı gibi görünürken, ikinci tür **raw flash** aygıtları olarak adlandırılır ve işletim sistemi çekirdeği tarafından Controller'a erişilebilir ve bozuk blok yönetimi işletim sisteminin kontrolünde gerçekleştirilebilir.

Flash Translation Layer



İlk zamanlarda çıkan PCMCIA flash kartları, **raw flash** türünde idi. Bu nedenle üzerinde blok tabanlı bir dosya sistemi çalıştırabilmek için (fat, ext3 vb.) yazılımsal bir FTL sürüsüne ihtiyaç vardı.

Sonrasında bu işlemler donanım üzerindeki bir *controller* aracılığıyla *firmware* içerisinde yapılmaya başlandı.

Günümüzde USB Flash Bellek, SSD, eMMC gibi sistemler, firmware tarafından sunulan *FTL* katmanı sayesinde ek bir sürücüye ihtiyaç duyulmaksızın, normal bir disk gibi kullanılabilmektedir. Donanımın firmware'i işlemleri bizim için yaptığından dolayı üst katmanda herhangi blok tabanlı dosya sistemi kullanılabilmektedir.

SSD ve büyük oranda eMMC'yi kenarda tutarak, diğer FTL tabanlı depolama ortamlarını gömülü Linux projelerinde, endüstriyel işlerde kullanmak iyi bir fikir olmayabilir. Bunu biraz daha açacak olursak:

- Üreticiler FTL işlevini gerçekleştiren firmware kodunu kapalı tuttuklarından, flash tabanlı bir ortam kullanmaktan gelen kısıtları (silme ömrü, wear-levelling vb.) bu firmware içerisinde ne derece *iyi* bir şekilde halletmiş olduklarını bilme şansımız kalmaz.
- FTL tabanlı ortamların çoğu, üzerinde *FAT* gibi basit bir dosya sisteminin çalıştırıldığı, güç kesintilerinde veri kaybına tahammülü olan, son kullanıcıya yönelik ürünler olarak tasarlanmaktadır. Endüstriyel bir kullanıma yönelik tasarlanmamışlardır.
- FTL controller, dosya sisteminde ne olup bittiğini tam olarak bilemeyeceğinden, örnek olarak silinmiş dosyalara ait kullanılmayan bloklar için de işlem yapmaya devam ederken, **raw flash** üzerinde çalışan dosya sistemleri, daha iyi bir yönetim yapabilir.

Bu olumsuzluklara rağmen, kaliteli bir firmware ve hızlı bir controller ile, yüksek başarım sağlanabilir. SSD ve eMMC aygıtlarının hedef kitlesi nedeniyle üretim kaliteleri ortalama olarak çok daha iyi olduğundan daha iyi performans elde edilebilmektedir. Bununla birlikte controller tarafında kapalı kod ile yapılan işlemler, sürecin toplamdaki kalitesi noktasında bir şüphe noktası oluşturmaktadır.

Raw NAND Flash

Raw NAND Flash aygıtları için Linux kernel seviyesinde tam bir kontrol sağlanabilmektedir.

Farklı çip üreticileri bulunduğundan kullanılan raw-flash çipi donanımından bağımsız olarak genel bir kontrol arayüzü ve katmanı sağlayabilmek adına, Linux kernel içerisinde **Memory Technology Device** katmanı geliştirilmiştir. Bu sayede raw-flash çip üreticileri alt katmanda donanımlarıyla MTD katmanının bağlantısını sağladığında, ilgili çip için kernel yazılımsal controller vazifesi görebilmektedir.

eMMC ve Gelecekteki Durum

eMMC aygıtları, NAND Flash depolama aygıtının board üzerinde gömülü (embedded) olup MMC arayüzü ile kernel içerisinde standart bir blok aygıtı gibi çalışmaktadır.

Son 2 yılda giderek artan hızda kullanım alanı bulmuş olup, kapasite, verimlilik, birim maliyet gibi alanlarda raw NAND flash aygıtlarına göre avantajlı hale gelmiş, üretim hacmi artmış ve sürekli fiyatları düşmektedir. Buna bağlı olarak çok yakın bir gelecekte raw NAND flash tabanlı aygıtları piyasada görememeye veya çok yüksek fiyatlardan bulmaya başlarsak şaşılmaması gerekir. **2014** sonu itibariyle piyasaki bir çok board, depolama birimi olarak raw NAND flash yerine, eMMC veya MMC ile gelmektedir.

Memory Technology Device - MTD Katmanı

MTD katmanı, donanım spesifik **raw flash** aygıt sürücüleri ile bunları kullanacak üst seviyeli katmanlar arasında bir soyutlama yapılabilmesini sağlar.

raw flash tabanlı ortamlarda kullanılır.

MTD katmanı tarafından algılanan aygıtlar, `/proc/mtd` dosyasından görüntülenebilir:

```
$ cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00020000 00020000 "u-boot env"
mtd1: 00020000 00020000 "UBL"
mtd2: 00080000 00020000 "u-boot"
mtd3: 00400000 00020000 "kernel"
mtd4: 08000000 00020000 "filesystem1"
mtd5: 07a00000 00020000 "filesystem2"
```

MTD Karakter Aygıtı Desteği

mtdchar sürücüsü sistemde algılanan her bir MTD bölümüne (*partition*) karşılık, karakter tabanlı erişimi simüle edecek bir karakter aygıtı oluşturur.

Major numarası **90** (*5A*) şeklindedir. İsimlendirmesi `/dev/mtdXX` şeklinde yapılır.

Read-Only erişimler için tek sayılı *minor* numaraları, read-write erişim için çift sayılı *minor* numaraları kullanılır.

raw flash üzerinde yapılacak işlemler ve silme operasyonları için ek **ioctl** çağrıları içerir.

Genellikle *mtd-utils* paketinden çıkan uygulamalar tarafından kullanılır.

MTD Blok Aygıtı Desteği

mtdblock sürücüsü sistemde algılanan her bir MTD bölümüne (*partition*) karşılık, blok tabanlı erişimi simüle edecek bir blok aygıtı oluşturur.

Major numarası **31** (*1F*) şeklindedir. İsimlendirmesi `/dev/mtdblockXX` şeklinde yapılır.

Minor numarası, MTD bölümünün numarasını gösterir.

Blok tabanlı okuma-yazma operasyonlarını destekler. Ancak bozuk blok yönetimi, yazma operasyonlarının farklı bloklara dağıtılması (*wear-levelling*) gibi destekleri bulunmamaktadır.

MTD Partitioning (Bölümlendirme)

MTD aygıtları, bir gereklilik olmamakla birlikte, çoğu durumda bölümlendirme yapılarak kullanılırlar.

Bu şekilde MTD aygıtının farklı alanlarını, farklı özelliklerde (read-write, read-only vb.) ve amaçlarda kullanmak kolaylaşır.

Blok tabanlı aygıtlarda mevcut olan bölümlendirme tablosu (*partition table*) yapısı, MTD aygıtlarında yer almaz.

Bu nedenle bölümlendirme tablosunun nasıl olduğunun harici bir yerde saklanması gerekir. Bu harici yer, *kernel* kodunun içerisinde hard-coded biçiminde veya *kernel* açılış parametreleri sayesinde her açılışta parametrelerin okunması suretiyle gerçeklenir.

Redboot gibi bazı boot yükleyici uygulamaları, sabit disk gibi aygıtlara benzer şekilde bölümlendirme tablosunu MTD aygıtının belirli bir yerinde tutma alternatif yöntemiyle de çalışabilmektedirler. Bu senaryoda *kernel* tarafından ilgili bölümlendirme kurallarının okunup anlaşılabilmesi için, kernel derleme sürecinde *Redboot Partition Table Parsing* seçeneğinin aktifleştirilmesi gereklidir.

MTD Partitioning

Hardcode Yöntemi

Doğrudan ilgili MTD sürücüsü içerisinde yapılır. Örnek olara `arch/arm/mach-omap2/board-omap3beagle.c` içerisine bakalım:

```
static struct mtd_partition omap3beagle_nand_partitions[] = {
    /* All the partition sizes are listed in terms of NAND block size */
    {
        .name       = "X-Loader",
        .offset     = 0,
        .size       = 4 * NAND_BLOCK_SIZE,
        .mask_flags = MTD_WRITEABLE,    /* force read-only */
    },
    {
        .name       = "U-Boot",
        .offset     = MTDPART_OFS_APPEND,   /* Offset = 0x80000 */
        .size       = 15 * NAND_BLOCK_SIZE,
        .mask_flags = MTD_WRITEABLE,    /* force read-only */
    },
    {
        .name       = "U-Boot Env",
        .offset     = MTDPART_OFS_APPEND,   /* Offset = 0x260000 */
        .size       = 1 * NAND_BLOCK_SIZE,
    },
    {
        .name       = "Kernel",
        .offset     = MTDPART_OFS_APPEND,   /* Offset = 0x280000 */
        .size       = 32 * NAND_BLOCK_SIZE,
    },
    {
        .name       = "File System",
        .offset     = MTDPART_OFS_APPEND,   /* Offset = 0x680000 */
        .size       = MTDPART_SIZ_FULL,
    },
};
```

Bu modelde çalışan çekirdek yeniden derlenmeden, bölümlendirme tablosunu değiştirmek mümkün olmaz. Kulağa hoş gelmiyor olsa da pek çok sistem bu şekilde çalışmaktadır. Gömülü sistemlerin dizaynı yapıldıktan sonra sahadaki kullanımında sonradan bölümlendirme tablosunun yeniden yapılandırılması ihtiyacı genellikle oluşmaz. Böyle bir ihtiyacın oluşması durumu, muhtemel bir tasarım süreci problemine işaret eder.

Kernel Parametresiyle Bölümlendirme

Kernel açılış parametresi olarak aşağıdaki formatta değer girilmek suretiyle bölümlendirme tablosunu tanımlamak mümkündür:

```
mtdparts=davinci_nand:4m(kernel)ro,32m(rootfs)ro,100m(data),-(archive)
```

Bu örnekte;

- 4 MB Kernel için read-only,

- 32 MB kök dosya sistemi için `rootfs` adında read-only,
- 100 MB `data` adında genel bir yazılabilir bölüm
- Ve flash'ın geri kalan kısmının tamamından oluşan `archive` adında bir bölüm daha oluşturulmuştur.

Kernel'ın ilgili flash çipini hangi anahtar kelime üzerinden tanıdığını önceden öğrenmek veya açılış loglarından takip etmek gerekir. Bu örnekte anahtar kelime `davinci_nand` şeklindedir. Sistemde birden fazla **NAND** flash çipi de bulunabilir. Bu şekilde hangi sürücü üzerinden hangi çipe erişmek istediğimizi de belirtmiş oluyoruz. Aynı sürücüde birden fazla çip olduğunda, *instance* numarası da kullanılır.

MTD Bölümünün Silinmesi

Bir MTD bölümünün tamamının veya belirli sayıdaki blokunun silinmesi için **mtd-utils** paketinden çıkan `flash_erase` uygulaması kullanılır:

```
flash_erase <mtd_device> <offset> <block_count>
```

Offset değeri olarak `0` ve *block count* değeri olarak `0` girilmesi durumunda, tüm bölüm silinir.

Bloklar daha önceden kilitlenmiş durumda ise, `-u` parametresi ile veya `flash_unlock` uygulaması ile kilit kaldırılır.

Örnek kullanım:

```
flash_erase -u /dev/mtd3 0 0
```

MTD Bölümüne Yazma

Bir MTD bölümüne yazma (dosyasistemi imajını) işlemi için *mtd-utils* paketinden çıkan `flashcp` ve `nandwrite` uygulamaları kullanılır.

```
nandwrite -p <mtd_device> <file_name>
flashcp <file_name> <mtd_device>
```

nandwrite uygulaması **NAND** flash tipinde, *flashcp* uygulaması **NOR** flash tipinde kullanılır.

nandwrite uygulaması biraz daha yetenekli oluğ, *padding* vb. gibi daha fazla seçenek sunmaktadır.

Örnek kullanım:

```
nandwrite -p /dev/mtd3 /tmp/jffs.image
flashcp /tmp/jffs.image /dev/mtd3
```

MTD Aygıt Simülasyonu

Geliştirme ortamınızdaki bilgisayarda NAND flash bulunmayacağı için sürekli bir cihaz üzerinde çalışmanın mümkün olmadığı senaryolarda, `nandsim` modülü üzerinden NAND flash ortamını simüle etmeniz mümkündür.

NAND simulator (nandsim) NAND flash aygıtını RAM veya bir dosya üzerinde simüle eder. `nandsim` modülünü yüklerken simülasyonun durumunu etkileyecek pek çok modül parametresi bulunmaktadır. Başlıca modül parametreleri aşağıdaki gibidir:

- `first_id_byte` : NAND flash üzerinden okunacak ilk byte'ın ne olacağını gösterir (`read ID` komutuna dönen cevap, `manufacturer ID` değeri olarak yorumlanacaktır). Kernel kaynak kodu içerisindeki `include/linux/mtd/nand.h` dosyasında NAND üretici kodları tanımlıdır.
- `second_id_byte` : NAND flash üzerinden okunacak ikinci byte'ın ne olacağını gösterir (`read ID` komutuna dönen cevap, `chip ID` değeri olarak yorumlanacaktır). `drivers/mtd/nand/nand_ids.c` dosyasında **chip ID** değerleri yer almaktadır. Örnek olarak `0x78` chip ID değeri, 128MiB 1,8V 8-bit NAND iken, `0xC5` 2GiB 3,3V 16-bit NAND çipini gösterir.

Bunların haricinde `nandsim` modülünün **20** kadar daha parametresi bulunmaktadır. Parametrelerin listesini `modinfo` komutuyla alabilirsiniz:

```
$ sudo modinfo nandsim
parm:     first_id_byte:The first byte returned by NAND Flash 'read ID' command (manuf
acturer ID) (uint)
parm:     second_id_byte:The second byte returned by NAND Flash 'read ID' command (chi
p ID) (uint)
parm:     third_id_byte:The third byte returned by NAND Flash 'read ID' command (uint)
parm:     fourth_id_byte:The fourth byte returned by NAND Flash 'read ID' command (uin
t)
parm:     access_delay:Initial page access delay (microseconds) (uint)
parm:     programm_delay:Page programm delay (microseconds (uint)
parm:     erase_delay:Sector erase delay (milliseconds) (uint)
parm:     output_cycle:Word output (from flash) time (nanoseconds) (uint)
parm:     input_cycle:Word input (to flash) time (nanoseconds) (uint)
parm:     bus_width:Chip's bus width (8- or 16-bit) (uint)
parm:     do_delays:Simulate NAND delays using busy-waits if not zero (uint)
parm:     log:Perform logging if not zero (uint)
parm:     dbg:Output debug information if not zero (uint)
parm:     parts:Partition sizes (in erase blocks) separated by commas (array of ulong)
parm:     badblocks:Erase blocks that are initially marked bad, separated by commas (c
harp)
parm:     weakblocks:Weak erase blocks [: remaining erase cycles (defaults to 3)] sepa
rated by commas e.g. 113:2 means eb 113 can be erased only twice before failing (charp
)
parm:     weakpages:Weak pages [: maximum writes (defaults to 3)] separated by commas
e.g. 1401:2 means page 1401 can be written only twice before failing (charp)
parm:     bitflips:Maximum number of random bit flips per page (zero by default) (uint
)
parm:     gravepages:Pages that lose data [: maximum reads (defaults to 3)] separated
by commas e.g. 1401:2 means page 1401 can be read only twice before failing (charp)
parm:     overridesize:Specifies the NAND Flash size overriding the ID bytes. The size
 is specified in erase blocks and as the exponent of a power of two e.g. 5 means a siz
e of 32 erase blocks (uint)
parm:     cache_file:File to use to cache nand pages instead of memory (charp)
parm:     bbt:0 OOB, 1 BBT with marker in OOB, 2 BBT with marker in data area (uint)
parm:     bch:Enable BCH ecc and set how many bits should be correctable in 512-byte b
locks (uint)
```

Şimdi Fujitsu üreticisi için (0x04), 128 MiB'lık NAND flash (0x78) çipini simüle edelim:

```
$ sudo modprobe nandsim first_id_byte=0x04 second_id_byte=0x78
```

Sonrasında `dmesg` komutuyla kernel buffer alanının son kısmına bakalım:

```
$ dmesg
[14399.796192] nand: device found, Manufacturer ID: 0x04, Chip ID: 0x78
[14399.796192] nand: Fujitsu NAND 128MiB 1,8V 8-bit
[14399.796193] nand: 128MiB, SLC, page size: 512, OOB size: 16
[14399.796203] flash size: 128 MiB
[14399.796203] page size: 512 bytes
[14399.796204] OOB area size: 16 bytes
[14399.796205] sector size: 16 KiB
[14399.796205] pages number: 262144
[14399.796206] pages per sector: 32
[14399.796206] bus width: 8
[14399.796207] bits in sector size: 14
[14399.796207] bits in page size: 9
[14399.796208] bits in OOB size: 4
[14399.796209] flash size with OOB: 135168 KiB
[14399.796209] page address bytes: 4
[14399.796210] sector address bytes: 3
[14399.796210] options: 0x42
[14399.796646] Scanning device for bad blocks
[14399.803631] Creating 1 MTD partitions on "NAND 128MiB 1,8V 8-bit":
[14399.803635] 0x000000000000-0x000008000000 : "NAND simulator partition 0"

$ cat /proc/mtd
dev:    size   erasesize  name
mtd0: 08000000 00004000 "NAND simulator partition 0"
```

Artık kernel açısından normal bir MTD aygıtımız oluşmuş oldu. MTD aygıtları üzerindeki tüm işlemleri burada yapabiliriz (Jffs2, Ubi, Ubifs çalışmaları, `flash_erase` , `nandwrite` vb.)

Unsorted Block Images - UBI Katmanı

UBI üzerinde işlem yapabilmek için, kontrol arayüzü olan `/dev/ubi_ctrl` dosyası sistemde yer almalıdır.

Bu özel dosyanın sabit bir major:minor numarası bulunmaz ve dinamik major:minor allokasyonu kullandığından, öntanımlı major:minor değerlerin sistemde halihazırda kullanımda olması durumuna karşı, farklı değerler de oluşabilir.

Sistemde **udev** veya **devtmpfs** gibi, */dev* dizini altındaki aygıt dosyalarını otomatik oluşturan bir araç kullanmıyorsanız, `/dev/ubi_ctrl` dosyasını elle oluşturmanız gerekecektir.

Doğru *major:minor* numaralarını, **/sys/class/misc/ubi_ctrl/dev** dosyasından öğrendikten sonra (Örnek: 10 ve 63) aşağıdaki gibi oluşturabilirsiniz:

```
mknod /dev/ubi_ctrl c 10 63
```

UBI Attach İşlemi

UBIFS dosya sistemini kullanabilmek için sistemde öncelikle *UBI* volume'lerin oluşturulması gereklidir.

UBI desteği kernel içerisinde mevcut ise, kernel açılış parametresiyle UBI attachment aşağıdaki şekilde ayarlanabilir:

```
ubi.mtd=rootfs
veya
ubi.mtd=3
```

Yukarıdaki örnekte, daha önceden bölümlendirilmiş ve bölüm etiketi olarak "rootfs" verilmiş olan /dev/mtd3 bölümü, bir UBI aygıtı olarak sisteme eklenmektedir.

Flash üzerindeki bölümlendirme etiketleri kullanılabileceği gibi, doğrudan MTD bölüm numarası da kullanılabilmektedir.

UBI desteği modül olarak derlenmiş ise, ubi modülüne yüklenirken yukarıdaki gibi parametreler geçilmelidir.

UBI Attach ve *UBI Detach* işlemleri, *mtd-utils* paketinden çıkan **ubiattach** ve **ubidetach** uygulamaları kullanılarak da yapılabilir.

/dev/mtd3 bölümünü *attach* etmek için:

```
ubiattach /dev/ubi_ctrl -m 3
```

/dev/mtd3 için mevcut attachment'ı sonlandırmak için:

```
ubidetach /dev/ubi_ctrl -m 3
```

> Not: UBI komutlarında kontrol aygıtı ismi verilmediğinde öntanımlı olarak **/dev/ubi_ctrl** varsayılır.

Soru: Root dosya sistemi olarak UBIFS kullanıldığında attachment işlemi nasıl yapılabilir?

UBI Attach işlemi tamamlandığında **udev** veya **devtmpfs** gibi */dev* dizini altındaki aygıt dosyalarını yöneten bir mekanizma kullanıyorsanız otomatik olarak sırasıyla `/dev/ubi0`, `/dev/ubi1` şeklinde attach edilen aygıtlara ilişkin dosyalar oluşacaktır.

Bu şekilde bir sistem kullanmıyorsanız bu dosyaları `mknod` uygulamasıyla elle oluşturmanız gerekecektir. Bu şekilde elle oluşturma durumu sözkonusu ise, gereken *major:minor* numaralarını `/sys/class/ubi/ubi0/dev` dosyasından öğrenebilirsiniz (Örnek: 253 ve 0) Sonrasında aşağıdaki komuta benzer şekilde ilgili aygıt dosyasını oluşturabilirsiniz:

```
mknod /dev/ubi0 c 253 0
```

Volume Yönetimi

UBI sisteminde attach edilmiş olan MTD aygıtları üzerinde, UBI katmanında mantıksal volume birimleri oluşturulmalıdır.

Örnek olarak 600 MB boyutundaki `/dev/mtd3` **ubiattach** ile sisteme `/dev/ubi0` şeklinde eklenmiş olsun. Bu UBI aygıtı üzerinde çok sayıda UBI volume birimi oluşturmamız mümkündür. Bu işlem için `ubimkvol` uygulaması kullanılacaktır.

Örnek olarak bu alan üzerinde 100 MB, 200 MB ve 300 MB boyutlarında 3 adet volume birimi oluşturalım:

```
ubimkvol /dev/ubi0 -N test0 -s 100MiB
ubimkvol /dev/ubi0 -N test1 -s 200MiB
ubimkvol /dev/ubi0 -N test2 -m
```

- `-N` parametresi ile birimlere etiket atanmaktadır
- Üçüncü komutta `-m` parametresi ile kalan maksimum alan ayrılmıştır.

Mevcut bir UBI volume birimini silmek içinse `ubirmvol` uygulaması kullanılır.

Etiketi `test2` olan volume birimini kaldırmak için:

```
ubirmvol /dev/ubi0 -N test2
```

Etiket yerine volume birim numaraları da kullanılabilir:

```
ubirmvol /dev/ubi0 -n 2
```

Gömülü Sistemlerde Kullanılan Dosya Sistemleri

Gömülü sistemlerde genellikle flash tabanlı depolama alanı kullanılmaktadır.

SSD, MMC, SD kartlar, USB flash diskler, Memory Stick, Compact Flash kartlar ve benzeri *FTL* (Flash Translation Layer) kullanan aygıtlar, *flash tabanlı* olarak nitelendirilmezler.

Flash Translation Layer, ilgili aygıtın normal bir disk gibi görünmesini ve üzerinde blok tabanlı işlemler yapılabilmesini sağlar.

Flash tabanlı sistemlere örnek olarak NAND ve NOR flash chip'lerini verebiliriz.

Flash tabanlı sistemler Linux altındaki **MTD** (Memory Technology Device) alt sistemi ile yönetilirler.

Aygıt (Device) Tipleri

Linux altında 2 aygıt tipi bulunur:

Karakter Tabanlı Aygıtlar	Blok Tabanlı Aygıtlar
Sabit bir boyutu yoktur	Sabit bir boyut bulunur
Rastgele erişim modu tanımlı değildir	Rastgele erişim yapılabilir
Veri erişimlerinde tampon kullanılmaz	Erişimlerde tampon kullanılır

Raw Flash tabanlı aygıtlar ise ne bir karakter aygıt gibi, ne de tam olarak blok aygıt gibi davranırlar.

MTD ve Blok Aygıtlar

MTD Aygıtlar	Blok Tabanlı Aygıtlar
eraseblock'lardan oluşur	**sektör**'lerden oluşur
eraseblock'lar genellikle 128KB gibi büyük değerlerde olur	sektörler 512, 1024 byte gibi küçük değerlerdedir
3 temel operasyonu destekler: eraseblock'tan okuma, eraseblock'a yazma ve eraseblok'u silme	2 temel operasyonu destekler: sektörden okuma ve sektöre yazma
Bozuk eraseblock'lar yazılım tarafında kontrol edilmelidir	Bozuk sektörler donanım tarafından kontrol edilir ve yazılımda bir şey yapmak gerekmez
eraseblock'lar 10000 silme işleminden sonra genellikle bozulmaya başlar	sektörler için benzeri bir bozulma söz konusu değildir

JFFS2 Dosya Sistemi

Journalling Flash FileSystem'in 2. versiyonudur. *David Woodhouse* tarafından geliştirilmiş olup **2.4.10** kernel versiyonundan itibaren Linux kaynak kodu ile birlikte dağıtılmaya başlanmıştır.

Blok tabanlı aygıtlar üzerinde değil, **raw flash** aygıtlar üzerinde çalışır. *Journal* desteğiyle çalıştığı için, ani güç kesintilerinden sonra dosya sistemini *kararlı* bir yapıya döndürebilmektedir. Kendisinden önceki ilk versiyona göre aşağıdaki ek özelliklere sahiptir:

- NAND flash desteğimo
- Hard link oluşturabilme desteği
- Dosya sistemini sıkıştırılmış formda kullanma desteği
- Performansı artıran yeni **garbage collection** mekanizması

Dezavantajları

Aşağıda sıralanan temel sebeplerden ötürü JFFS2 kullanımı günümüzde oldukça azalmıştır:

- Tüm *node*'ların mount işlemi sırasında taranması ihtiyacı gerektiğinden, flash kapasitelerinin GigaByte seviyelerine ulaşması nedeniyle çok uzun mount süreleri gerektirmesi
- Çok sayıda küçük dosyanın yazılması durumunda sıkıştırma desteğinin negatif bir etki üretmesi
- Bellekte her node için bir veri yapısı tuttuğundan kullanılan alan arttıkça bellek kullanımının artması ve performansın düşmesi
- Tamamen senkron çalışan bir dosya sistemi olduğundan yazma işlemleri doğrudan flash ortama yapılır. Çok sayıda dosya üzerinde küçük değişiklikler yapıldıkça sıkıştırma işlemleri de tekrar yapılır, zamanla fragmentation artar, performans düşer.

Yukarıda sayılan nedenler yüzünden yerini büyük oranda **UBIFS**'e bırakmasına karşılık, dosya sistemi için ayrılmış alanın çok küçük olduğu senaryolarda (örneğin konfigürasyon işlemleri için **1 MB**'lık bir bölüm ayırmış olabilirsiniz), **UBIFS** overhead'i fazla olduğundan birinci alternatif olmaktadır.

Kullanım

Bir MTD bölümü *JFFS2* formatında veri içermese dahi, blok tipi aygıt ismi üzerinden *jffs2* türünde doğrudan mount edilmeye çalışıldığında işlem başarılı olur, ancak bu yöntem önerilmez.

Bunun yerine öncelikle ilgili MTD bölümünün `flash_erase` ile silinmesi gereklidir:

```
flash_erase /dev/mtd8 0 0
```

Sonra mount işlemi aşağıdaki gibi gerçekleştirilebilir:

```
mount -t jffs2 /dev/mtdblock8 /mnt
```

`flash_erase` ile yapılan silme işleminin MTD'nin istenilen bölümüne ait **karakter aygıt dosyası** üzerinden, mount işleminin ise gene MTD'nin aynı bölümüne ait **blok aygıt dosyası** üzerinden gerçekleştirildiğine dikkat ediniz.

> Not: jffs2 dosya sistemi **raw flash** çiplerine özel üretilmiş olmasına rağmen ilgili **MTD** bölümünü **mount** uygulamasına *mtdblock* emülasyonu üzerinden gösteriyor olmamız, tamamen mount uygulamasının blok tabanlı aygıt arayüzleri ile çalışmasından kaynaklanmaktadır. Mount sonrası jffs2 tarafından **mdtblock** emülasyon katmanı kullanılmaz.

Yukarıdaki örnekte mount işlemi sonrası `/mnt` dizini altına kopyalama yapmak suretiyle ilgili bölüme yazılabilir.

Dosya Sisteminin Yazılması

Bir MTD bölümünü **flash_erase** ile silip *jffs2* türünde mount ettikten sonra standart yöntemlerle kopyalama yaparak ilerlemek yerine, dosya sistemini başka bir yerde oluşturup, tüm dosya sistemine ait *jffs2* imajını hazırlayıp tek seferde kopyalamak da mümkündür.

Bunun için `flash_erase` ile ilgili MTD bölümü silindikten sonra, `nandwrite` ile imajın yazılması sağlanır:

```
flash_erase /dev/mtd8 0 0
nandwrite -p /dev/mtd8 /tmp/jffs.image
```

Ardından ilgili bölüm mount edildiğinde, imaj içerisinde yer alan verilere ulaşılabilecektir:

```
mount -t jffs2 /dev/mtdblock8 /mnt
```

İmaj Oluşturma

Geliştirme yapılan bilgisayardaki bir dizin hiyerarşisini *jffs2* imajına çevirmek için, *mtd-utils* paketinden çıkan `mkfs.jffs2` uygulaması kullanılır. Örnek kullanım:

```
mkfs.jffs2 -p -n -e 128 -d /dizin -o /tmp/jffs.img
```

- `-p` parametresi ile eraseblock'ların sonuna dek *padding* yapılması sağlanır.
- `-n` parametresi *cleanmarker*'ların **konmaması** gerektiğini belirtir. Cleanmarker özel bir JFFS2 node'udur ve blok silme işleminin düzgün tamamlandığını tutmak için kullanılır. NAND flash tiplerinde bu bilgi *OOB* alanında tutulduğundan cleanmarker node'larını istemiyoruz, NOR flash tiplerinde ise bu gereklidir.
- `-e` parametresi ile eraseblock büyüklüğünü belirtiyoruz, kullanılan flash tipine göre doğru değerin ne olduğu öğrenilmelidir. Burada 128 KB verilmiştir.

Erase Block Summary

EBS hesaplamasının temel amacı mount süresinin kısaltılmasıdır.

Temel mantık, özet (*summary*) bilgilerinin her eraseblock sonunda tutulmasından ibarettir.

Bu sayede mount esnasında tüm node'ların taranmasına gerek kalmaz ve sadece özet bilgiler okunur.

`mkfs.jffs2` ile oluşturulan imaj dosyasına *summary* bilgilerini eklemek için *mtd-utils* paketinden çıkan `sumtool` uygulaması kullanılır. Örnek:

```
sumtool -p -n -e 128 -i /tmp/jffs.img -o /tmp/jffs-summed.img
```

Bu yöntem hem NOR hem de NAND tipi flash'lar ile çalışmaktadır. NAND flash tiplerinde hız artışı daha çok olur.

Loopback Mount

Geliştirme yapılan PC mimarisindeki bilgisayarda muhtemelen **raw flash** aygıtı bulunmayacağından ötürü, *jffs2* imajını bilgisayarımızda mount etmek için, MTD emülasyon modülünü kullanabiliriz. İzlenmesi gereken adımlar sırasıyla:

- Eraseblock büyüklüğü bulunur ve byte cinsine çevrilir (128 Kb = 131072 byte)
- `losetup` ile imaj dosyasından bir loopback device oluşturulur:

```
$ sudo losetup /dev/loop0 /tmp/jffs.image
```

- **mtdblock** modülü yüklenir:

```
$ sudo modprobe mtdblock
```

- **block2mtd** modülü aşağıdaki şekilde yüklenir:

```
$ sudo modprobe block2mtd block2mtd=/dev/loop0,131072
```

- Oluşan mtdblock aygıtı mount edilir:

```
$ sudo mount -t jffs2 /dev/mtdblock0 /mnt
```

YAFFS2 Dosya Sistemi

Yet **A**nother **F**lash **F**ilesystem'in ikinci versiyonu olup, **raw flash** aygıtları üzerinde çalışan bir dosya sistemidir.

Charles Manning tarafından Yeni Zelanda kuruluşlu *Aleph One* firması için geliştirilmiştir.

Küçük ve kolay port edilebilir bir yapıya sahip olduğundan, Linux dışındaki ortamlarda ve özellikle çeşitli kapalı kodlu gömülü sistemlerde de kullanım alanı bulmuştur. Kapalı kodlu sistemlerde kullanılabilmesi için ikinci bir lisanslama opsiyonu sunmaktadır.

JFFS2'ye göre tek eksiği sıkıştırılmış dosya sistemi modunda çalışamıyor olmasıdır.

Bununla birlikte diğer tüm alanlarda JFFS2'den daha iyi performans verir.

Günümüzde yerini UBIFS'e bırakmaya başlamıştır.

Kullanım

Bir MTD bölümü *YAFFS2* formatında veri içermese dahi, blok tipi aygıt ismi üzerinden *yaffs2* türünde doğrudan mount edilmeye çalışıldığında işlem başarılı olur, ancak bu yöntem önerilmez.

Öncelikle ilgili MTD bölümünün `flash_erase` ile silinmesi gereklidir:

```
flash_erase /dev/mtd8 0 0
```

Sonra mount işlemi aşağıdaki gibi gerçekleştirilebilir:

```
mount -t yaffs2 /dev/mtdblock8 /mnt
```

Mount işlemi sonrası `/mnt` dizini altına kopyalama yapmak suretiyle ilgili bölüme yazılabilir.

Dosya Sistemi İmajının Yazılması

Bir MTD bölümünü `flash_erase` ile silip *yaffs2* türünde mount ettikten sonra standart yöntemlerle dosyaları kopyalayarak ilerlemek yerine, dosya sistemini başka bir yerde (kendi bilgisayarımızda) oluşturup, *yaffs2* imajını hazırlayıp tek seferde bu hazırlanan dosya sistemi imajını kopyalamak da mümkündür.

Bunun için `flash_erase` ile ilgili MTD bölümü silindikten sonra, `nandwrite` ile imajın yazılması gerçekleştirilebilir.

```
flash_erase /dev/mtd8 0 0
nandwrite -a -o /dev/mtd8 /tmp/yaffs.image
```

Ardından ilgili bölüm mount edildiğinde, imaj içerisinde yer alan verilere ulaşılabilecektir:

```
mount -t yaffs2 /dev/mtdblock8 /mnt
```

İmaj Oluşturma

Geliştirme yapılan bilgisayardaki bir dizin yapısını *yaffs2* imajına çevirmek için, *yaffs2utils* paketinden çıkan `mkyaffs2` uygulaması kullanılır. Örnek kullanım:

```
mkyaffs2 /work/dir /tmp/yaffs.img
```

- İlk parametre olarak imajı oluşturulacak olan dizin belirtilir.
- İkinci parametre ise imajın nereye çıkartılacağını gösterir.
- *yaffs2utils*: http://code.google.com/p/yaffs2utils

UBIFS Dosya Sistemi

UBIFS (Unsorted Block Image File System), JFFS2 gibi **raw flash** aygıtları üzerinde çalışan bir dosya sistemidir.

Nokia mühendisleri ve *Szeged* Üniversitesi'nin (Macaristan) katkılarıyla geliştirilmiştir.

Kernel **2.6.27** versiyonundan itibaren Linux kaynak kodu ile birlikte dağıtılmaya başlanmıştır.

JFFS2'den farklı olarak doğrudan **MTD** aygıtın üzerinde değil, **U**nsorted **B**lock **I**mages aygıtları üzerinde çalışır.

UBI ise MTD aygıtının üzerinde çalışmaktadır.

Tasarım Prensipleri

UBIFS'in tasarımındaki temel hedefler:

- Artan flash kapasiteleriyle büyük dosya sistemleriyle performanslı çalışmak
- Mount süresinin kısaltılması
- Yazma hızının artırılması
- Büyük dosyalara hızlı erişim

şeklinde özetlenebilir.

Kullanım

Oluşturulan bir UBI volume birimi, içerisinde dosya sistemi henüz olmasa dahi **ubifs** türünde mount edilebilir:

```
mount -t ubifs ubi0:test1 /mnt
```

Mount işleminde etiketler kullanılabildiği gibi, volume birim numaraları da kullanılabilir:

```
mount -t ubifs ubi0:1 /mnt
```

UBIFS İmajı Oluşturma

Mevcut bir dizinden UBIFS imajı oluşturmak için *mtd-utils* paketinden çıkan `mkfs.ubifs` uygulaması kullanılır.

```
mkfs.ubifs -m 512 -c 100 -e 128KiB -x lzo -r /work/dir /tmp/ubifs.img
```

- `-m` parametresi minimum IO işlem büyüklüğünü gösterir
- `-c` parametresi ile maksimum *logical eraseblock* adedi belirlenir.
- `-x` parametresi ile sıkıştırma algoritması seçilir.
- `-r` parametresi ile imajı oluşturulacak olan dizin belirtilir.

İmaj oluşturulması tamamlandıktan sonra `ubiupdatevol` uygulaması ile bu volume imajı, volume birimi üzerine yazılır:

```
ubiupdatevol /dev/ubi0_0 /tmp/ubifs.img
```

UBI İmajı Oluşturma

Mevcut bir UBI aygıtı (*/dev/ubi0*) bir veya birden fazla UBI volume biriminden oluşabilir.

Bu durumda bir UBI aygıtının imajı oluşturulacak olursa, `ubiformat` uygulaması ile başka bir UBI aygıtı üzerine yazıldığında, aynı volume birimleri de oluşturulmuş olacaktır.

```
ubiformat /dev/mtd0 -f /tmp/ubi.img
```

UBI imajı oluşturmak için *mtd-utils* paketinden çıkan `ubinize` uygulaması kullanılır.

Ubinize uygulamasını çalıştırmadan önce, INI formatında bir konfigürasyon dosyası oluşturulmalıdır.

Örnek kullanım:

```
ubinize -o /tmp/ubi.img -m 512 -p 128KiB ubinize.cfg
```

Konfigürasyon dosyası içeriği:

```
# ubinize.cfg
[rootfs_volume]
mode=ubi
image=rootfs.ubifs
vol_id=1
vol_type=static
vol_name=rootfs
vol_alignment=1

[data_volume]
mode=ubi
image=data.ubifs
vol_id=2
vol_type=dynamic
vol_name=data
vol_alignment=1
vol_flags=autoresize
```

Root Dosya Sistemi Olarak UBIFS

Root dosya sistemi olarak UBIFS kullanılmak istendiğinde, kernel açılış parametrelerine aşağıdaki değerler eklenmelidir:

- ubi.mtd=0
- root=ubi0:rootfs_label
- rootfstype=ubifs

`rootfs_label` ilgili UBI aygıtı üzerindeki volume birim etiketini göstermektedir.

UBIFS - Write-Back Desteği

JFFS2 dosya sistemi **write-through** modunda çalışır ve değişiklikler senkron biçimde NAND flash'a yazılır.

UBIFS ise **write-back** desteği sayesinde, asenkron modda çalışmaktadır.

Her tür **write** sistem çağrısı ile yapılan yazma işlemleri, Linux'te öncelikle **page cache**'lere yazılır.

Page Cache, genel bir bellek yönetim mekanizmasıdır.

Bir dosyaya yazmaya çalıştığınızda veriler aslında bellekteki page cache'e yazılır ve ilgili page'ler *dirty* olarak işaretlenir. Bu aşamada sistem başarılı bir değer döndürür ama gerçekte veriler fiziksel olarak henüz yazılmış değildir. Asıl yazma işlemi, bazı parametrelere bağlı olarak daha sonra gerçekleşir.

Yazma Operasyonları ve Page Cache

Yazma işlemlerinde Page Cache kullanımından kaynaklanan bu durumla ilgili **write** sistem çağrısının açıklamalarında yer alan önemli notu tekrar vurgulamamız, uygulama geliştirme sürecinde tespiti zor hatalarla karşılaşmamamız için yerinde olacaktır:

```
$ man 2 write
...
NOTES

A successful return from write() does not make any guarantee that data
has been committed to disk.  In fact, on some buggy  implementations,
it does not even guarantee that space has successfully been reserved
for the data. The only way to be sure is to call fsync(2) after you
are done writing all your data.
```

Linux `write-back` Konfigürasyonu

Linux sistemlerde `/proc/sys/vm` dizini altında write-back mekanizmasını ayarlayabileceğimiz dosyalar bulunur. Bu değerler UBIFS'e özgü olmayıp sistemdeki tüm dosya sistemlerini etkiler.

- `dirty_writeback_centisecs` : write-back thread'inin hangi sıklıkta çalışıp *dirty* page cache'leri yazacağını belirtir. (Örnek: 500)
- `dirty_expire_centisecs` : verinin maksimum ne kadar süre *dirty* olarak kalabileceğini gösterir, süre bitiminde periyodik çalışan write-back thread'i gerekli işlemleri yapacaktır. (Örnek: 3000)
- `dirty_background_ratio` : Toplam bellek alanının maksimum ne kadarlık kısmının *dirty* page cache tutmak için kullanılacağını belirtir. Bu değer aşılırsa periyodik write-back thread'i ayarlanan oranı sağlayabilmek için gerekli işlemleri yapar. Soft-Limit olarak düşünülebilir. (Örnek: 40)
- `dirty_ratio` : *dirty* page-cache oluşturabilecek süreçlerin, yazma işlemi öncesinde toplam bellek alanının ne kadarının bu amaçla kullanılabileceğini gösteren orandır. Yazma öncesinde yapılan kontrolde bu değere erişildiği tespit edilirse, öncelikle mevcut *dirty* page cache'ler commit edilir ve bu oran sağlanmaya çalışılır. Hard-Limit olarak düşünülebilir. (Örnek: 60)

UBIFS Write-Buffer

Genel Linux write-back mekanizmasına ek olarak, UBIFS'in kendine özgü **write-buffer** mekanizması da bulunmaktadır.

UBIFS write-buffer alanları, UBIFS kodu içerisinde, page cache ve flash alanı arasında durur. Dolayısıyla write-back mekanizması verileri flash'a değil, UBIFS write-buffer alanına yazar.

Write-buffer alanı page cache ile karşılaştırdığımızda çok küçüktür ve genellikle *NAND page size* değerine eşittir. (512 byte, 2Kb, 4Kb vb.)

NAND page size değeri, NAND aygıtı üzerinde yapılabilecek minimum okuma/yazma miktarını gösterir. Dolayısıyla *dirty* veriler *dirty_expire_centisecs*'lik süre sonrasında commit edilir ancak verinin son kısımları write-buffer yüzünden ekstra bir 3-5 saniyelik gecikmeye tabi tutulabilir.

`write-back` ve Senkronizasyon

Write-back ve write-buffer mekanizmaları karşısında verilerin commit edilme zamanını kontrol için aşağıdaki yöntemler kullanılabilir:

- İlgili bölüm `-o sync` parametresi ile mount edilecek olursa tüm yazma işlemleri senkron olarak yapılır (JFFS2'deki gibi) ancak performans çok ciddi biçimde düşer.
- `fsync(int fd)` çağrısı ile sadece belirli bir *file descriptor*'a ait verilerin commit edilmesi sağlanabilir.
- `fdatasync(int fd)` çağrısı ile dosyanın meta-data bilgilerinin commit **edilmemesi** sağlanır, sadece dosya içeriği commit edilir.
- `open()` çağrısında `O_SYNC` bayrağı kullanılması durumunda ilgili *file descriptor* için yapılan tüm *write* işlemleri senkron olarak gerçekleşir. (fsync ve fdatasync örneğinde pek çok write işleminin bir defada commit edilebileceğini unutmayın)

Ve Daha Çok Senkronizasyon Problemi...

Write-Back mekanizması ve asenkron çalışma yöntemiyle ilgili pek çok detay daha bulunmaktadır ve Kernel geliştiricileri ile uygulama geliştiriciler arasında bir çatışmaya yol açmaktadır. (Bkz: http://lwn.net/Articles/326471)

Konuyla ilgili uzun tartışmalar genelde Linux çekirdeğinin duruma göre bazı ince ayarlara otomatik karar vermesi ve yapması ile kullanıcı adına böyle bir karar vermeyip, zaten halihazırda kullanıcı seviyesinden `sysctl` mekanizmasıyla yapılabilmekte olan bu ince ayarların, kullanıcının sorumluluğuna bırakılması noktasında yaşanmaktadır.

Fikir vermesi açısından aşağıda konuyu özetleyen bir mail ve *Torvalds*'ın yorumu yer almaktadır. İlgili kişiler:

- `Andrew Morton` : 1959 doğumlu ana Linux geliştiricilerinden, halihazırda **Ext3** ve blok

tabanlı aygıtlar için *journaling* desteğiyle ilgili kısımlardan sorumlu

- `Theodore Y. Ts'o` : 1968 doğumlu ana Linux geliştiricilerinden, **ext2** ve **ext3** içerisinde çalışmış, halihazırda **ext4** kodundan sorumlu
- `Linus Benedict Torvalds` : 1969 doğumlu, fazla söze gerek yok, Linux'un yanı sıra `Git` versiyon kontrol sistemini de geliştiren kişi

```
From:     Linus Torvalds <torvalds-AT-linux-foundation.org>
To:       Andrew Morton <akpm-AT-linux-foundation.org>
Subject:  Re: Linux 2.6.29
Date:     Thu, 26 Mar 2009 20:38:43 -0700 (PDT)

On Thu, 26 Mar 2009, Andrew Morton wrote:
>
> Why does everyone just sit around waiting for the kernel to put a new
> value into two magic numbers which userspace scripts could have set?
>
> My /etc/rc.local has been tweaking dirty_ratio, dirty_background_ratio
> and swappiness for many years.  I guess I'm just incredibly advanced.

.. and as a result you're also testing something that nobody else is.

Look at the complaints from people about fsync behavior that Ted says he
cannot see. Let me guess: it's because Ted probably has tweaked his
environment, because he is advanced. As a result, other people see
problems, he does not.

That's not "advanced". That's totally f*cking broken.

Having different distributions tweak all those tweakables is just even
_more_ so. It's the anti-thesis of "advanced". It's just stupid.

We should aim to get it right. The "user space can tweak any numbers they
want" is ALWAYS THE WRONG ANSWER. It's a cop-out, but more importantly,
it's a cop-out that doesn't even work, and that just results in everybody
having different setups. Then nobody is happy.
```

Cramfs Dosya Sistemi

Cramfs *Linus Torvalds* ve *Daniel Quinlan* tarafından yazılan, write desteği olmayan, gömülü sistemlerde kullanılmaya uygun basit bir read-only dosya sistemidir.

Blok tabanlı aygıtlar için geliştirilmiş olmakla birlikte, yazma işlemi tanımlı olmadığından **mtdblock** sürücüsü üzerinden **raw flash** tabanlı sistemlerde kullanılabilmektedir.

Bozuk blok yönetimi bulunmadığından bozuk bloklarla karşılaşılması halinde sorunlar yaşanabilir.

Zlib kütüphanesi ile sıkıştırma desteği sunmaktadır. Maksimum dosya boyutu **16 MB**, maksimum dosya sistemi boyutu **272 MB**'tır.

SquashFS dosya sistemi öncesinde yoğun bir kullanım alanı bulmuş olsa da günümüzde nadir olarak kullanılmaktadır. SquashFS benzer başlıklarda daha iyi performans sunmaktadır.

İmaj Oluşturma

Cramfs imajı oluşturmak için *cramfstools* paketinden çıkan `mkcramfs` uygulaması kullanılır.

```
mkcramfs /work/dir cram.img
```

Oluşturulan imaj dosyası `dd` uygulaması ile MTD aygıtı üzerine yazılır:

```
dd if=cram.img of=/dev/mtdblock3
```

Aşağıdaki şekilde mount edilir:

```
mount -t cramfs /dev/mtdblock3 /mnt
```

SquashFS Dosya Sistemi

Write desteği olmayan, read-only bir dosya sistemidir.

Blok tabanlı aygıtlar için geliştirilmiş olmakla birlikte, yazma işlemi tanımlı olmadığından **mtdblock** sürücüsü üzerinden **raw flash** tabanlı sistemlerde kullanılabilmektedir.

Bozuk blok yönetimi bulunmadığından bozuk bloklarla karşılaşılması halinde sorunlar yaşanabilir.

Sıkıştırma oranı ve performansı nedeniyle, read-only dosya sistemleri arasında en çok kullanılan tercih olmuştur.

İmaj Oluşturma

SquashFS imajı oluşturmak için *mtd-utils* paketinden çıkan `mksquashfs` uygulaması kullanılır.

```
mksquashfs /work/dir squash.img
```

Oluşturulan imaj dosyası `dd` uygulaması ile MTD aygıtı üzerine yazılır:

```
dd if=squash.img of=/dev/mtdblock3
```

Aşağıdaki şekilde mount edilir:

```
mount -t squashfs /dev/mtdblock3 /mnt
```

Minix Dosya Sistemi

FAT Dosya Sistemi

Ext2, Ext3, Ext4 Dosya Sistemleri

Watchdog Kullanımı

Watchdog mekanizmaları, çalışan ana yazılımların beklenmedik bir şekilde sonlanması sonrasında sistemin yeniden başlatılabilmesine imkan tanır.

Bu mekanizma genellikle donanım tabanlı bir zamanlayıcı (Watchdog Timer - WDT) ile gerçekleştirilir.

Sistemin genel çalışma prensibi oldukça basittir:

- WDT zamanlayıcısını örnek olarak 60 saniye için başlat
- 60 saniyelik süre bitmeden hayatta olduğunu ispat et ve yeni bir 60 saniye kazan

Periyodik olarak zamanlayıcıya hayatta olduğunuzu söylediğiniz müddetçe sorun olmayacaktır. Ancak bu işlemi yapamaz hale gelirseniz (uygulamanız bir SEGFAULT ile sonlanmışsa vb.) zamanlayıcının süresi dolduğunda sistem otomatik olarak yeniden başlatılacaktır.

> Buradaki yeniden başlatma işlemi kontrollü (reboot) değil, **reset** şeklindedir.

Kernel Panic Sistemi İle Fark

Watchdog haricinde görece benzer bir işlevi Linux çekirdeği içerisindeki **panic handler** ile de yapabildiğimizi hatırlayınız. Eğer çekirdek içerisindeki yer alan kodlarda herhangi bir kritik hata oluşması nedeniyle tüm çekirdek çalışamaz hale gelirse, sistemin `panic` parametresi ile açılmış olması halinde otomatik yeniden başlatma (reset) işlemi çekirdek tarafından yapılmaktadır.

Ancak bu yöntem sadece çekirdek seviyesindeki hatalarda işe yaramaktadır. Watchdog ise kullanıcı kipi hatalarını da yakalamaya ve bir şeyler ters gittiğinde en azından sisteminizi yeniden başlatabilmenize imkan tanımaktadır.

Çekirdek Desteği

Sistemdeki watchdog donanımının kullanımı için çekirdek seviyesinde sürücü desteği sağlanmış olmalıdır.

Linux çekirdeğinde watchdog donanım sürücüleri için genel bir Watchdog Driver API mevcut olup, tüm donanımlar tarafından aynı arayüz, yetenekleri doğrultusunda sağlanır. Detaylar için https://www.kernel.org/doc/Documentation/watchdog/watchdog-api.txt belgesini

inceleyebilirsiniz.

Temel Kullanım Modu

Tüm watchdog sürücülerinin en az bu modu desteklemesi beklenir, dolayısıyla watchdog donanımızın desteklenmesi halinde her durumda bu çalışma yöntemini kullanabilirsiniz.

Çalışma mantığı şu şekildedir:

- Uygulama katmanında `/dev/watchdog` aygıt dosyası yazma modunda açılır
- Öntanımlı olarak 60 saniyelik bir zamanlayıcı başlar
- 60 saniye dolmadan, bu dosyaya herhangi bir veri yazılır (tek karakter de olabilir) ve zamanlayıcının yeniden başlatılması sağlanır
- Uygulama, kendi içinde bir döngüde kalıp zamanında watchdog dosyasına yeni bir veri yazamaz ise, süre dolumunda sistem yeniden başlatılır
- Açık durumdaki `/dev/watchdog` dosyası kapatılır ise, watchdog sistemi devre dışı bırakılır

Görüleceği üzere Unix sistemlerde hemen her şeyin dosya arayüzü ile kullanıcı katmanına sunumu yaklaşımı burada da benimsenmiştir. Uygulamanız içerisinde yukarıdaki iş akışını dosya işlemleriyle kolaylıkla gerçekleyebilirsiniz.

Yalnız burada dikkat edilmesi gereken husus, `/dev/watchdog` dosyasının yazma modunda açılması ile başlayan zamanlayıcının dosyanın kapatılması halinde devre dışı kalıyor oluşudur. İlk başta önemli bir problem olarak görünmeyebilir, ancak Linux çekirdeği herhangi bir yöntemle sonlanan kullanıcı kipi uygulamaları için, açık halde bulunan tüm dosyaları da kapatır. Bu nedenle watchdog mekanizmasını işletmek için uygulamanızda `/dev/watchdog` açık halde iken uygulamanız kontrolsüz biçimde sonlanacak olursa, dosya kapatıldığı için watchdog da devre dışı kalacak ve beklediğiniz zamanlayıcı süresi dolumundaki yeniden başlatma işlemi gerçekleşmeyecektir.

Peki o halde watchdog ne işe yarayacak diye sorabilirsiniz. Endişeye mahal yok.

Bu dizayn ile temelde 3 çözüm yönteminiz bulunuyor:

1. Ana uygulama(ları)nız içerisinde watchdog yönetmek yerine, watchdog'u yöneten ayrı bir uygulama yazmak (veya hazır bir watchdog daemon kullanmak)
2. Magic Close özelliği destekleniyorsa kullanmak
3. Çekirdek derleme sürecinde `CONFIG_WATCHDOG_NOWAYOUT` opsiyonu seçerek, sürücünüzün de desteklemesi halinde watchdog dosyasının kapanmasıyla zamanlayıcının durdurulmasını engellemek

Magic Close Özelliği

Bazı sürücüler Magic Close adı verilen bir özelliği desteklemektedir. Bu destek sayesinde, watchdog dosyası kapatılmadan önce **V** karakteri dosyaya yazılmaz ise, dosya kapatılsa dahi zamanlayıcı çalışmaya devam etmekte ve zamanı dolduğunda sistemi yeniden başlatmaktadır.

Eğer özellikle watchdog zamanlayıcısını durdurmak istiyorsanız, dosyayı kapamadan önce **V** karakterini yazabilir ve sonra kapama işlemini gerçekleştirebilirsiniz.

Magic Close desteği daha rahat bir kontrol sağlıyor olmasına karşın sisteminizde desteklenmiyor olması ihtimali kuvvetle muhtemeldir. Bu özelliğin desteklenip desteklenmediğini ioctl arayüzü üzerinden `WDIOC_GETSUPPORT` opsiyonuyla sorgulayabilirsiniz. Aşağıdaki örnek uygulama ile watchdog sürücünüzün Magic Close, Set Timeout ve Keep Alive Ping özelliklerini destekleyip desteklemediğini öğrenebilirsiniz:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/watchdog.h>

#define WATCHDOG_DEVICE "/dev/watchdog"

int main(void)
{
    struct watchdog_info info;
    int fd;

    if ( (fd = open(WATCHDOG_DEVICE, O_RDWR)) < 0) {
        perror("open failure");
        exit(EXIT_FAILURE);
    }

    if (ioctl(fd, WDIOC_GETSUPPORT, &info) < 0) {
        perror("ioctl");
        exit(EXIT_FAILURE);
    }

    printf("Magic Close Feature  : %s\n", (info.options & WDIOF_MAGICCLOSE) ? "Yes" :
"No");
    printf("Set Timeout Feature  : %s\n", (info.options & WDIOF_SETTIMEOUT) ? "Yes" :
"No");
    printf("KeepAlivePing Feature: %s\n", (info.options & WDIOF_KEEPALIVEPING) ? "Yes"
 : "No");

    close(fd);
}
```

ioctl Arayüzü

Tüm watchdog sürücüleri, destekledikleri özellikler için ioctl arayüzü de sağlamaktadırlar.

Watchdog zamanlayıcıyı beslemek ve yeni bir tur daha süre kazanmak için `WDIOC_KEEPALIVE` ile aşağıdaki gibi bir ioctl çağrısı yapabilirsiniz. 3. parametrenin ne olduğunun bir önemi yoktur:

```c
ioctl(fd, WDIOC_KEEPALIVE, 0);
```

Sürücü tarafından desteklenmesi halinde 60 saniyelik öntanımlı watchdog zamanlayıcı değerini değiştirmek isterseniz, `WDIOC_SETTIMEOUT` ile aşağıdaki gibi bir ioctl çağrısı yapabilirsiniz:

```
int timeout = 20; /* saniye */
ioctl(fd, WDIOC_SETTIMEOUT, &timeout);
```

Benzer şekilde, zamanlayıcıda kalan zamanı `WDIOC_GETTIMEOUT` ile aşağıdaki gibi sorgulayabilirsiniz:

```
int remaining;
ioctl(fd, WDIOC_GETTIMEOUT, &remaining);
```

Sistemin son defa yeniden başlatılması işleminin watchdog yüzünden olup olmadığını `WDIOC_GETBOOTSTATUS` ile sorgulayabilirsiniz:

```
int bootfromwatchdog;
ioctl(fd, WDIOC_GETBOOTSTATUS, &bootfromwatchdog);
```

Daha detaylı özellikler için çekirdek içerisindeki dokümantasyonu inceleyebilirsiniz.

CPU Frequency Scaling

Cpu Frequency Scaling katmanı, Linux çekirdeğinin güç tüketimini azaltmak için işlemci çalışma frekansını azaltıp artırmasına imkan verir.

Frekans yönetimini sağlayan *conservative*, *ondemand*, *userspace*, *powersave* ve *performance* adında 4 farklı governor (yönetici) opsiyonu bulunmaktadır.

Linux çekirdeği versiyon **3.4** ve yukarısında gerekli modüller otomatik olarak yüklenmekte ve **ondemand** governor aktif edilmektedir.

Cpufreq Governor

Her bir governor basitçe CPU çalışma frekansının ne olması gerektiğine dinamik olarak karar veren algoritmalar olarak düşünülebilir.

Governor algoritmaları, belirlenen minimum ve maksimum değerler içerisinde kalmak şartıyla, istatistiki olarak seçilen politikaya uygun olarak kararlar vermeye çalışırlar.

Performance Governor

Bu governor sistemden sürekli maksimum performans istendiği durumda kullanılır.

Seçilen çalışma frekansı, *scaling_max_freq* değerini geçemez.

Powersave Governor

Sistemde güç tüketimini minimuma getirmeye çalışır.

Anlık cpu kullanım isteklerine tepkisi daha yavaş olur.

Seçilen değerler *scaling_min_freq* ve *scaling_max_freq* arasında olur.

Güç tüketiminin çok önemli olduğu senaryolarda önerilir.

Ondemand Governor

Bu governorda CPU frekansı ihtiyaç doğrultusunda anlık olarak değiştirilir.

Seçim algoritması cpu kullanımına dair aralıklarla örnekler alır ve bu örneklemi karar mekanizmasında kullanır.

Algoritma, sysfs üzerinden optimize edilebilir.

ondemand/sampling_rate dosyası üzerinden kaç mikrosaniyede bir örnek alınacağı belirtilebilir.

Bunun yanı sıra *up_threshold*, *ignore_nice_load*, *sampling_down_factor*, *powersave_bias* parametreleriyle algoritmanın karar üretme süreci daha hassas biçimde ayarlanabilir.

Conservative Governor

Ondemand governor'un özelleştirilmiş bir hali olarak düşünülebilir.

Ondemand'dan farklı olarak, Cpu frekansını azaltmak veya artırmak gerektiğinde bu işlemi kademeli olarak yapmaktadır.

freq_step, *down_threshold* ve *sampling_down_factor* parametreleriyle algoritmanın davranışında bir takım özelleştirmeler yapmak mümkündür.

Userspace Governor

Bu governor sistemdeki **root** kullanıcısının CPU frekansını spesifik olarak belirli bir değere ayarlayabilmesini sağlar.

Governor yüklendikten sonra, sysfs üzerinden *scaling_setspeed* dosyası aracılığıyla istenen çalışma frekansı ayarlanabilir.

Herhangi bir cpu hızını seçmek mümkün değildir. İlgili CPU için kullanılabilecek CPU hız değerleri sysfs üzerinde *scaling_available_frequencies* dosyasında belirtilmiştir:

```
$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
300000 600000 800000 1000000
```

Aktif Governor Seçimi

Sistemde o anda aktif olan governor *scaling_governor* dosyası üzerinden öğrenilebilir:

```
$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
performance
```

Aktif governor bu dosyaya ilgili governor ismi yazılmak suretiyle değiştirilebilir. Örnek olarak *userspace* governor kullanmak için:

```
# echo userspace > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

Örnek Uygulama - 5000 Asal Sayı Bulma

Aşağıdaki örnek uygulamayı `prime.c` adıyla kaydedip farklı cpu governor'lar aktif iken test edelim.

```
/* prime.c */
#include <stdio.h>

int main()
{
        int n = 5000;
        int i = 3, count, c;

        for (count = 2; count <= n;) {
                for (c = 2; c <= i - 1; c++) {
                        if (i % c == 0) {
                                break;
                        }
                }
                if (c == i) {
                        count++;
                }
                i++;
        }
        printf("found %d prime numbers\n", count);
        return 0;
}
```

Uygulamayı şu şekilde derleyebilirsiniz:

```
$ gcc -o prime prime.c
```

Ardından sistemdeki farklı cpu governor veya saat frekansları için toplam zamanları `time` komutu yardımıyla ölçtüğümüzde aşağıdakine benzer bir sonuç alınmaktadır:

Aktif Governor	İşlem Süresi
Powersave	15.91 s
Conservative	6.84 s
Ondemand	5.05 s
Performance	4.70 s
Userspace - 300 Mhz	15.91 s
Userspace - 1000 Mhz	4.70 s

cpufreq-utils

Frekans değişimi işlemlerini kolaylaştırmak amaçlı `cpufreq-utils` paketinden çıkan uygulamalar kullanılabilir.

`cpufreq-utils` paketini, http://ftp.sunet.se/pub/Linux/kernel.org/linux/utils/kernel/cpufreq/cpufrequtils.html adresinden indirip derleyebilirsiniz. Paket içerisinden çıkan `cpufreq-info` ve `cpufreq-set` uygulamaları ile işlemleri kolay bir şekilde gerçekleştirmek mümkündür.

cpufreq-info

```
$ sudo cpufreq-info
cpufrequtils 008: cpufreq-info (C) Dominik Brodowski 2004-2009
Report errors and bugs to cpufreq@vger.kernel.org, please.
analyzing CPU 0:
  driver: generic_cpu0
  CPUs which run at the same hardware frequency: 0
  CPUs which need to have their frequency coordinated by software: 0
  maximum transition latency: 300 us.
  hardware limits: 300 MHz - 1000 MHz
  available frequency steps: 300 MHz, 600 MHz, 800 MHz, 1000 MHz
  available cpufreq governors: conservative, ondemand, userspace, powersave, performan
ce
  current policy: frequency should be within 300 MHz and 1000 MHz.
          The governor "powersave" may decide which speed to use
          within this range.
  current CPU frequency is 300 MHz (asserted by call to hardware).
  cpufreq stats: 300 MHz:nan%, 600 MHz:nan%, 800 MHz:nan%, 1000 MHz:nan%
```

cpufreq-set

- `-d` parametresi ile minimum frekans, `-u` parametresi ile maksimum frekans, `-g` parametresi ile governor seçilir.
- Minimum 300 Mhz, maksimum 600 Mhz ve *ondemand* governor seçmek için:

```
$ sudo cpufreq-set -d 300000 -u 600000 -g ondemand
```

- *userspace* governor aktif iken `-f` parametresi ile spesifik bir frekans seçimi yapmak için:

```
$ sudo cpufreq-set -g userspace
$ sudo cpufreq-set -f 600000
```

Buildroot

Gömülü Linux sistemlerinin kullanımında temel olarak 3 alternatif yöntem bulunmaktadır:

- Önceden hazırlanmış bir dağıtım kullanmak (Debian, Ångström vb.)
    - Desteklenen mimari sayısı az
    - Özelleştirmek ve ihtiyaç duyulmayan bileşenlerin çıkartılması zor
    - Kaynak kodlardan yeniden inşa etmek çok güç
- Elle özel bir dağıtım hazırlamak
    - Uzun ve yorucu bir süreç
    - Çapraz derlemeyle ilgili sorunlu uygulamalara dair detayların öğrenilmek zorunda olması
    - Uygulamaların birbirine olan bağımlılıklarının yönetilmesi güç
- Elle özel bir dağıtım hazırlarken otomatik inşa sistemi kullanmak
    - Her iki yöntemin kötü yanlarını bertaraf edip iyi yanlarını biraraya getirmeye çalışır

Buildroot projesi, benzer işlemleri sıklıkla yapmak zorunda kalan gömülü Linux sistem geliştiricilerinin hayatını kolaylaştırmayı, bunu yaparken de çok karmaşık ve öğrenilmesi başlı başına problem olan yeni bir sistem daha üretmek yerine, geleneksel araçlarla süreci yönetmeyi hedefler.

Tarihçe

Buildroot, **2001** yılında *uClibc* geliştiricileri tarafından test amaçlı geliştirilmeye başlandı.

2005 yılında geliştirici sayısı biraz daha arttı, ancak profesyonel bir araç olmanın halen uzağında idi.

2009 yılına kadar görece kontrolsüz bir yapıda proje değişen hızlarda büyümeye devam etti.

2009 yılından itibaren **Peter Korsgaard** projenin yeni sorumlusu oldu. Bu gelişmenin ardından önemli dizayn değişiklikleri, düzenli çıkan versiyonlar ve buildroot kullanan sistemlerdeki artış projeyi bugünkü haline getirdi.

2015 Ocak ayı itibariyle, Buildroot projesi toplam **35 MB** kaynak kodundan oluşmakta olup, içerisinde **1415** adet uygulama paketi yer almaktadır. **300**'den fazla geliştirici projeye çeşitli seviyelerde katkıda bulunmaktadır.

Yaklaşım

Buildroot temel olarak **Makefile** sistemi üzerine inşa edilmiştir. Sistem yaklaşık olarak **1500** adet Makefile dosyasından (`.mk` uzantılı) oluşur.

Tüm süreç öncelikle ana dizinde yer alan `Makefile` dosyasının okunmasıyla başlar. Yapılan işlemlere göre gerektiğinde ilgili diğer `.mk` uzantılı Makefile dosyaları *include* edilerek ilerlenir.

> Not: Projede çalışırken **make** hedeflerini (`make strace` , `make rootfs-clean` vb.) **TAB** tuşu ile otomatik tamamlattırmak mümkündür, ancak otomatik tamamlama fonksiyonunun tüm dosyaları işlemesi gerektiğinden Makefile sayısının çokluğu nedeniyle bu işlem biraz zaman almaktadır. Böyle bir durumla karşılaşırsanız sabırla bekleyin (5-20 saniye) ve bu özelliği şimdilik fazla kullanmamaya bakın.

Konfigürasyon

Buildroot projesinde konfigürasyon işlemleri için, Linux kernel ve Busybox gibi projelerde kullanılan **Kconfig** sistemi kullanılmaktadır.

- Kaynak: https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt
- Kconfig sistemiyle kolay bir şekilde
    - Seçenekler arasındaki bağımlılıklar
    - Farklı türlerde değişkenler
    - Şartlı akışlar ve öntanımlı değerler
    - Sistem içi yardım metinleri

kolayca tanımlanabilmektedir.

Konfigürasyon Seçenekleri

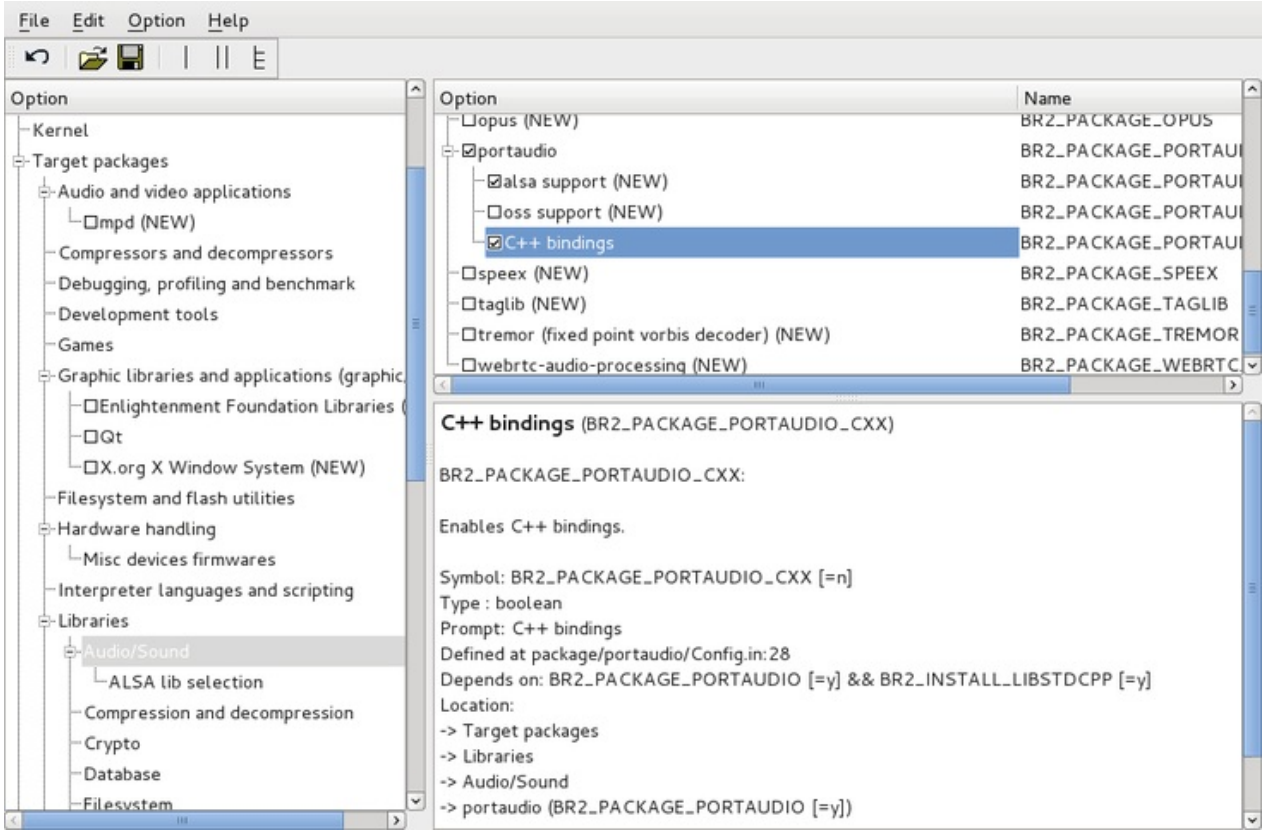
- Kconfig sistemi ile yapılacak konfigürasyon işleminin sonucunda yapılan seçimler kayıt edilirse, ana dizinde **.config** dosyası oluşacaktır
- Temel amaç bu dosyayı oluşturmak olduğundan, daha önce oluşturulmuş olan farklı bir .config dosyası kopyalanmak suretiyle de aynı seçim kümesine ulaşılır
- Buildroot altında öntanımlı gelen config dosyaları, **configs/** dizini altında bulunur (Örnek: beaglebone_defconfig)

- Konfigürasyon işleminde kullanacağımız GUI aracı için birden fazla seçenek mevcuttur:

Yöntem	Açıklama	İhtiyaç Duyulan Kütüphaneler
xconfig	Qt-4 backend	qt4-dev
gconfig	Gtk-2 backend	gtk-2-dev, libglade-2-dev
nconfig	Ncurses backend	libncurses-dev
menuconfig	curses backend	libncurses-dev

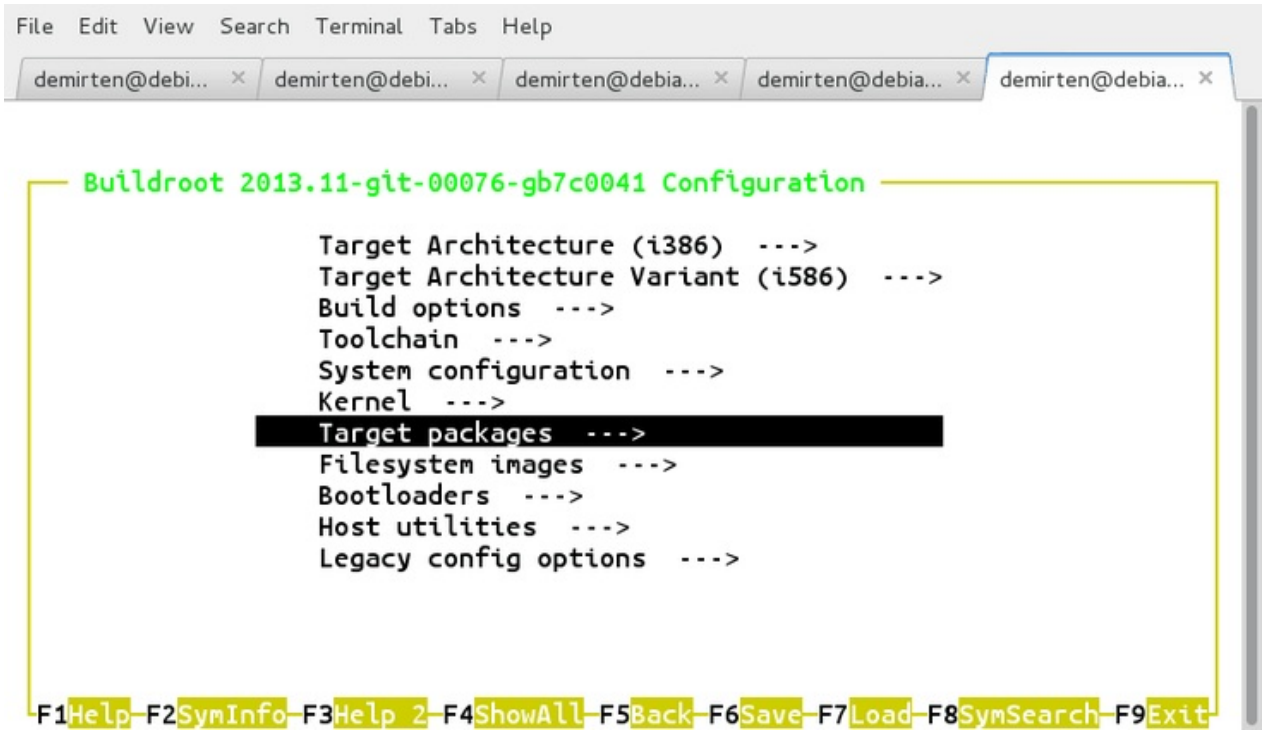
xconfig

- Kullanım: `make xconfig`
- Qt4 tabanlı bir arayüz sunar.



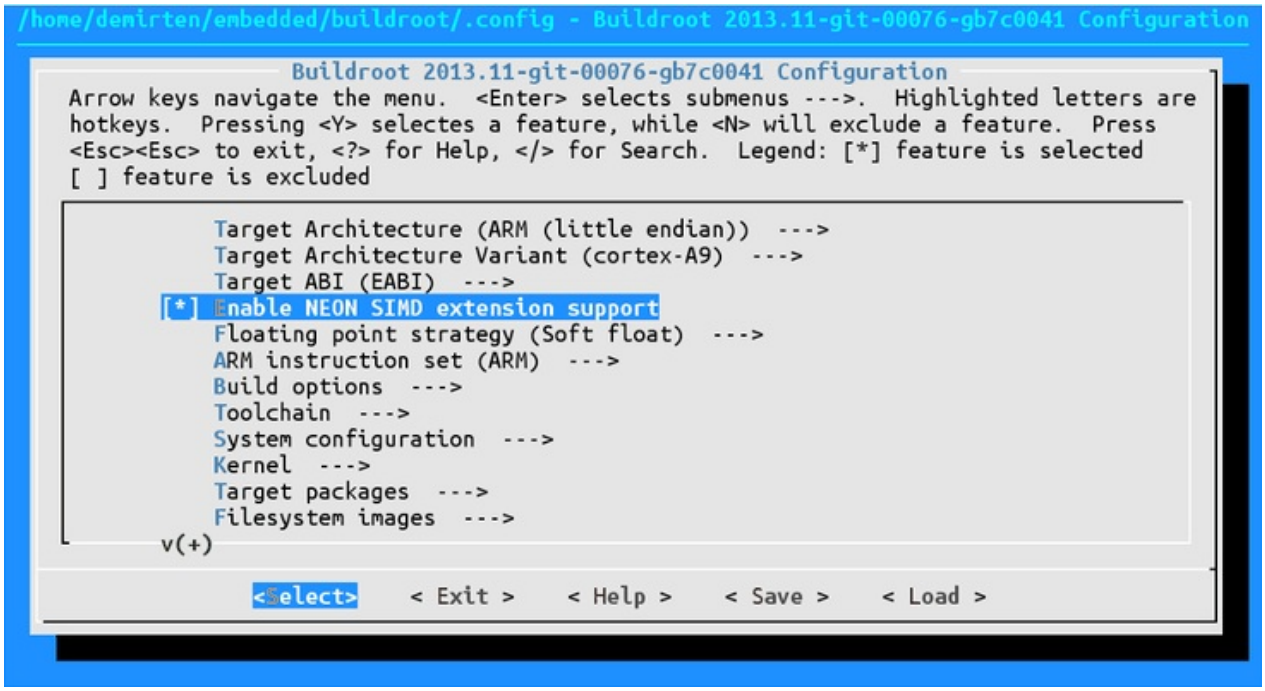
nconfig

- Kullanım: `make nconfig`
- ncurses tabanlı bir arayüz sunar.



menuconfig

- Kullanım: `make menuconfig`
- curses tabanlı bir arayüz olup, kullanımını önermekteyiz.
- Tüm arayüzlerde ilgili seçenek için girilmiş yardım bilgisine ulaşılabilmektedir (curses arayüzünde **h** kısayolu ile)



Buildroot Kodunun İndirilmesi

- buildroot kararlı sürümler her 3 ayda bir yayınlanmakta olup, http://buildroot.uclibc.org/download.html adresinden güncel kararlı sürüm indirilebilir.
- Özellikle buildroot arşivine projenize özgü eklemeler yapacaksanız, bu durumda **git** üzerinden projeyi klonlayıp, farklı bir **branch** oluşturarak, değişikliklerinizi de versiyon takibine alabilirsiniz.
- Bu şekilde buildroot projesindeki yenilikleri de **pull** edip, kendi çalıştığınız branch içerisine **merge** etmek mümkün olacaktır.

  $ git clone git://git.buildroot.net/buildroot

Out-Of-Tree Çalışma Modeli

- Buildroot out-of-tree çalışma modelini destekler
- Bu şekilde aynı buildroot çalışma dizinini temel alarak, farklı konfigürasyonlarda, farklı cpu aileleri için build işlemini yapmak mümkündür
- Bu sayede firma içerisinde aynı veya farklı projelerde buildroot için hazırlanmış ek paketler, konfigürasyonlar var ise, tek bir kaynak kod dizininde bulundurulabilir bu dizinde yer alan buildroot aracılığıyla, farklı çalışma dizinlerinde farklı profillerde çıktılar üretilebilir
- Bu model kullanılmadığında oluşan çıktılar, buildroot kaynak kodlarının bulunduğu yerde **output/** dizini altında oluşturulur (bu yöntemi önermemekteyiz)

Hazır Konfigürasyon Kullanımı

- Buildroot içerisinde *Linux kernel* projesinde olduğu gibi, önceden tanımlanmış hazır konfigürasyon dosyaları bulunur
- Bu dosyalar içerisinde genellikle bir paket seçim kümesi ve temel dosya sistemi ayarları yer almaktadır
- Mevcut bir konfigürasyonu kullanmak için:

  $ make pandaboard_defconfig

- Mevcut konfigürasyon dosyaları `configs/` alt dizininde yer almaktadır.
- Temel konfigürasyon bu şekilde hızlıca yapıldıktan sonra, `make menuconfig` komutu yardımıyla *toolchain* vb. diğer ayarlar da yapılıp inşa sürecine geçilebilir

Boş Konfigürasyon İle Başlamak

- Eğer hazır bir konfigürasyon dosyası kullanmak istemiyorsanız, aşağıdaki komutla boş bir konfigürasyon dosyası ile de işleme başlayabilirsiniz:

  $ make menuconfig

- Bu şekilde işleme başlandığında, mevcut paketler arasından sadece `busybox` 'ın ve mimari olarka `i386` 'nın seçişi olduğu temel bir konfigürasyon karşınıza gelecektir.
- Sırasıyla buradaki menülerden gerekli seçimleri yapıp oluşturduğunuz konfigürasyonu kaydedip, inşa sürecine geçebilirsiniz.

Out-Of-Tree Modeli İle Başlamak

- Out-of-tree modeli çalışma yöntemini kullanmak istiyorsak, `O` değişkenine değer atamak suretiyle *make* uygulamasını çalıştırmalıyız
- Hazır bir konfigürasyon dosyasını baz alacaksak:

  $ make O=/path/to/project_dir pandaboard_defconfig

- Boş bir konfigürasyon ile başlayacaksak:

  $ make O=/path/to/project_dir

- Bu komutun ardından parametre olarak verdiğimiz dizine geçmeli ve geri kalan tüm işlemleri, ilgili dizin altında iken yapmalıyız.

Target Options Bölümü

- Konfigürasyon ekranındaki ilk menü *Target options* şeklindedir
- Burada temel olarak işlemci ailesine ilişkin seçimlerin yapılması gereklidir
- Yapılan seçimlere bağlı olarak inşa sürecinin devamında derlenecek olan tüm uygulamara, *-march*, *-mcpu*, *-mtune* gibi mimari spesifik doğru derleyici parametreleri geçirilecektir
- Örnek bir seçim kümesi şu şekilde olabilir:

Başlık	Değer
Target Architecture	ARM (little endian)
Target Architecture Variant	cortex-A8
Target ABI	EABI
Floating point strategy	VFPv3-D16
ARM instruction set	ARM

Build Options Bölümü

- Bu bölümde temel olarak inşa sürecinin genelini ilgilendiren ayarlar yapılır
- Burada yer alan ayarların çoğu öntanımlı halinde bırakılabilir
- Üzerinde durulması gereken başlıklar ise:
  - Download dir
  - gcc optimization level
  - location of a package override file

Build Options Bölümü : Download Dir

- *Download dir* seçimi ile, inşa sürecinde otomatik olarak indirilecek dosyaların hangi dizinde tutulacağı belirtilir
- Bir seçim yapılmaması halinde, öntanımlı olarak buildroot inşa sürecinde kullandığınız ana dizin altında (Out-of-tree modelini kullanıp kullanmadığınıza göre değişkenlik gösteren) `dl/` alt dizinde tutulacaktır
- Buildroot ile çalışırken download işlemlerini buildroot çalışma dizinlerinin dışında, ortak bir dizin altına yapmakta fayda vardır
- Bu şekilde birden fazla buildroot projesinde çalışırken aynı dosyaları tekrar tekrar download etmek zorunda kalmaz ve hız kazanırsınız
- Birden fazla çalışanın olduğu veya internet erişiminin sınırlandığı çalışma ortamlarında da yerel ağda bir NFS veya http sunucusu üzerinden ortak bir download dizini kullanmak tercih edilebilir

Build Options Bölümü : gcc optimization

- Bu bölümde inşa sürecinde tüm uygulamalar için *gcc* derleyicisine geçirilecek optimizasyon seviyesi parametresi genel olarak belirlenmektedir
- Merkezi bir noktada bu şekilde optimizasyon seviyesinin belirlenebilmesi, sisteminizde yer alacak tüm uygulamaların istediğiniz optimizasyon seviyesinde üretilmesini garanti eder
- Öntanımlı değeri *optimize for size* olmakla birlikte, günümüzde gömülü sistemler için de ana depolama birimi kapasitelerinin arttığını göz önünde bulunduracak olursak, optimizasyon seviyesi 2 veya 3'ü de seçmek anlamlı olmaktadır.

Build Options Bölümü : package override file

- Buildroot içerisindeki ileri düzey konulardan biri olan **OVERRIDE** mekanizması için package override dosyasının nerede aranacağı burada belirtilir
- Package override dosyası temel olarak, buildroot içerisinde tanımlı paketlere ilişkin bazı kuralları değiştirmenize olanak sağlamaktadır
- Özellikle geliştirme yapmakta olduğunuz bir yazılım için buildroot paketi aracılığıyla derleme yapmak istediğinizde kullanımı zorunlu olmaktadır
- Konu ayrıntılarına ilerleyen bölümlerde değilinecek olup, bu değer öntanımlı olarak bırakılabilir veya sistem içerisinde ortak bir dizine yer alan ve diğer buildroot projelerinde de kullanılması istenen bir dosya yolu seçilebilir.

Toolchain Seçimi

- Bu bölüm içerisinde inşa sürecinde çapraz derleme işleminde kullanılacak **toolchain** seçimi ve detaylı konfigürasyonu yapılacaktır
- Temel olarak 2 tip toolchain bulunmaktadır:
    - Buildroot toolchain
    - External toolchain
- Buildroot toolchain tipi seçildiğinde, öncelikle toolchain'in kendisi inşa edilecektir
- Bu süreç oldukça uzun vakit almakta ve buildroot'tan bağımsız olarak, toolchain inşa süreçlerinin genel anlamda sıkıntılı olması sebebiyle başarısızlıkla da sonuçlanabilmektedir

- Özel bir nedeni yok ise toolchain'in kendisini üretme sürecinden kaçınılmasını önermekteyiz

Toolchain Seçimi : External Toolchain

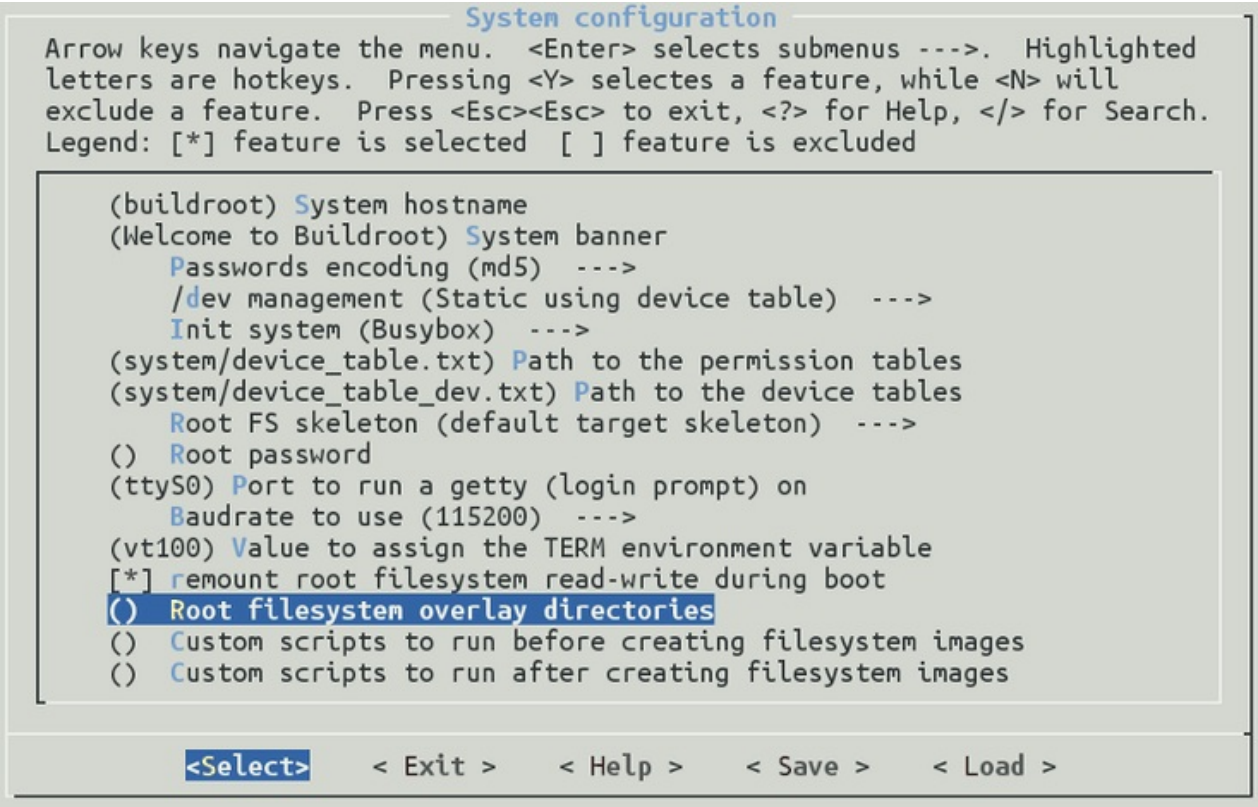
- Toolchain tipi olarak *External Toolchain* seçilmesi durumunda **Toolchain** ve **Toolchain origin** başlığında 2 yeni seçenek belirecektir
- Burada *Sourcery CodeBench*, *Arago*, *Linaro* tarafından üretilen çeşitli toolchain versiyonları listelenmektedir
- Bu versiyonlardan biri seçildiğinde, *Toolchain origin* olarak *to be downloaded or installed* opsiyonu seçilerek internet üzerinden download edilebilir veya daha önceden sisteminizde zaten var ise ilgili dizin beliritilebilir
- **Custom toolchain** opsiyonun seçilmesi halinde ise sisteminizde önceden mevcut olan bir toolchain dizinini seçmeniz mümkündür

System Configuration

- Bu bölümde oluşturulacak olan dosya sistemiyle ilgili bazı temel ayarlar yapılabilmektedir

Başlık	Ayar
System hostname	Sistemin adı, hostname
System banner	Login işlemi sonrası görünecek metin
Password encoding	Parole kripto algoritması
/dev management	/dev dizini yönetim metodolojisi
Init system	Sistem açılışında kullanılacak `init` metodolojisi
Root FS skeleton	Sistemde kullanılacak iskelet dosya sistemi
Root password	Root parolası
Port to run getty	Seri konsol login için kullanılacak port
read-write remount	Açılış süreci sonrası dosya sistemi yazma konumu
Root filesystem overlay dirs	Dosya sistemi üretimi öncesinde kopyalanması istenen dizinler
Custom scripts	Dosya sistemi üretimi öncesi ve sonrası çalıştırılacak betikler

System Configuration



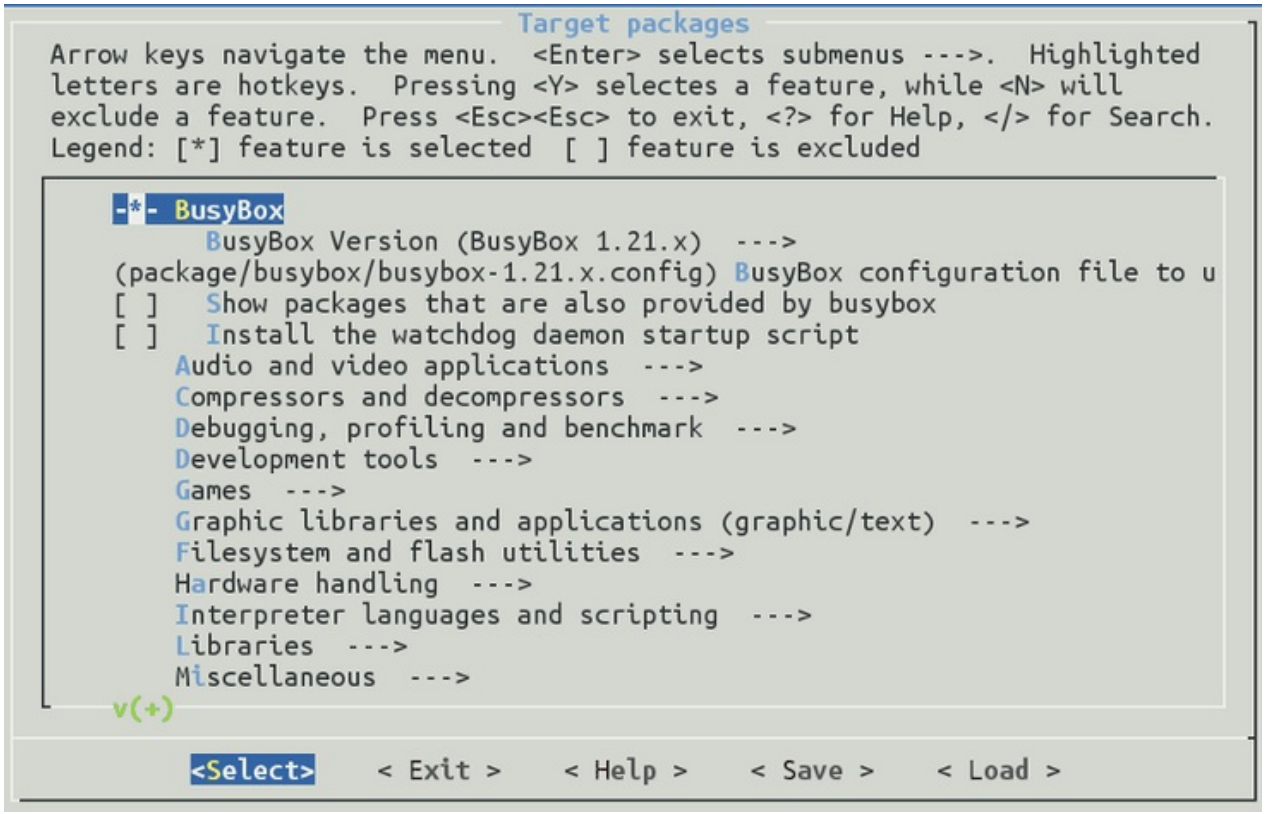
/dev management

- Önemli seçimlerden biri, **/dev** aygıt dosya sisteminin nasıl yönetileceğinin belirlenmesidir
- Geçmiş yıllarda statik /dev yönetimi genel olarak kullanılmaktaydı. Halen daha bu seçeneğin anlamlı olduğu projeler olabilir
- Ancak */dev* dizinini *kernel* üzerinden gelen *event*'ler ile dinamik olarak oluşturan çözümlerin tercih edilmesini önermekteyiz
- Bu amaçla *userspace*'de çalışan, **udev** veya **mdev** yardımcı araçlarını kullanabileceğiniz gibi, *kernel* seviyesinde çalışan **devtmpfs** yöntemini de kullanabilirsiniz
- Özellikle sistemin açılış hızına olumlu etkisi düşünüldüğünde, *udev* veya *mdev* yerine, sadece *devtmpfs* yöntemini kullanmanızı öneririz
- Bu kullanımı şekli için kernel derleme sürecinde, *CONFIG_DEVTMPFS* ve *CONFIG_DEVTMPFS_MOUNT* seçenekleri seçilmiş olmalıdır

Kernel

- Buildroot üzerinden *kernel* derlemek mümkündür
- Bunun için Kernel başlığı altında aşağıdaki seçenekler ayarlanmalıdır:
  - derlenecek version
  - hangi yöntemle download edileceği
  - varsa uygulanacak *patch*'lerin bulunduğu dizin
  - derleme işleminde kullanılacak kernel konfigürasyon dosyası
  - kernel binary formatı
- Bununla birlikte, bir çok board için standart kernel yerine üretici tarafından gelen özelleştirilmiş kernel kullanıldığından, derlemeyi buildroot altından yapmak çok fazla ek iş yükü gerektirebilir. Bu durumlarda kernel derleme sürecini buildroot dışında halletmek anlamlı olacaktır.

Paket Seçimi : Target Packages



Paket Seçimi

- Sistemde yer alacak tüm paket seçimleri bu bölüm ve alt menülerinden gerçekleştirilmektedir

- Paketler gruplanmış olarak, çeşitli başlıklar altına yer alır
- Birbirine bağımlı olan paketler, **Kconfig** yapısı aracılığıyla bağımlılık ilişkileri doğrultusunda listelenir
- **Busybox** paketi sistemde temel bir önem arzettiğinden, hangi versiyonunun derleneceği ve derleme sırasında kullanılacak busybox konfigürasyon dosyası da seçenek olarak sunulmaktadır

Filesystem images

- Derleme işlemleri bitiminde üretilecek olan dosya sistemi imajı bu bölümden belirlenmektedir
- **squashfs**, **cramfs** gibi read-only compressed dosya sistemi imajlarından, **jffs2**, **ubifs** gibi *NAND* flash aygıtlar için uygun dosya sistemlerine kadar geniş bir seçenek kümesi yer almaktadır
- Seçilen dosya sistemi imajı türüne göre açılan ek menülerden parametre geçirmek de mümkündür. Özellikle *ubifs* dosya sistemi için aşağıdaki parametrelerin doğruluğu teyit edilmelidir:
  - UBI logical eraseblock size
  - UBI minimum IO size
  - Maximum LEB count
  - UBI physical eraseblock size
  - UBI sub-page size

Bootloaders

- Buildroot üzerinden aşağıdaki bootloader uygulamarını mimariniz için derlemeniz mümkündür:
  - U-Boot
  - Barebox
  - X-Loader
  - Freescale mxs
  - grub, syslinux

- Gömülü sistemler için yoğun biçimde U-Boot kullanılmakta olsa da, kod kalitesi ve build karmaşıklığı nedeniyle gelecekte yerini yeni nesil u-boot olarak adlandırabileceğimiz **barebox**'a bırakacağı düşünülmektedir

Derleme İşlemi - Hazırlık

- Konfgürasyon işlemi tamamlandıktan sonra `make` komutu ile derleme işlemi başlatılır
- Öncelikle halihazırda mevcut olmayan gerekli dizinler yaratılır (target, stamps, host vb.)
- Hedef dosya sistemi iskeleti, *target* dizini altına kopyalanır
- Bu bölümde kopyalanacak olan iskelet dosya sistemi, konfigürasyon sırasında değiştirilebilir

Derleme İşlemi - Toolchain

- Dizinler hazırlandıktan sonra toolchain seçimine göre aşağıdaki yöntemlerden biri izlenir:
  - *Buildroot toolchain* seçimi yapılmış ise, toolchain üretimi gerçekleştirileceğinden *binutils*, *glibc*, *gcc* gibi çeşitli toolchain bileşenleri download edilir ve toolchain üretilir
  - *External toolchain* seçimi yapılmış fakat belirli bir versiyonun internet üzerinden download edilmesi istenmişse download işlemi gerçekleştirilir
- Her iki yöntem sonrasında, üretilen, download edilen veya zaten sistemde belirli bir dizinde mevcut olan toolchain, *host/usr/toolchain-type/sysroot* dizini altına kopyalanır

Paketler

Dosya Sistemi Üretimi

Buildroot Paketi Oluşturma

Yardımcı Adresler

- Web: http://buildroot.org

- Eposta Listesi: buildroot@uclibc.org (http://buildroot.org/lists.html)
- IRC kanalı: Freenode üzerinde **#buildroot**
- Hata takip sistemi: https://bugs.uclibc.org

Android Platformu

Bu bölümde **AOSP** (Android Open Source Project) projesi kapsamında üretilmiş olan yazılım bileşenlerinin kullanılarak gömülü Linux sistemimiz için özelleştirilmiş bir Android imajı oluşturulması ve Android geliştirme sürecinin temel bileşenlerinin incelemesi yapılacaktır.

Geliştirme Ortamının Hazırlanması

Gerekli Bileşenler

Geliştirme ortamı Google tarafından Ubuntu LTS 12.04 versiyonu için test edilmiş olup, hiç bir sorunla uğraşmak zorunda kalmak istemiyorsanız bu dağıtımın kullanılması önerilir. Biz burada Debian Wheezy ve Debian Jessie dağıtımlarını kullanıyor olacağız.

> Ubuntu tabanlı sistemleri Google sadece önermiyor, aynı zamanda kendileri de Android geliştirme ve build işlemlerinde kullanıyorlar.

Diğer gereklilikler şu şekilde sıralanmaktadır:

- 4 GB Bellek
- Minimum 100 GB boş alan
- Python versiyon 2.6 veya 2.7 versiyonu
- GNU make 3.81 veya 3.82 versiyonu
- JDK 7
- Git versiyon 1.7 veya yukarısı

JDK 7 versiyonunu aşağıdaki gibi kurup:

```
$ sudo apt-get install openjdk-7-jdk
```

sonrasında sisteminizde birden fazla JDK versiyonu bulunuyor ise öntanımlı olacak olanı versiyon **7** olacak şekilde aşağıdaki komutlarla güncelleyin:

```
$ sudo update-alternatives --config java
$ sudo update-alternatives --config javac
```

Sonrasında aşağıdaki paketleri yüklemelisiniz:

```
$ sudo apt-get install git gnupg flex bison gperf build-essential \
    zip curl libc6-dev libncurses5-dev:i386 x11proto-core-dev \
    libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 \
    libgl1-mesa-dev g++-multilib mingw32 tofrodos \
    python-markdown libxml2-utils xsltproc zlib1g-dev:i386
```

> Not: Debian dağıtımında 64 bitlik sisteminizde bu paketleri kurabilmek için **Multi-Arch** desteğini ayarlamış olmanız gerekir. Özet olarak `sudo dpkg --add-architecture i386` ve ardından `sudo apt-get update` komutlarını vermeniz gereklidir. Ayrıntılar için: https://wiki.debian.org/Multiarch/HOWTO

Ubuntu 12.04 için kurulum sonrası aşağıdaki komutla `libGL` kütüphanesi için sembolik link oluşturuyoruz:

```
$ sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-gnu/libGL.so
```

Ubuntu 14.04 kullanıyorsanız aşağıdaki ek paketleri de kurmanız gerekmektedir:

```
$ sudo apt-get install bison g++-multilib git gperf libxml2-utils
```

USB Erişimlerinin Ayarlanması

Modern Linux dağıtımlarında USB veriyolundaki aygıtlar üzerinde normal kullanıcınızla doğrudan erişiminiz olmadığından, sürekli **root** kullanıcı haklarıyla çalışmak zorunda kalmamak için, USB veriyoluna takılan ilgilendiğimiz aygıt tipleri için oluşturulacak aygıt dosyalarının sahibini kendi kullanıcımız olacak şekilde değiştirebiliriz.

Bunun için USB hotplug sürecini yöneten **udev** servisine, kullanacağımız cihazların **Vendor ID** ve **Product ID** bilgileri doğrultusunda aşağıdaki gibi bir ek kural dosyası tanımlamalıyız. Dosya içerisinde ilgilenmediğiniz aygıtlarla ilgili satırları kaldırabilirsiniz.

Aşağıdaki örnek dosyayı sisteminizde `/etc/udev/rules.d/51-android.rules` adıyla oluşturup, dosya içerisindeki `<username>` şeklinde olan kısımları kendi kullanıcı adınızla değiştirmelisiniz.

```
# adb protocol on passion (Nexus One)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e12", MODE="0600", OWNER=
"<username>"
# fastboot protocol on passion (Nexus One)
SUBSYSTEM=="usb", ATTR{idVendor}=="0bb4", ATTR{idProduct}=="0fff", MODE="0600", OWNER=
"<username>"
# adb protocol on crespo/crespo4g (Nexus S)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e22", MODE="0600", OWNER=
"<username>"
# fastboot protocol on crespo/crespo4g (Nexus S)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e20", MODE="0600", OWNER=
"<username>"
# adb protocol on stingray/wingray (Xoom)
SUBSYSTEM=="usb", ATTR{idVendor}=="22b8", ATTR{idProduct}=="70a9", MODE="0600", OWNER=
"<username>"
# fastboot protocol on stingray/wingray (Xoom)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="708c", MODE="0600", OWNER=
"<username>"
# adb protocol on maguro/toro (Galaxy Nexus)
SUBSYSTEM=="usb", ATTR{idVendor}=="04e8", ATTR{idProduct}=="6860", MODE="0600", OWNER=
"<username>"
# fastboot protocol on maguro/toro (Galaxy Nexus)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e30", MODE="0600", OWNER=
"<username>"
# adb protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d101", MODE="0600", OWNER=
"<username>"
# adb protocol on panda (PandaBoard ES)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="d002", MODE="0600", OWNER=
"<username>"
# fastboot protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d022", MODE="0600", OWNER=
"<username>"
# usbboot protocol on panda (PandaBoard)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d00f", MODE="0600", OWNER=
"<username>"
# usbboot protocol on panda (PandaBoard ES)
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d010", MODE="0600", OWNER=
"<username>"
# adb protocol on grouper/tilapia (Nexus 7)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e42", MODE="0600", OWNER=
"<username>"
# fastboot protocol on grouper/tilapia (Nexus 7)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e40", MODE="0600", OWNER=
"<username>"
# adb protocol on manta (Nexus 10)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4ee2", MODE="0600", OWNER=
"<username>"
# fastboot protocol on manta (Nexus 10)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4ee0", MODE="0600", OWNER=
"<username>"
```

Kullanacağınız cihaza ait kural burada yer almıyorsa, USB kablosunu bilgisayarınıza taktıktan sonra, `lsusb` komutunu çalıştırın. Komut çıktısında yeni taktığınız cihazı tahmin edip, `8087:0024` formatındaki gibi **Vendor ID : Product ID** kısımlarını not ederek, *udev* kurallarına bu aygıtı da eklemelisiniz.

Repo Aracının Kurulumu

Android geliştirme ortamında yapılacak işlemleri kolaylaştırmak adına `repo` adlı bir araç kullanılmaktadır. Bu aracı kendi ev dizininizde `bin` alt dizinine kurmanız öneriliyor. `PATH` 'inizde bulunan herhangi bir başka dizine de kurabilirsiniz.

```
$ mkdir -p ~/bin
$ export PATH=~/bin:$PATH
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

> Not: Bazı dağıtımlarda `~/bin` dizini kullanıcıların öntanımlı `PATH` 'inde bulunuyorken bazılarında bulunmamaktadır. Böyle bir durumda kolaylık olması için yukarıdaki `export` komutunu, `~/.bashrc` dosyanızın sonuna ekleyebilirsiniz.

`repo` aracını kurduktan sonra, çalışmalarımızı yapacağımız ana dizini belirleyip oluşturalım, sonrasında ilgili dizine geçelim. Örnek olarak tüm çalışmaları `~/embedded/android` dizini altında yapacaksak:

```
$ mkdir -p ~/embedded/android
$ cd ~/embedded/android
```

Ardından `repo` uygulaması ile `master` branch'i aşağıdaki komut ile ilklendiriyoruz:

```
$ repo init -u https://android.googlesource.com/platform/manifest
```

Komutun bitmesine yakın size isim ve e-posta adresi sorulacaktır. Bu esnada e-posta adresi olarak bir **Google** hesabı girilmelidir:

```
 ...
 * [new tag]         studio_1.0.0 -> studio_1.0.0
 * [new tag]         studio_1.0.1 -> studio_1.0.1

Your Name  [Murat Demirten]:
Your Email [demirten@debian]: mdemirten@yh.com.tr

Your identity is: Murat Demirten <mdemirten@yh.com.tr>
is this correct [y/N]? y

Testing colorized output (for 'repo diff', 'repo status'):
  black    red      green    yellow   blue     magenta   cyan     white
  bold     dim      ul       reverse
Enable color display in this user account (y/N)? y

repo has been initialized in /home/demirten/embedded/android
```

Android Kaynak Kodlarının İndirilmesi

Yukarıdaki işlemleri yaptıktan sonra, `repo` aracının `sync` komutuyla kaynak kodlarının indirilmesi sağlanır:

```
$ repo sync
```

Bu işlem bir miktar uzun sürebilir (bazen çok uzun, belki 1 gün kadar). Ocak 2015 itibariyle yaklaşık toplamı **20 GB**'ı bulan kaynak dosyalar indirilecektir.

Kullanıcı Otorizasyonu

Google repo'larında IP adresi bazında yapılacak isteklere yönelik bir kısıtlama mevcuttur. Tek kişi çalışıyorken bu bir sıkıntı oluşturmasa da, aynı **public** IP adresinin paylaşıldığı ve birden fazla kişinin bu repo'lara eriştiği durumlarda sorun yaşayabilirsiniz. Ortak IP kullanıyor olmaktan kaynaklanacak sorunları bertaraf edebilmek için, https://android.googlesource.com/new-password adresindeki yönergeleri izleyerek kendinize ait bir parola oluşturabilir ve aşağıdaki komutla repo init işlemini tekrar edebilirsiniz. Bu işlem sonrasında yaptığınız istekler IP bazlı kontrol edilmek yerine sizin kullanıcınıza göre kontrol edilecektir.

```
$ repo init -u https://android.googlesource.com/a/platform/manifest
```

Build Sistemi

Android build sistemi, açık kaynak dünyasında kendisinden önceki projelerden önemli farklılıklar içermektedir. İyi ve kötü yanlarını bir kenara bırakıp bu sistemin genel olarak öğrenilmesinde fayda vardır.

Build süreci geleneksel `make` uygulaması ile çalışır. Bu yönüyle bir benzerlik taşıdığı düşünülebilirse de, esasen tek benzerlik bundan ibaret olup, make sistematiği alışılagelmişin dışında kullanılır.

Açık kaynak dünyasında bu denli büyük projelerde genellikle en üst dizinde bir ana Makefile bulunur. Build sürecinin bileşenlerinin seçimi ve diğer özelleştirmeler, ana dizinde yer alan bir konfigürasyon dosyasında tutulur (`.config`). Konfigürasyon dosyasının üretimi görece uzun ve detaylı bir işlem olduğundan, bu süreci kolaylaştıracak çeşitli araçlar sunulur (menuconfig, xconfig, sistem içi yardım vb.). Ana dizindeki Makefile, *.config* dosyasını işler, alt dizinleri *recursive* olarak dolaşır ve yapılan seçimler doğrultusunda alt dizinlerde yer alan *Makefile* dosyalarını da *make* ile işler.

Android build sisteminde ise ana dizinde genel bir *Makefile* ve konfigürasyon dosyası bulunmaz. Alt dizinler dolaşılıp her birindeki *Makefile* dosyaları üzerinden *make* de işletilmeye çalışılmaz. Bunun yerine, dizin ve alt dizinlerde, `Android.mk` dosyası aranır. Bu make uygulaması için hazırlanmış bir Makefile dosyasıdır. Eğer bir dizinde Android.mk bulunur ise make ile işlenir ve bu noktadan sonra, dosyanın bulunduğu dizinin alt dizinlerinde arama işlemi devam etmez. Yani Android.mk bulunduğunda, daha derinlerdeki dizinlere inilmez. Bir üst seviyedeki dizinlerin aranmasına ve oralarda bulunan Android.mk dosyalarının aynı şekilde işlenmesine devam edilir.

Konfigürasyon

Build işlemine başlarken öncelikle `envsetup.sh` betiği çalışılan kabuk içerisinde işletilir:

```
$ source build/envsetup.sh
```

Daha sonra `lunch` komutuyla üretilecek hedef imaja ait parametreler belirtilir. Girilecek parametrenin formatı `BUILD-BUILDTYPE` şeklinde olup aşağıdaki tabloya göre belirlenebilir:

Build İsmi	Cihaz	Açıklama
aosp_arm	ARM emülator	Tüm uygulamalar, diller ve giriş yöntemleri aktif bir imaj
aosp_maguro	maguro	Galaxy Nexus GSM/HSPA+ ("maguro") imajı
aosp_panda	panda	PandaBoard üzerinde çalışacak imaj

Build tipleri ise aşağıdakilerden biri olabilir:

Build Tipi	Kullanım Amacı
user	Nihai ürün kullanımına uygun, kısıtlı erişim
userdebug	Root erişimin ve debug imkanın eklendiği "user" tipi
eng	Ek debug araçlarıyla zenginleştirilmiş geliştiricilere yönelik

Şimdi geliştiricilere yönelik ARM emülatorü üzerinde çalışacak imajımızı üretmeye başlayalım:

```
$ lunch aosp_arm-eng
```

Yukarıdaki komutun çalışması bittiğinde, ayarlanan değişkenlerle ilgili aşağıdaki gibi bir çıktı verilecektir:

```
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=5.0.50.50.50.50
TARGET_PRODUCT=aosp_arm
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
TARGET_CPU_VARIANT=generic
TARGET_2ND_ARCH=
TARGET_2ND_ARCH_VARIANT=
TARGET_2ND_CPU_VARIANT=
HOST_ARCH=x86_64
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.16.0-4-amd64-x86_64-with-debian-jessie-sid
HOST_BUILD_TYPE=release
BUILD_ID=AOSP
OUT_DIR=out
```

> Not: Herhangi bir parametre verilmediğinde `lunch` uygulaması seçim yapabilmemiz için bizi yönlendirecektir.

Derleme

Derleme süreci oldukça fazla zaman alacaktır. Bu nedenle `ccache` gibi bir *compiler cache* sistemi kullanılmasında fayda vardır.

Bunun için `USE_CCACHE` ortam değişkeninin değerini **1** olarak atamalıyız:

```
$ export USE_CCACHE=1
```

Bu işlemin kalıcı olması için `~/.bashrc` dosyasının sonuna yazabilirsiniz.

> Not: ccache öntanımlı olarak `~/.ccache` dizinini kullanacaktır. Farklı bir dizin kullanmak istiyorsanız (ortak bir dizin de olabilir) `CCACHE_DIR` ortam değişkenini de benzer şekilde tanımlamanız yeterlidir.

Sonrasında bir defaya mahsus, `ccache` için kullanılacak maksimum disk alanını limitleyebiliriz:

```
$ prebuilts/misc/linux-x86/ccache/ccache -M 50G
  Set cache size limit to 50.0 Gbytes
```

Derleme sürecinde GNU make'in `-j` parametresiyle paralel derleme yeteneklerinden mutlaka faydalanılmalıdır. Genel bir kural olarak CPU core sayısının 2 katı olacak şekilde bir değer verilmesi önerilir. Örneğin 4 core'lu bir sistemde, `-j 8` parametresinin kullanılması uygun olacaktır.

Derleme işlemi, standart make komutuyla aşağıdaki gibi başlatılır:

```
$ make -j8
```

Sistem Çağrıları

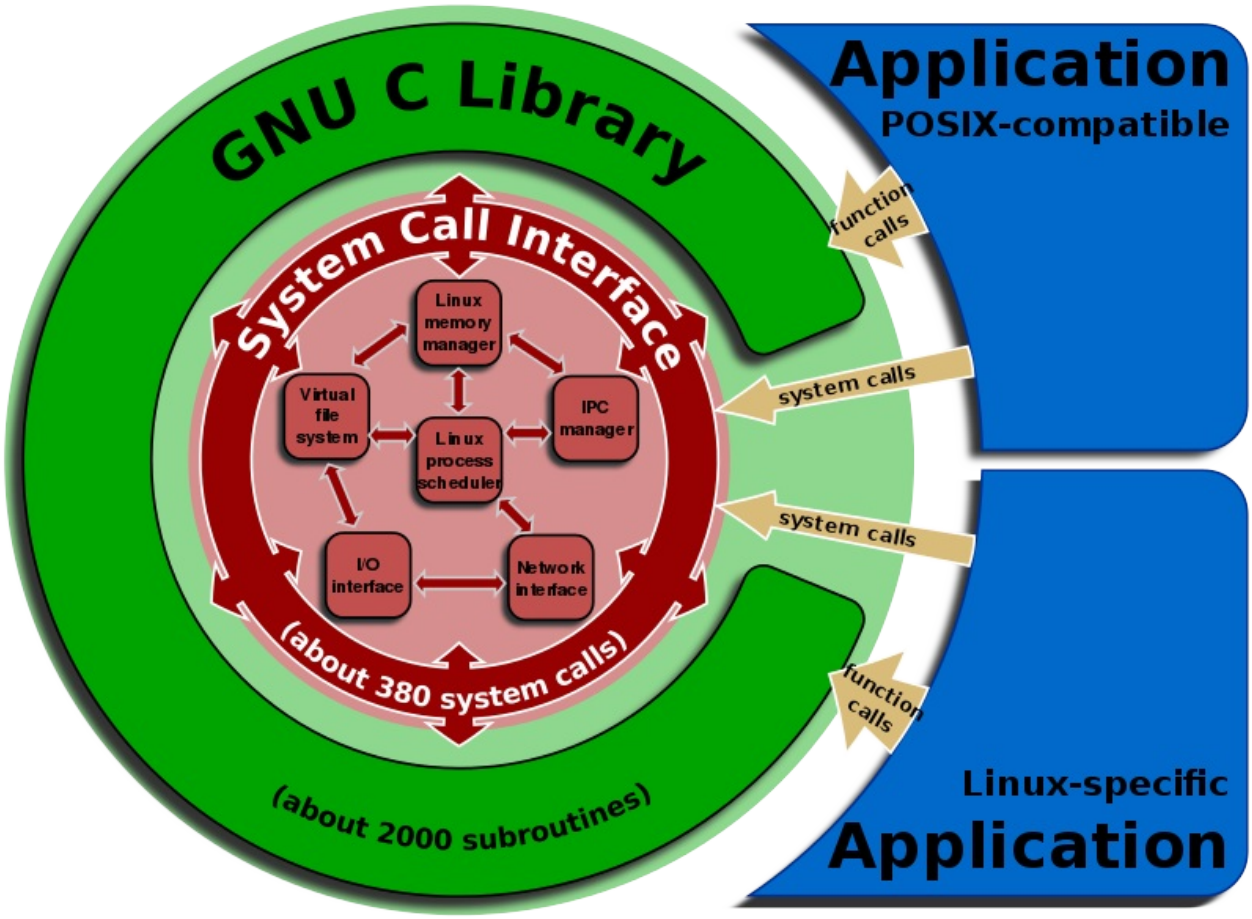
Modern işletim sistemlerinde, çekirdek kipinde çalışma ve kullanıcı kipinde çalışma modları, sert bariyerlerle birbirinden ayrılmış durumdadır.

Bu şekildeki bir tasarım, sistemin sağlıklı çalışması için elzemdir.

Kullanıcı kipinde çalışan bir uygulamanın, sistem çağrıları aracılığıyla işletim sistemi çekirdeğinden ihtiyaç duyduğu servisleri alabilmesi sağlanır.

Bununla birlikte, sistem çağrıları normal fonksiyon çağrılarına oranla oldukça yüksek maliyetli işlemlerdir.

Her sistem çağrısında uygulamanın o anki durumunun saklanması, çekirdeğin işlemcinin kontrolünü ele alması ve ilgili sistem çağrısı ile çekirdek kipinde talep edilen işlemleri gerçekleştirmesi, sonra ilgili uygulamanın tekrar çalışma sırası geldiğinde, uygulamanın saklanan durumunun yeniden üretilip işlemlerin kaldığı yerden devamının sağlanması gereklidir.



Sistem Çağrısı Nasıl Gerçekleşir?

Sistem çağrılarının kernel tarafındaki gerçekleştirimi mimariden mimariye değişkenlik gösterir. Linux çekirdeğinin farklı bir mimariye port edilirken yapılan temel işlem adımlarından biri, sistem çağrılarının en verimli şekilde yapacak şekilde uygun bir kodlamanın mimari spesifik olarak yapılmasıdır.

Her sistem çağrısının 1, 5, 27 gibi ilişkili bir numarası vardır. Bu numaralar da mimariden mimariye değişkenlik göstermektedir. Temel sistem çağrıları tüm mimarilerde bulunmakla birlikte, tüm mimarilerde eşit sayıda sistem çağrısı bulunmaz.

> Konunun devamında aksi belirtilmedikçe verilen örnekler 32 bit Intel mimarisi için geçerlidir.

Kullanıcı kipindeyken herhangi bir sistem çağrısı yapıldığında `INT 0x80` makine dili kodu ile *trap* oluşturulur.

Aynı zamanda talep edilen sistem çağrısının numarası, `EAX` yazmacına yazılır.

Talep edilen sistem çağrısının parametreleri var ise, bu parametrelerin diğer yazmaçlar kullanılarak belirtilmesi gerekir. Ancak her mimaride bu amaçla kullanılabilecek yazmaç sayısı limitlidir. Bazılarında daha çok genel amaçlı yazmaç var iken bazılarında daha az olduğu görülmektedir.

32 bitlik Intel platformu için Linux çekirdek versiyonu **2.3.31** ve sonrası, maksimum **6** sistem çağrısı parametresini desteklemektedir. Bu parametreler sırasıyla `EBX`, `ECX`, `EDX`, `ESI`, `EDI` ve `EBP` yazmaçlarında saklanır.

Sistem çağrısı için 6'dan fazla parametre gerekli olduğunda, bellekteki bir veri yapısı hazırlanarak parametrelar burada saklanır, sonrasında ilgili bellek adresi sistem çağrısına parametre olarak geçirilir.

Mimari Bağımlılığı

Sistem çağrılarının doğrudan işlemci mimarisine bağımlı olduğuna değinmiştik.

Örnek olarak Intel 32 bitlik işlemcilerde `INT 0x80` ile *trap* oluşturulurken, **ARM** mimarisinde aynı işlem **supervisor call** `SVC` ile yapılır

Benzer şekilde Intel mimarisinde `EAX` yazmacına yazılan sistem çağrısının numarası, ARM mimarisinde `R8` yazmacına koyulur.

ARM mimarisinde sistem çağrısına ait 4 adede kadar parametre, `R9`, `R10`, `R11` ve `R12` yazmaçlarına aktarılır. 4 adetten fazla parametre geçilmesi gerektiğinde, bellek üzerinde veri yapısı hazırlanarak bu bölümün adresi geçirilir.

Genel olarak sistem çağrıları performansının *ARM* mimarisinde *x86*'ya göre daha düşük olduğunu söyleyebiliriz (yazmaç/register sayısının azlığı bunda etken olabilir mi düşününüz).

Sistem Çağrısı Nasıl Yapılır?

Sistem çağrılarını daha zor bir yoldan doğrudan yapmak mümkün olsa da bu önerilen bir durum değildir.

Sistem çağrıları, `glibc` kütüphanesindeki *wrapper* fonksiyonlar üzerinden kullanılır.

`glibc` kütüphanesi, üzerinde çalıştığı çekirdek versiyonuna göre, hangi Linux sistem çağrısını yapacağını belirler.

Bazı durumlarda ise bundan daha fazlasını yaparak, üzerinde çalışılan çekirdek versiyonunda hiç desteklenmeyen bir özelliği de sunuyor olabilir. Örnek olarak, Linux **2.6** versiyonuyla birlikte gelen *POSIX Timer API*'nin olmadığı Linux **2.4** versiyonu üzerinde çalışan ve aynı anda pek çok *timer* kullanan bir uygulamanız var ise, *glibc* çekirdek tarafından alamadığı desteği kullanıcı kipinde her *timer* için bir *thread* açarak sağlar. Elbette timer sayınız fazla ise bu çok yavaş bir çözüm olur ancak uygulamanın çalışmasını da mümkün kılar.

Sistem Çağrıları → Glibc Fonksiyonları İlişkisi

Pek çok sistem çağrısı, aynı isimdeki `glibc` *wrapper* fonksiyonları üzerinden çağrılmaktadır.

> Not: Bu duruma `strace` çıktılarını okurken de dikkat etmemiz gereklidir.

Örnek olarak strace çıktısındaki `open()` çağrısına bakalım:

```
open("/tmp/index.jpeg", O_RDONLY)       = 3
```

Burada kastedilen `glibc` içerisindeki `open()` fonksiyonu değil, `open` sistem çağrısıdır.

Strace üzerinden sistem çağrısına geçirilen argümanları ve geri dönüş değerini (3) görmekteyiz.

Şaşırtıcı Bir Durum: Performans

Genel olarak bu dahil pek çok dokümanda sistem çağrılarının çok yavaş olduğunu okuyabilirsiniz. Her sistem çağrısında kullanıcı kipinden kernel kipine geçiş ve *context switch* önemli bir yük getirir. Dolayısıyla bu süreç ne kadar verimli bir şekilde yönetilebilirse genel sistem performansı da aynı şekilde doğrudan etkilenecektir.

2002 yılında Linux Kernel eposta listelerine *Mike Hayward*'ın şaşkınlığını içeren bir eposta düştü. *Hayward* elindeki **Pentium 3 - 850 Mhz** dizüstü bilgisayarıyla **Pentium 4 - 2 Ghz** ve **Xeon - 2.4 Ghz** sistemlerinin, sistem çağrıları açısından performansını ölçmek için bir test uygulaması yazdı ve 1K'lık buffered dosya okuma testinde aşağıdaki şaşırtıcı sonuçları elde etti:

Sistem	Saniyedeki IO
Pentium 3 - 850 Mhz	149
Pentium 4 - 2 Ghz	108
Xeon - 2.4 Ghz	69

Aynı testi dosya okuma yerine farklı sistem çağrılarıyla da test ettiğinde benzer sonuçların alındığını tespi etti.

Bunun sebebi, bazı **x86** serisi işlemcilerde çekirdek kipine daha hızlı geçiş için `SYSENTER/SYSEXIT` özel instruction'ının bulunmasıydı. Pentium 3 serisinde varolan bu destek, Pentium 4 ve Xeon işlemcilere yeterince olgunlaşmadığından konulmamıştı. Pentium 3'teki bu imkanı iyi değerlendiren Linux çekirdeği, kendisinden daha üstün Xeon işlemcilerden bile daha iyi performans göstermekteydi.

Benzer zamanlarda **AMD** de benzer şekilde `SYSCALL/SYSRET` özel instruction'ınını sunmaya başlamıştır.

Linux çekirdeği de bu yeni imkanları kullanarak geleneksel `INT 0x80` kesme yöntemine göre önemli oranda performans iyileşmesi sağlanmıştır.

Günümüz **x86** ve **x86_64** işlemcilerinde bu mekanizma tümüyle desteklenmektedir.

Performans Problemi - Detaylı Bakış

Özellikle **x86** tabanlı mimarilerde `SYSENTER` özel yolu sayesinde sistem çağrılarının hızlanmasını sağladık. Ancak bu yeterli olacak mıdır?

Bir çok uygulamada, özellikle `gettimeofday()` gibi sistem çağrılarının çok sık kullanıldığını görürürüz.

Uygulamalarınızı `strace` ile incelediğinizde, bilginiz dahilinde olmayan pek çok farklı `gettimeofday()` çağrısını yapıldığını görebilirsiniz.

`glibc` kütüphanesinden kullandığınız bazı fonksiyonlar, *internal* olarak bu fonksiyonaliteyi kullanıyor olabilir.

Java Virtual Machine gibi bir **VM** üzerinde çalışan uygulamalar için de benzer bir durum söz konusudur.

Görece basit bir işlem olmasına rağmen sık kullanılan bu operasyon yüzünden sistemlerde nasıl bir sistem çağrısı yükü oluşmaktadır? sorusunu kendimize sorabiliriz.

Linux Dynamic Linker/Loader: `ld.so`

Paylaşımlı kütüphaneler kullanan uygulamaların çalıştırılması sırasında, **Linux Loader** tarafından gereken kütüphaneler yüklenerek uygulamanın çalışacağı ortam hazırlanır. En basit **Hello World** uygulamamız bile `libc` kütüphanesine bağımlı olacaktır.

`ldd` ile uygulamanın linklenmiş olduğu kütüphanelerin listesini alabiliriz:

```
$ ldd hello
    linux-vdso.so.1 (0x00007fff7d88a000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2fb43a0000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f2fb476d000)
```

Görüldüğü üzere `libc` ve `ld.so` bağımlılıkları listelendi. Fakat `linux-vdso.so.1` kütüphanesi nedir?

`find` komutu ile tüm sistemimizi arattığımızda neden bu kütüphaneyi bulamıyoruz?

Virtual DSO: `linux-vdso.so.1`

linux-vdso.so.1 sanal bir **D**namically Linked **S**hared **O**bject dosyasıdır. Gerçekte böyle bir kütüphane dosya sistemi üzerinde yer almaz. Linux çekirdeği, çok sık kullanılan bazı sistem çağrılarını, bu şekilde bir hile kullanarak kullanıcı kipinde daha hızlı gerçekleştirmektedir.

Örnek olarak, sistem saati her değişiminde sonucu tüm çalışan uygulamaların adres haritalarına da eklenmiş olan özel bir bellek alanına koyarsa, `gettimeofday()` işlemi gerçekte bir sistem çağrısına yol açmadan kullanıcı kipinde tamamlanabilir.

Şimdi bu konuları biraz daha detaylandıralım.

> Not: Konunun bundan sonrası meraklıları için olup, çok gerekli olmayan bu bölümün yeni başlayan kullanıcılar için atlanması önerilir.

`/proc/self/maps`

Linux `proc` dosya sisteminde `/proc/<PID>/maps` dosyasında ilgili `PID` (process id) için çekirdek tarafından yapılmış olan adres haritalaması gösterilir.

Özel bir durum olarak, `<PID>` yerine `self` ibaresi kullanıldığında, o an bu dosya erişimini yapan process ile ilgili dizinde işlem yapılmış olur.

```
$ cat /proc/self/maps
    00400000-0040c000 r-xp 00000000 08:02 1703938   /bin/cat
    0060b000-0060c000 r--p 0000b000 08:02 1703938   /bin/cat
    0060c000-0060d000 rw-p 0000c000 08:02 1703938   /bin/cat
    024de000-024ff000 rw-p 00000000 00:00 0         [heap]
    7ff7033c5000-7ff70369f000 r--p 00000000 08:02 2362900 /usr/lib/locale/locale-archi
ve
    7ff70369f000-7ff70383e000 r-xp 00000000 08:02 393230  /lib/x86_64-linux-gnu/libc-2
.19.so
    7ff70383e000-7ff703a3d000 ---p 0019f000 08:02 393230  /lib/x86_64-linux-gnu/libc-2
.19.so
    7ff703c69000-7ff703c6a000 rw-p 00000000 00:00 0
    7fff8cd95000-7fff8cdb6000 rw-p 00000000 00:00 0 [stack]
    7fff8cdfc000-7fff8cdfe000 r-xp 00000000 00:00 0 [vdso]
    7fff8cdfe000-7fff8ce00000 r--p 00000000 00:00 0 [vvar]
    ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0   [vsyscall]
```

`vsyscall`

`/proc/self/maps` dosyasına `cat` uygulaması ile bir kaç defa baktığınızda, `vsyscall` (Virtual System Call Page) haricindeki bölümlerin başlangıç adreslerinin değiştiğini görmekteyiz.

`vsyscall` bölümü, sadece bir uygulama için değil, sistemdeki tüm uygulamalar için aynı statik yeri göstermektedir.

Bu sayede dinamik linkleme (dolayısıyla paylaşımlı kütüphane) kullanmayan, tamamen statik uygulamaların da bu **statik** adres üzerinden `vsyscall` bölümüne erişimi mümkün olmaktadır.

Bu bölgenin uzunluğu kısıtlı olduğundan, sadece belirli sayıda girdiye sahiptir: *vgettimeofday()*, *vtime()*, *vgetcpu()*

Tüm uygulamalar için aynı adrese haritalanması, özellikle **return to libc** türü ataklarıyla sistem çağrısı yapılabilmesine neden olmaktadır.

Linux **3.0** versiyonuna kadar **vsyscall** tablosu kullanılmış olmakla birlikte, **3.1** ve sonrasında bu yöntem artık önerilmiyor. **vDSO** mekanizması hem daha güvenli hem daha hızlı.

`vdso` Bölümünü Dışarı Çıkartmak

İnceleme amacıyla uygulamanın adres haritasındaki `[vdso]` biçiminde işaretlenmiş alanı diske çıkartmaya çalışalım.

Örneğimizde bu bölümün `7fff8cdfc000` ile `7fff8cdfe000` adresleri arasında, 2 adet `Page` büyüklüğünde olduğunu görüyoruz.

Acaba `dd` komutu ile bu bölümü dışarı çıkartabilir miyiz:

```
$ dd if=/proc/self/mem of=dso.out bs=1 skip=$((0x7fff8cdfc000)) count=8192
```

Maalesef bu yöntem artık çalışmıyor. Bunun 2 nedeni var:

1. `/proc` sanal dosya sistemi altındaki girdiler normal bir dosya gibi görünmesine karşılık, `stat()` ile bakıldığında `st_size` değeri **0** olmaktadır. Bu durum `dd` uygulamasının ilgili offset adresine **seek** yapılamayacağını söylemesine neden oluyor. Çözüm için ufak bir yama gerekiyor
2. Yeni Linux çekirdek versiyonlarında buradaki başlangıç değer adresi, `7fff8cdfc000` her uygulama için aynı değildir. **Return to libc** tarzı atakları zorlaştırmak için bu değer ancak çalışan uygulama içerisinden öğrenilebilir. Bunun için `dd` kodunun değiştirilmesi veya ufak bir test uygulaması yazılması gerekiyor.

`extract_region.c`

Bu işlemi yapabilmek için `extract_region` adını verdiğimiz aşağıdaki gibi bir uygulama hazırlayalım:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define _FILE_OFFSET_BITS 64

int main (int argc, char *argv[])
{
```

```
    if (argc != 3) {
        fprintf(stderr,
            "Kullanım: %s <cikti> <section>\n"
            "\t<cikti>\t\t: export edilecek dosya\n"
            "\t<section>\t: export edilecek map region\n\n", argv[0]);
        return 1;
    }

    off_t start_addr, end_addr;
    char buf[4096];
    const char *out = argv[1];
    const char *region = argv[2];

    int found = 0;
    FILE *fp = fopen("/proc/self/maps", "r");
    while (fgets(buf, sizeof(buf), fp)) {
        printf("%s", buf);
        if (strstr(buf, region)) {
            found = 1;
            break;
        }
    }
    fclose(fp);
    if (!found) {
        fprintf(stderr, "%s bölümü bulunamadı\n", region);
        return 1;
    }
    end_addr = strtoull((strchr(buf, '-') + 1), NULL, 16);
    *(strchr(buf, '-')) = '\0';
    start_addr = strtoull(buf, NULL, 16);

    printf("\nÇıkartılacak Alan Başlangıç: 0x%llx, Bitiş: 0x%llx\n\n", start_addr, end
_addr);

    FILE *dst = fopen(out, "w+");
    if (dst == NULL) {
        fprintf(stderr, "%s açılamadı\n", out);
        return 1;
    }
    FILE *src = fopen("/proc/self/mem", "r");
    char *tmp = malloc(end_addr - start_addr);
    fseeko(src, start_addr, SEEK_SET);
    fread(tmp, end_addr - start_addr, 1, src);
    fwrite(tmp, end_addr - start_addr, 1, dst);
    fclose(src);
    fclose(dst);
    return 0;
}
```

Test Uygulamamızı Çalıştıralım

```
$ ./extract_region vdso.out vdso
...
7ff6db951000-7ff6dbaf0000 r-xp 00000000 08:02 393230    /lib/x86_64-linux-gnu/libc-2.1
9.so
7ff6dbf19000-7ff6dbf1a000 r--p 00020000 08:02 393236    /lib/x86_64-linux-gnu/ld-2.19.
so
7ff6dbf1a000-7ff6dbf1b000 rw-p 00021000 08:02 393236    /lib/x86_64-linux-gnu/ld-2.19.
so
7ff6dbf1b000-7ff6dbf1c000 rw-p 00000000 00:00 0
7fffc477d000-7fffc479e000 rw-p 00000000 00:00 0         [stack]
7fffc47fc000-7fffc47fe000 r-xp 00000000 00:00 0         [vdso]

Çıkartılacak Alan Başlangıç: 0x7fffc47fc000, Bitiş: 0x7fffc47fe000
```

İşlem bitiminde **8192** byte uzunluğunda **vdso.out** dosyası oluşacaktır.

`file` komutu ile dosyanın tipine baktığımızda standart bir kütüphane gibi görünecektir:

```
$ file vdso.out
vdso.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
BuildID
[sha1]=538bea2738a229413dcc98af8f4f7127f9bca874, stripped
```

`vdso` İçine Bakalım

`objdump` ile dışarı çıkarttığımız bu bölüme bir bakalım:

```
$ objdump -T vdso.out

vdso.out:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000418 l    d  .rodata    0000000000000000              .rodata
0000000000000970  w   DF .text    000000000000057d  LINUX_2.6   clock_gettime
0000000000000000 g    DO *ABS*    0000000000000000  LINUX_2.6   LINUX_2.6
00000000000000ef0 g    DF .text    00000000000002b9  LINUX_2.6   __vdso_gettimeofday
00000000000011d0 g    DF .text    000000000000003d  LINUX_2.6   __vdso_getcpu
00000000000000ef0  w   DF .text    00000000000002b9  LINUX_2.6   gettimeofday
00000000000011b0  w   DF .text    0000000000000015  LINUX_2.6   time
00000000000011d0  w   DF .text    000000000000003d  LINUX_2.6   getcpu
0000000000000970 g    DF .text    000000000000057d  LINUX_2.6   __vdso_clock_gettime
00000000000011b0 g    DF .text    0000000000000015  LINUX_2.6   __vdso_time
```

Basit Bir Test Uygulaması

100.000 defa `gettimeofday()` fonksiyonunu çağıran ve işlem bitiminde başlangıç ve bitiş zamanlarını gösteren örnek bir uygulama yapalım:

```
#include <stdio.h>
#include <sys/time.h>

int main ()
{
    int i;
    struct timeval now;
    struct timeval before;
    struct timeval after;
    gettimeofday(&before, NULL);
    for (i = 0; i < 100000; i++) gettimeofday(&now, NULL);
    gettimeofday(&after, NULL);
    printf("Before: %li.%li\n", before.tv_sec, before.tv_usec);
    printf("After : %li.%li\n", after.tv_sec, after.tv_usec);
    return 0;
}
```

`X86_64` ve `ARM` Üzerinde Test

100.000 defa `gettimeofday` çağrısı yapan `time_test` örnek uygulamasını, 1 Ghz hızına düşürülmüş **X86_64** *i5* işlemcili platform ile 1 Ghz saat frekansındaki **ARM** *BeagleBoneBlack* platformunda karşılaştıralım

```
(X86_64) $ ./time_test
Before: 1419786575.719463
After : 1419786575.722560

(ARM) $ ./time_test
Before: 1419786909.186960
After : 1419786909.252160
```

Görüldüğü üzere X86_64'te 3-4 milisaniyede gerçekleşen işlem, ARM sistemimizde 70 milisaniyelerde gerçekleşmektedir.

Şimdi test uygulamamızı bir de `strace` kontrolünde her iki platformda çalıştıralım:

```
(X86_64) $ strace ./time_test
...
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ff4ee1f1
000
write(1, "Before: 1419787456.897162\n", 26Before: 1419787456.897162) = 26
write(1, "After : 1419787456.904669\n", 26After : 1419787456.904669) = 26
exit_group(0)

##########################

(ARM) $ strace ./time_test
...
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x400b3000
gettimeofday({1419787571, 234967}, NULL) = 0
gettimeofday({1419787571, 235064}, NULL) = 0
gettimeofday({1419787571, 235142}, NULL) = 0
gettimeofday({1419787571, 235257}, NULL) = 0
gettimeofday({1419787571, 235394}, NULL) = 0
...
...
write(1, "Before: 1419787571.234967\n", 26) = 26
write(1, "After : 1419787571.976285\n", 26) = 26
exit_group(0) = ?
```

- `X86_64` platformunda `strace` ile yaptığımız incelemede, herhangi bir `gettimeofday()` sistem çağrısı göremedik
- Beklediğimiz şekilde `linux-vdso.so` mekanizması sayesinde, işlem tamamen kullanıcı kipinde gerçekleştirildi, herhangi bir sistem çağrısı yapılmadı
- Sadece `printf()` fonksiyonu nedeniyle `write()` sistem çağrısı kullanılarak son çıktı konsola gönderildi
- `ARM` mimarisindeki örneğimize baktığımızda, benzeri bir mekanizma olmadığı için, her defasında karşılık gelen bir `gettimeofday()` sistem çağrısı yapıldığını görüyoruz (örnek ekran çıktımızda ... olarak belirttiğimiz bölümde 100000 adet benzer çağrı bulunmaktadır)

Sistem Çağrılarının Kesintiye Uğraması

Kendimize şu soruyu soralım: uygulamamız bir sistem çağrısı yaparak çekirdek kipinde kod işletiliyor durumunda iken sinyal (software interrupt) gelirse ne olur?

Bu durumda sistem çağrısı sona erecek ve **EINTERRUPTED** hatası dönecektir.

Sistem çağrılarını `glibc` üzerinden kullandığımız için, glibc tarafında sistem çağrısından `EINTR` hatası geldiğinde, uygulamaya geri dönüş değeri olarak `-1` dönülür fakat `errno`

global değişkeni `EINTR` şeklinde ayarlanır.

Bu aslında hata olmayan istisnai durum, zaman zaman pek çok uygulama kodunda gözardı edilmektedir.

Bazı kullanım senaryolarında yukarıdaki senaryo istisnai olmaktan çıkıp, ilgili yazılımın doğası gereği sürekli veya sıklıkla da (read, write, open, connect vb.) oluşabilir.

Uygulama perspektifinden baktığımızda tüm sistem çağrılarını sarmalayan fonksiyonlar için aşağıdaki kural geçerlidir:

- Eğer bir sistem çağrısının geri dönüş değeri `0` 'dan küçükse ve `errno` değişkeni `EINTR` sabitine eşitse, herhangi bir hata söz konusu değildir.

Bahsedilen senaryo oluştuğunda ilgili fonksiyonun (yani sistem çağrısının) yeniden çağrılması gerekir.

Bu süreç, ilgili sinyallerin oluşturulmasında `SA_RESTART` bayrağının işaretlenmesi suretiyle otomatik hale getirilebilir. Peki neden öntanımlı olarak bu şekilde değil?

Esasen bir zamanlar öyleydi. Ancak sistem çağrısının otomatik olarak yeniden başlatılmasını istemeyeceğimiz durumlar da olabilir. Bu yüzden öntanımlı olarak bir aksiyon alınmıyor.

I2C Protokolü

Bu bölümde, tümleşik devreler (IC) arasında haberleşmede yaygın olarak kullanılan **I2C** (Inter-Integrated Circuit) protokolünü inceleyeceğiz.

Bölüm içerisinde ilk olarak I2C protokolü hakkında bilgi verecek, sonrasında Linux altında I2C protokolünün nasıl ele alındığına bakacağız.

Uygulama kısmında ise örnek board'lar üzerinden sayısal sıcaklık sensörü ile, I2C üzerinden bir sıcaklık değeri okuma örneği yapacağız.

I2C Protokolünün Tanıtılması

I2C (Inter-Integrated Circuit), 1980'li yılların başında, **Philips Semiconductor** tarafından geliştirilmiş bir seri iletişim protokülüdür.

İşlemci ve mikrodenetleyiciler, aynı veri yolu (bus) üzerinden, birden çok çevre birimiyle (peripheral) haberleşebilmelidir. Çevre birimlerine örnek olarak, EEPROM, Analog Sayısal Dönüştürücü (ADC), LCD sürücü ve zamanlayıcı (Real Timer) aygıtlarını gösterebiliriz. Ayrıca başka işlemciler de çevre birimi olabilmektedir.

Aynı veri yolu birden çok birim tarafından kontrollü bir şekilde paylaşılmalı ve iletişim kurulacak çevre birimi, işlemci tarafından, herhangi bir yöntemle seçilebilmelidir. Bu seçim ayrı bir adres yolu üzerinden yapılabilir. Fakat bu durumda işlemcinin, normalde başka amaçlarla kullanabileceği bazı uçların, sadece adresleme için tahsis edilmesi zorunluluğu ortaya çıkacaktır. Ayrıca bu adres yolları sebebiyle, PCB'nin karmaşıklığı da artacaktır.

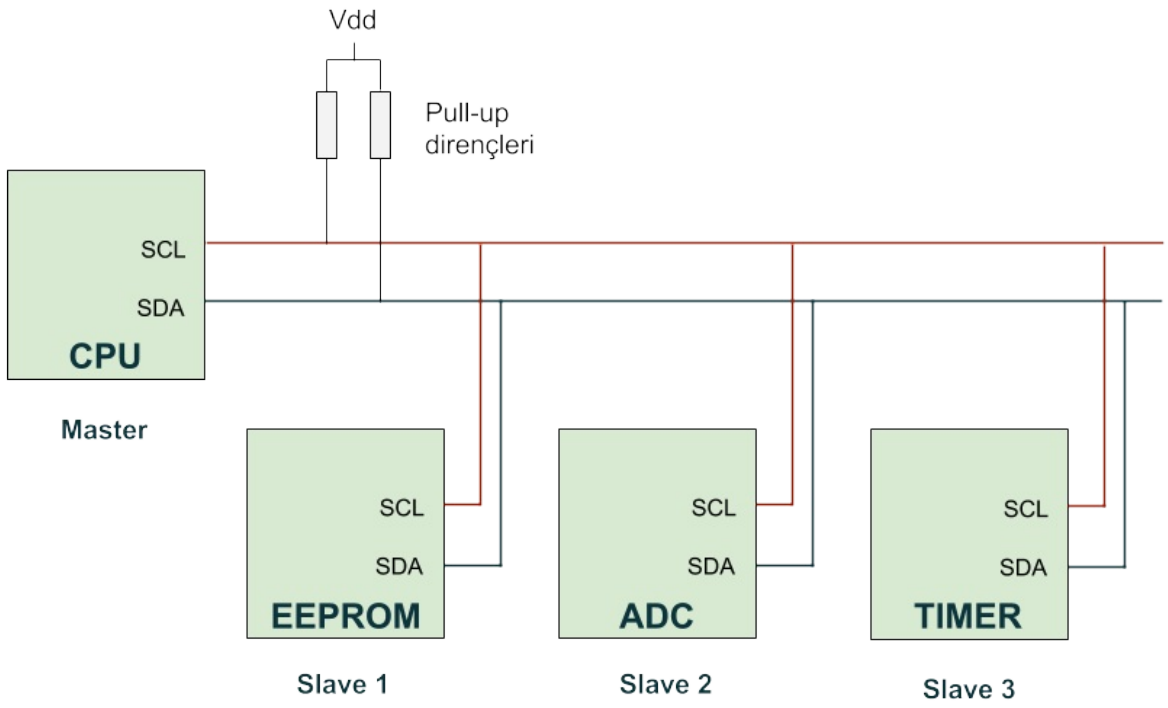
Yukarıda bahsettiğimiz dezavantajları gidermek için Philips, yakın mesafelerde düşük band genişliği ile çalışan, 2 kablolu (2 Wired), I2C protokolünü geliştirmiştir. I2C protokolü, biri clock diğeri de veri olmak üzere 2 adet iletişim kanalına sahiptir. Bu kanallar, **SCL** (Serial Clock) ve **SDA** (Serial Data) olarak isimlendirilmektedir. SDA veri iletişimi için kullanılmakta, SCL ile ise gönderen ve alan taraflar veri senkronizasyonunu sağlamaktadır.

SDA ve SCL veri yolları üzerinde birçok aygıt bulunabilir. Bu aygıtlar, düğümleri (node) oluşturmaktadır. İletişimi başlatan taraf (CPU, mikrodenetleyici) **master**, karşı taraf ise **slave** olarak isimlendirilir. SCL master'ın kontrolündedir ancak slave de, ihtiyaç halinde, SCL'deki elektriksel seviyeyi değiştirebilmektedir.

Slave aygıtlar genel olarak, bir bölümü sabit diğerleri ise programlananabilen **7** bit'lik adreslere sahiptir. Örneğin, ilk 4 bit üretim aşamasında belirlenirken diğer 3 bit ise elektriksel olarak programlanabilir. Bu kullanıma ileride değineceğiz. Master, bu adres üzerinden slave aygıta ulaşabilmektedir. Adresleme, veri transferinde olduğu gibi, SDA üzerinden seri biçimde olmaktadır. I2C, standard **100kHz**, fast **400kHz** ve high speed **3.4MHz** olmak üzere 3 farklı hızı desteklemektedir.

> **Not:** Protokole sonradan yapılan eklentilerle beraber 10bit'lik adreslerin de kullanılabilmesi mümkündür.

Örnek bir I2C bağlantısı aşağıdaki gibidir.



> **Not:** I2C veri yolu üzerindeki birimler **open drain** veya **open collector** özelliği göstermektedir. Bu birimler tarafından, SDA ve SDC üzerindeki elektriksel seviye ancak aşağıya yani lojik 0 seviyesine çekilebilir, lojik 1 yapılamaz. Bu sebeple SDA ve SDC hatları birer pull-up direnciyle lojik 1 seviyesine çekilmelidir. Pull-up direnci olarak 1K ile 10K aralığında dirençler kullanılabilmektedir.

I2C ile ilgili temel özellikleri maddeler halinde aşağıdaki gibi özetleyebiliriz.

- I2C birçok aygıt arasında, ayrı bir adres yoluna ihtiyaç duymaksızın, 2 kablolu bir iletişim kanalı oluşturur. Ayrıca referans olarak ortak bir toprak hattına (ground) ihtiyaç vardır.
- I2C, SDA ve SDC olmak üzere 1 adet veri ve 1 adet clock kanalına sahiptir.
- CPU ve mikrodenetleyici gibi, iletişimi başlatan birim master olarak isimlendirilir ve saat frekansını (clock) üretmekten sorumludur. Ayrıca iletişim yine master tarafından sonlandırılır.
- Slave çevre birimleri genellikle 7 bitlik adreslere sahiptir ve bu birimlere bu adresler üzerinden erişilir. Aynı veriyolu üzerine **112** tane aygıt bulunabilir. Bazı adresler rezerve edilmiş durumdadır bu sebeple 7 bitlik adres alanının tamamı aygıtlar için kullanılamamaktadır. (CBUS adres desteği, Start Byte, General Call, 10 bit adresleme desteği için ayrılmış adresler, farklı bus formatlarını desteklemek için ayrılmış adresler gibi rezerve adresler bulunmaktadır)
- Master ve slave arasındaki iletişim çift yönlüdür (bidirectional). İletişimin yönünü master belirler.
- İletişim yolu üzerinde birden çok master bulunabilir.

- İletişim hızını yani verinin ne zaman gönderileceğini ve okunacağını, SCL üzerinden, master dikte etmektedir. Fakat slave, bu hızda veri gönderemediği veya işleyemediği durumda SCL'nin elektriksel seviyesini değiştirerek (lojik 0 yaparak) iletişimi belli bir süre bloklayabilmektedir. Bu işlem *clock stretching* olarak isimlendirilir.
- I2C ile aynı kart üzerinde olduğu gibi kartlar arasında da iletişim mümkündür, maksimum iletişim kanal uzunluğu yaklaşık 4m'dir.
- I2C veriyolu herhangi bir anda ya iletişim modundandır yada *idle* konumdadır.

I2C İletişim Aşamaları

I2C veri transferi aşağıdaki aşamalardan oluşmaktadır.

- **Hattın boşta olması durumu (Idle):** Herhangi bir veri transferinin gerçekleşmediği durumdur. Bu aşamada SCL ve SDA hatları lojik 1 seviyesindedir.
- **Başlangıç aşaması:** SCL lojik 1 iken, SDA'nın lojik 1'den 0 çekilmesiyle oluşur. Bu işlem master tarafından yapılmaktadır.
- **Adresleme aşaması:** Bu aşamada, master tarafından, iletişime geçilecek slave aygıt adresi ve iletişim modu gönderilmektedir. İletişim modu, master tarafından okuma mı yoksa yazma işlemi mi yapılacağını belirler.
- **Veri iletim aşaması:** Bit düzeyinde veri iletişiminin yapıldığı aşamadır.
- **Sonlandırma aşaması:** SCL lojik 1 iken, SDA'nın lojik 0'dan 1'e çekilmesiyle oluşur. Bu işlem master tarafından yapılmaktadır, başlangıç aşamasının tersi olarak düşünülebilir.
- **Başlangıç aşamasının tekrarlanması (Repeated start):** Bazen, bir iletişim kurulmuş olmasına karşın, iletişim sonlandırılmadan, yeni bir başlangıç durumu oluşturulmakta ve slave yeniden adreslenmektedir. Bu sayede hattın kullanımı kaybedilmemekte ve aygıtlar arasındaki iletişim kesilmeden, birbirini takip eden okuma ve yazma işlemleri (combined read/write) yapılabilmektedir. Ayrıca EEPROM gibi çevre birimlerine yeni komutlar gönderebilmek için başlangıç durumunun yeniden oluşturulması gerekebilmektedir. EEPROM ancak başlangıç sinyalinden sonra bazı komutları anlayabilmektedir.

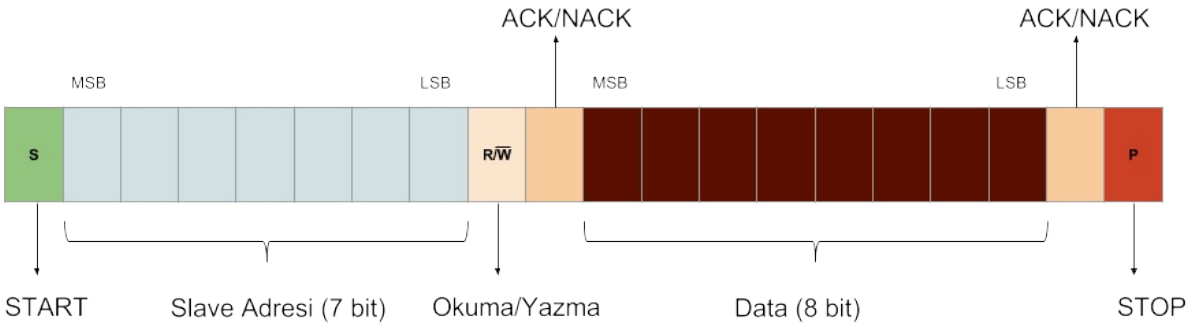
Şimdi IC2 aşamalarında daha yakından bakalım.

Veri transferi sırasında, SDA hattındaki verinin, SCL lojik 0 iken değişmesi ve 1 iken değişmemesi gerekmektedir. SCL lojik 0'dan 1'e geçerken (yükselen kenar) veri örneklenmektedir. Bu sebeple SCL lojik 1 iken SDA'nın lojik seviyesinin değişmesi, iletişimin başlatılması ve sonlandırılması gibi, özel anlamlara gelmektedir.

IC2 üzerindeki iletişim 8 bit'lik paketler şeklindedir, sonrasında alıcı taraf **9.bit** olarak bir onay biti (ACK/NACK) göndermektedir. Lojik 0 ACK, lojik 1 ise NACK anlamına gelmektedir. ACK ve NACK master ve slave için aşağıdaki anlamlara gelmektedir.

Eğer veriyi gönderen taraf master ve alıcı slave ise, slave tarafından gönderilen ACK verinin doğru bir şekilde alındığını, NACK ise bir problem olduğunu gösterir. Veriyi gönderen taraf slave ise, master yeni bir veri paketi talep ediyorsa ACK, etmiyorsa NACK gönderir ve sonrasında iletişimi sonlandırır. Buradaki NACK biti bir probleme işaret etmemektedir.

Master ve slave arasında 1 byte'lık alışveriş sırasında, SDA üzerinden, gidip gelen veri aşağıdaki şekilde gösterilmiştir.



Şimdi, alt seviye programlamayla uğraşmaksızın, Linux üzerinde I2C protokolünün nasıl kullanılabildiğine bakalım.

Linux Altında I2C İşlemleri

Linux üzerinde, bir önceki bölümde anlatılan detaylarla uğraşmaksızın, bir sürücü (driver) üzerinden I2C protokolünü kullanmak mümkündür.

> **Not:** Linux yüklü sistem üzerinde bir I2C controller chip'i olduğunu varsayıyoruz. Ayrıca, yazılımsal olarak, GPIO pinleri üzerinden, I2C protokolünü oluşturmak da mümkündür (*bit banging*).

I2C protokülünü kullanabilmek için ilk olarak aygıt sürücüsüyle etkileşime geçebiliyor olmalıyız. Bu amaçla **/dev** altında gerekli aygıt düğümleri bulunuyor olmalıdır. I2C sürücüsüne, bu aygıt düğümleri üzerinden, standart Giriş/Çıkış sistem çağrılarını kullanarak erişeceğiz.

> **Not:** I2C desteği kernel içinde built-in olabileceği gibi modül şeklinde de bulunabilir. Bu durumda gerekli modül (i2c-dev.ko) kernel adres alanına yüklenmelidir.

IC2 protokolünü kullanabilmek için birden çok yöntem olmakla birlikte, temel olarak **open, read, write** ve **ioctl** sistem çağrılarının kullanılması yeterlidir. Burada üç farklı yöntemden bahsedeceğiz. Her üç yöntemde de ilk olarak open sistem çağrısıyla aygıt dosyası açılmakta ve elde edilen *handle* ile işlemlere devam edilmektedir. Temelde birbirine benzeyen bu yöntemlerin detaylarına geçelim.

`read`, `write` Sistem Çağrıları İle Okuma/Yazma

Burada yapılan işlemler standart dosyalar üzerinde yapılanlara benzemektedir. İşlem adımları aşağıdaki gibidir:

1. `open` ile aygıt dosyasını aç ve daha sonraki çağrılarda kullanılacak handle değerini sakla
2. `ioctl` çağrısı ile iletişime geçilecek aygıtı adresle
3. `read`, `write` fonksiyonlarıyla I2C üzerinden okuma ve yazma işlemlerini gerçekleştir
4. Dosyayı kapat

Şimdi bu adımlara daha yakından bakalım. Her bir adıma ilişkin örnek kod aşağıdaki gibidir.

Aygıt dosyasını aşağıdaki gibi açabiliriz.

```
/* I2C aygıt dosyasının /dev/i2c-1 olduğu varsayılmıştır */
#define I2C_DEVICE "/dev/i2c-1"

int i2c_fd;
i2c_fd = open(I2C_DEVICE, O_RDWR);
if (i2c_fd < 0) {
    perror("Opening " I2C_DEVICE);
    exit(1);
}
```

İletişime geçilecek slave aygıt `ioctl` ile aşağıdaki gibi belirtilebilir. **I2C_SLAVE** istek kodu, slave adresinin gönderildiğini belirtmektedir. Fonksiyon prototipi ve örnek kullanımı şu şekildedir:

```
int ioctl(i2c_fd, I2C_SLAVE, SLAVE_ADDR);
```

```
/* Slave adresinin 0x20 olduğunu kabul edelim. */
#define SLAVE_ADDR 0x20

int rc;
rc = ioctl(i2c_fd, I2C_SLAVE, SLAVE_ADDR);
if (rc < 0) {
    perror("ioctl slave addressing");
    exit(1);
}
```

Devamında I2C üzerinden `read` çağrısı ile 1 byte bilgi okuyalım:

```
char data[] = {0};
rc = read(i2c_fd, data, 1)
if (rc < 0) {
    perror("reading");
    exit(1);
}
```

I2C üzerinden 1 byte bilgi gönderelim:

```
char data[] = {0};
rc = write(i2c_fd, data, 1)
if (rc < 0) {
    perror("writing");
    exit(1);
}
```

Bu yöntem oldukça anlaşılır ve basit olmasına karşın, her bir yazma ve okuma işlemi ayrı ayrı yapılmalıdır. Tek seferde, atomik olarak, yazma ve okuma işlemleri yapılamamakta, ayrıca her bir okuma ve yazma işleminde I2C protokolündeki bütün aşamalardan geçilmektedir. Yani her bir okuma veya yazma işleminde önce başlangıç durumu oluşturulmalı, slave aygıt yeniden adreslenmeli, bilgi alışverişi sağlanmalı ve iletişim sonlandırılmalıdır. Bir sonraki yöntemde ise bu dezavantajlar bertaraf edilmiştir.

`ioctl` Çağrısı İle Okuma Okuma/Yazma

Bu yöntemde `read`, `write` çağrıları yerine sadece `ioctl` çağrısı kullanılmaktadır. Bu sayede iletişim sonlandırılmadan birleşik (combined) okuma/yazma işlemleri yapılabilmektedir. I2C aşamalarını anlattığımız bölümdeki başlangıç aşamasının tekrarlanması (repeated start) bahsini hatırlayınız.

Bu yöntem için, **i2c-tools** kaynak kodundan çıkan, **i2c-dev.h** başlık dosyasına ihtiyaç duyulmaktadır. i2c-tools, I2C ile ilgili birçok aracı barındıran açık kaynak kodlu bir projedir. Bu aşamada i2c-dev.h içindeki sadece sembolik sabitlere ve structure tanımlarına ihtiyaç duymaktayız.

> **Not:** i2c-tools kaynak koduna aşağıdaki gibi erişebilirsiniz.
>
> wget -c https://github.com/groeck/i2c-tools/archive/master.zip
>
> Ayrıca kernel içinde de i2c-dev.h isimli bir başka başlık dosyası daha bulunmaktadır. Bu dosya kernel içerisindeki I2C sürücüsü tarafından kullanılmakta olup, uygulama örneğimizde bu başlık dosyasını değil, `i2c-tools` paketinden çıkanı kullanacağız.

İlk olarak, okuma veya yazma yapılacak tampon alanlarının başlangıç adresleri, uzunlukları ve iletişim kurulacak slave adreslerini gösteren veri alanı hazırlanmalıdır. Bu amaçla, i2c-dev.h içinde **i2c_rdwr_ioctl_data** isimli bir yapı türü tanımlanmıştır. ioctl çağrısına, bu yapı türünden bir adres geçirilmelidir.

`i2c_rdwr_ioctl_data`, gerekli verilere erişimi sağlayan, **i2c_msg** isimli bir yapı elemanına sahiptir.

```
/* I2C_RDWR ioctl çağrılarında kullanılan yapı */
struct i2c_rdwr_ioctl_data {
    struct i2c_msg *msgs;   /* mesajlaşmak için gerekli verilerin başlangıç adresi */
    __u32 nmsgs;            /* mesaj sayısı */
};

 /* I2C_RDWR ioctl çağrısı için gerekli mesaj alanlarını tutan yapı */
struct i2c_msg {
    __u16 addr;             /* slave adresi */
    unsigned short flags;   /* Okuma yazma ve diğer seçenekler */
#define I2C_M_TEN   0x10
#define I2C_M_RD    0x01    /* Okuma işlemi yapılacağını gösteren sembolik sabit */
#define I2C_M_NOSTART       0x4000
#define I2C_M_REV_DIR_ADDR  0x2000
#define I2C_M_IGNORE_NAK    0x1000
#define I2C_M_NO_RD_ACK     0x0800
    short len;              /* tampon alanın uzunluğu */
    char *buf;              /* tampon alanın başlangıç adresi */
};
```

Örnek bir kullanım aşağıdaki gibidir, flags'e atanan **I2C_M_RD** okuma, 0 değeri ise yazma işlemi yapılacağını gösterir.

```
#define I2C_DEVICE "/dev/i2c-1"
#define SLAVE_ADDR 0x20

int i2c_fd;
struct i2c_rdwr_ioctl_data msgset;
struct i2c_msg iomsgs[2];

/*yazma işleminde kullanılacak tampon alanı*/
unsigned char wbuf[1] = {0x15};

/*okuma işleminde kullanılacak tampon alanı*/
unsigned char rbuf[1];

int rc;

int i2c_fd;
i2c_fd = open(I2C_DEVICE, O_RDWR);
if (i2c_fd < 0) {
    perror("Opening " I2C_DEVICE);
    exit(0);
}

iomsgs[0].addr = SLAVE_ADDR;
/*yazma işlemi*/
iomsgs[0].flags = 0;
iomsgs[0].buf = wbuf;
iomsgs[0].len = 1;

iomsgs[1].addr = SLAVE_ADDR;
/*okuma işlemi*/
iomsgs[1].flags = I2C_M_RD;
iomsgs[1].buf = rbuf;
iomsgs[1].len= 1;

msgset.msgs = iomsgs;
/*mesaj sayısı*/
msgset.nmsgs = 2;

rc = ioctl(i2c_fd, I2C_RDWR, &msgset);
if (rc < 0) {
    perror("ioctl I2C_RDWR");
    exit(0);
}
```

Örnekte, 0x20 adresli slave aygıta 1 byte'lık 0x15 değeri gönderilmiş, ardından iletişim sonlandırılmadan 1 byte'lık veri okunmuştur.

`i2c-dev.h` Fonksiyonları İle Okuma/Yazma

ic2-tools paketinden çıkan i2c-dev.h başlık dosyası ayrıca, **i2c_smbus_** önekiyle başlayan, inline fonksiyon tanımları da içermektedir. Veri okuma ve yazmaya ilişkin en temel fonksiyon kümesi aşağıdaki gibidir:

```
__s32 i2c_smbus_read_byte(int file);
__s32 i2c_smbus_read_byte_data(int file, __u8 command);
__s32 i2c_smbus_write_byte(int file, __u8 value);
__s32 i2c_smbus_write_byte_data(int file, __u8 command, __u8 value);
__s32 i2c_smbus_read_word_data(int file, __u8 command);
__s32 i2c_smbus_write_word_data(int file, __u8 command, __u16 value);
```

Bu fonksiyonlar SMBus protokolü ile I2C veriyolunu kullanmaktadır. **SMBus**, 1995 yılında Intel tarafından tanımlanmış, I2C protokülünü temel alan bir protokoldür. I2C aygıtları, garanti olmamakla birlikte, çoğunlukla bu protokolü de desteklemektedir. Bir sonraki bölümde SMBus desteğinin nasıl kontrol edileceğinden bahsedeceğiz.

İsminde byte geçen fonksiyonlar 1 byte, word geçenler ise 2 byte'lık okuma veya yazma fonksiyonlarıdır. Ayrıca, sonu **_data** ile biten fonksiyonlar mevcuttur. Örneğin;

```
__s32 i2c_smbus_read_byte(int file);
__s32 i2c_smbus_read_byte_data(int file, __u8 command);
```

i2c_smbus_read_byte, slave aygıttan 1 byte veri okumaktadır. **i2c_smbus_read_byte_data** ise öncesinde 1 byte'lık bir veri göndermekte, sonrasında cevap olarak gelen veriyi okumaktadır. Master aygıt ilk olarak slave aygıta okuma işlemine ilişkin bir komut göndermekte ve sonrasında gelen veriyi okumaktadır. Örneğin, master tarafından gönderilen bilgi, okuma yapılacak olan slave aygıtın register adresi olabilir. İlerideki örneklerimizde bu kullanıma değineceğiz.

Fonksiyonlardaki komut parametresine ilave olarak kullanılan `file` ve `value` parametreleri ise sırasıyla, `open` ile elde edilmiş handle değerini ve slave aygıta gönderilecek değeri göstermektedir.

Parametre	Anlamı
file	open ile elde edilen handle
command	Okuma veya yazma öncesi gönderilen komut
value	Gönderilecek veri

Sürücü Desteğinin Test Edilmesi

Kullanılacak I2C sürücüsü, tanımlı tüm I2C özelliklerini desteklemeyebilir. Bu sebeple uygulama içinde kullanılacak olan özelliklerin desteklenip desteklenmediğini kontrol etmek doğru bir yaklaşım olacaktır.

`I2C_FUNC` ioctl çağrısı ile sürücünün yetenekleri hakkında bilgi alınabilir. ioctl çağrısına ayrıca `long` türünden bir değişkenin adresi geçirilmelidir. Desteklenen özellikleri temsil eden değer bu adrese yazılacaktır. Örnek bir kullanım aşağıdaki gibidir:

```
long funcs;
int rc;
rc = ioctl(fd, I2C_FUNCS, &funcs);
if (rc < 0) {
    perror("ioctl I2C_FUNCS");
    exit(0);
}
```

Bu aşamadan sonra `func` değerini, `i2c-dev.h` içindeki sembolik sabitlerle bit and işlemine tabi tutarak herhangi bir özelliğin desteklenip desteklenmediği bilgisine erişebiliriz.

Örneğin `I2C_FUNC_I2C` sembolik sabiti en temel seviyedeki I2C komutlarının desteklendiğini göstermektedir. Bu desteğin mevcut sürücüde var olup olmadığını aşağıdaki gibi kontrol edebiliriz:

```
if (!(funcs & I2C_FUNC_I2C)) {
    puts("i2c support not available");
}
```

Diğer bazı sembolik sabitler ise aşağıdaki gibidir.

Sembolik Sabit	Anlamı
I2C_FUNC_I2C	Temel I2C komut desteği
I2C_FUNC_10BIT_ADDR	10bit'lik Slave adres desteği
I2C_FUNC_SMBUS_BYTE	SMBus read_byte ve write_byte desteği
I2C_FUNC_SMBUS_BYTE_DATA	SMBus read_byte_data ve write_byte_data desteği
I2C_FUNC_SMBUS_WORD_DATA	SMBus read_word_data ve write_word_data desteği

I2C Bus Hızının Değiştirilmesi

Linux tarafında I2C bus hızını değiştirmek isterseniz, kullandığınız kartın I2C sürücü kaynak kodlarını incelemeniz gereklidir. Temel olarak aşağıdaki senaryolarlardan biriyle karşılaşacak olsanız da en doğru kaynak kullandığınız I2C sürücüsünün kodu olacaktır.

Bazı kartlarda I2C desteği modül olarak derlendiğinde, modül yükleme aşamasında bir parametre vermek suretiyle bus hızı değiştirilebilmektedir. Örnek olarak Raspberry Pi kartında I2C sürücüsüne ait modül ismi `i2c_bcm2708` şeklinde olup modül yükleme aşamasında aşağıdaki gibi `baudrate` parametresini vermek suretiyle değiştirilebilmektedir:

```
# modprobe  i2c_bcm2708 baudrate=400000
```

Modül şeklinde yapılıyor olmasının bir avantajı da, modülü sistemden kaldırıp (`modprobe -r i2c_bcm2708`) ardından farklı bir `baudrate` parametresi ile yeniden yükleme imkanının bulunmasıdır.

Bazı kartlarda ise baudrate değeri sürücü kodu içerisinde sabit olarak tanımlanmıştır. Platformunuzda bu şekilde bir I2C sürücüsü kullanıyorsanız tek yöntem, sürücü kaynak kodunu değiştirerek sistemi yeni derlediğiniz çekirdek ile açmak olacaktır.

Device Tree Block kullanan yeni nesil bir kart kullanıyor iseniz, kartınıza uygun **dtb** dosyasının üretim sürecinde kullanılan **dtsi** kaynak dosyalarında gerekli güncellemeleri yapıp, **DTC** derleyicisi ile **dtb** dosyasını yeniden üretmelisiniz.

Örnek bir dtsi dosyası içerisinde I2C bloğu aşağıdaki gibidir:

```
&i2c1 {
        clock-frequency = <100000>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_i2c1_2>;
        status = "okay";
        wm8903: wm8903@1a {
                compatible = "wlf,wm8903";
                reg = <0x1a>;

                clocks = <&clks 201>;
                amic-mono;
                gpio-cfg = <0xffffffff 0xffffffff 0 0xffffffff 0xffffffff>;
       };

};
```

Yukarıdaki örnekte `clock-frequency` değerini değiştirebilirsiniz.

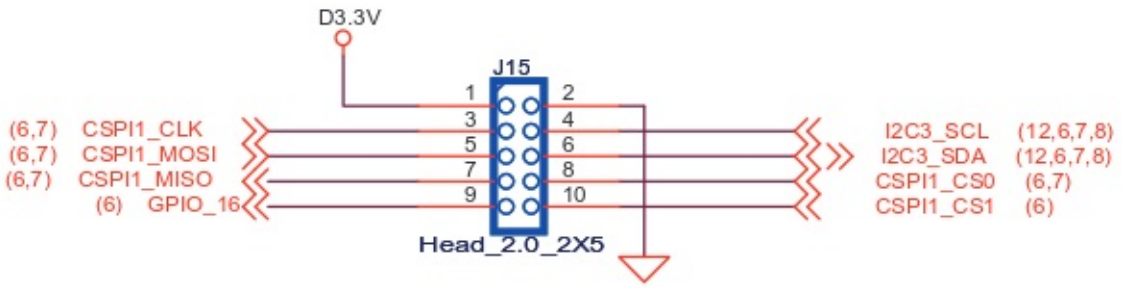
Board Seçimi ve İlk İşlemler

Örneklerimizde embedded board olarak, Poslab Corp tarafından üretilen, **SavageBoard Quad (i.MX6Q)** seçilmiştir. Diğer alternatifler Raspberry Pi veya BeagleBone Black olabilirdi. İzlenecek adımlar karttan karta çok fazla değişmeyecektir.

SavageBoard Quad'ın genel özellikleri aşağıdaki gibidir.

Özellik	Değer
İşlemci	4 adet Freescale i.MX6 Q ARM Cortex-A9
Bellek	DDR3 RAM 1GB x64 bit at 533MHz
eMMC	8GB
Diğerleri	MicroUSB, USB, Ethernet, MicroSD, HDMI

İlk olarak kart üzerindeki I2C pinlerini belirlememiz gerekiyor. Kartın şematiğine baktığımızda, J15 header üzerinde 3 numaralı I2C portunun bizim erişimimize açık olduğunu görüyoruz.



J15 üzerindeki gerekli pin numaraları ve karşılıkları aşağıdaki gibidir:

Pin	Görevi
1	Vdd
2	Ground
4	SCL
6	SDA

Cihazı, I2C desteği ile derlenmiş bir kernel ile, açtığımızda **/dev/i2c-0**, **/dev/i2c-1** ve **/dev/i2c-2** aygıt dosyalarının oluştuğunu görmekteyiz.

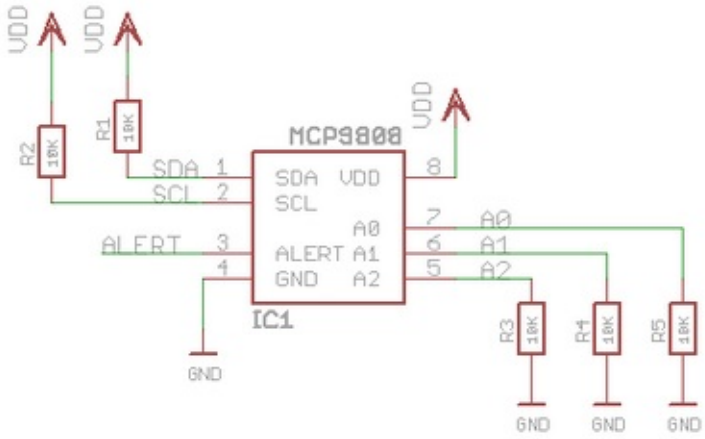
Bu aşamadan sonra I2C üzerinden iletişim kuracağımız bir çevre birimine ihtiyacımız olacaktır, bu amaçla bir sayısal sıcaklık sensörü kullanacağız.

Sıcaklık Sensörünün Seçilmesi

Sıcaklık ölçümü için, **Adafruit** firması tarafından üretilen, **MCP9808** breakout board seçilmiştir. MCP9808, gerçekte PCB üzerindeki, Microchip tarafından üretilmiş, I2C sıcaklık sensörünün adıdır.

> **Not:** Breakout board'lar, genellikle bir elektriksel komponent ile çalışmayı kolaylaştırmak için tasarlanmış baskı devrelerdir (PCB, Printed Circuit Board). Elektriksel komponentlerin pinleri birbirine oldukça yakın olabilmektedir, bu durumda bu komponentleri breadboard üzerinde kullanmak veya lehim yapmak oldukça zordur. Breakout board'lar kullandıkları komponentin pinlerine karşılık gelen, araları açılmış, pinlere sahiptir. Bu sayede bu komponentlerle breadboard üzerinde de çalışmak kolaylaşmaktadır.
>
> Adafruit firmasının, bu şekilde değişik sensörlerin kullanılmasını kolaylaştıran breakout PCB tasarımları mevcuttur.

PCB'nin şematiği aşağıdaki gibidir:



Teknik Özellikler

MCP9808, kullanıcı tarafından programlanabilen ve okunabilen yazmaçlara sahiptir. Sıcaklık ölçüm çözünürlüğü ve sıcaklık limiti gibi değerler bu yazmaçlar yoluyla belirlenmekte ayrıca sıcaklık değeri de yine bir yazmaç üzerinden okunmaktadır. Yazmaçlar 8 bit'lik adreslere sahiptir ve 16 bit genişliğindedir.

Bir yazmaç içeriği okunmak istendiğinde, I2C üzerinden, ilk önce yazmaç adresi gönderilmeli ardından cihazdan 2 byte'lık veri okunmalıdır. Okunan ilk byte, yüksek anlamlı byte'dır.

MCP9808, besleme ve I2C uçları (SDA, SCL) hariç, 1 adet alert ve 3 adet adres uçuna daha sahiptir. Kullanıcı tarafından belirlenen sıcaklık değerinin aşılması ya da altına düşülmesi gibi durumlarda alert ucundaki elektriksel seviye değişmektedir.

> **Not:** alert ucu open collector özelliği göstermektedir, bu yüzden bir pull-up direnci ile Vdd'ye çekilmelidir.
>
> SDA ve SCL uçları ise, PCB üzerinde, 10K'lık dirençlerle Vdd'ye çekilmiştir.

Alert ucu, mikroişlemcili bir sistem için interrupt kaynağı olarak kullanılabilir. A0, A1 ve A2 şeklinde isimlendirilen diğer uçlar ise adres uçlarıdır. Bu 3 uç üzerindeki elektriksel seviye değiştirilerek sensörün adresi değiştirilebilmektedir.

MCP9808 4 bit'i sabit, diğer 3 bit ise kullanıcı tarafından değiştirilebilen 7 bitlik bir adrese sahiptir. Adresin sabit ve değişebilen kısmı aşaşıdaki gibidir.

A6	A5	A4	A3	A2	A1	A0
0	0	1	1	x	x	x

> **Not:** Bu bilgi Microchip MCP9808 datasheet'inde PIN DESCRIPTION bölümünde yer almaktadır.

Yukarıdaki PCB şematiğinde gösterildiği gibi **A2**, **A1** ve **A0** adres uçları, birer pull-down direnci ile, toprak seviyesine çekilmiştir. Bu durumda bu uçlar lojik 0 seviyesindedir. Bu uçların elektriksel seviyesi üzerinde bir işlem yapılmazsa MCP9808 adresi **0011000** yani **0x18** olacaktır.

Linux altındaki I2C işlemlerini incelerken, slave adresinin ioctl çağrısı ile kullanıldığını hatırlayınız. Örneğin:

```
int slave_addr = 0x18;
rc = ioctl(i2c_fd, I2C_SLAVE, slave_addr);
```

Örnekteki 0x18 değeri sıcaklık sensörünü tanımlamaktadır.

Programlanabilir adres uçları sayesinde, 8 farklı sensörü aynı I2C hattı üzerine bağlamak mümkündür. Biz bu uçlardaki elektriksel seviyeyi değiştirmeyeceğiz.

MCP9808 ile ilgili önemli gördüğümüz özellikleri maddeleyecek olursak:

- I2C ve SMBus standartları desteklenmektedir.
- -40°C to 125°C sıcaklık aralığı 0.25°C tipik doğruluk ile ölçülebilmektedir.
- Ölçüm çözünürlüğü değiştirilebilmektedir (+0.5°C, +0.25°C, +0.125°C, +0.0625°C). Default değer 0.0625°C'dir.
- 2.7V ile 5.5V arasında çalışabilmektedir.
- Ayarlanabilir adresi pinleri sayesinde 8 tane sensör aynı veri yolu üzerinde bulunabilir.
- Sıcaklık sınırı belirlenebilir ve alert ucu bir interrupt kaynağı olarak kullanılabilir.

Sıcaklık Değerinin Okunması

MCP9808 sıcaklık değerine karşılık gelen analog bir gerilim üretmekte ve daha sonra bu değeri sayısallaştırarak bir yazmaça yüklemektedir. Bu 16 bit'lik yalnız okunabilen yazmaç, Microchip dokümanlarında, ortam sıcaklık yazmacı (Ambient Temperature Register) olarak isimlendirilmektedir.

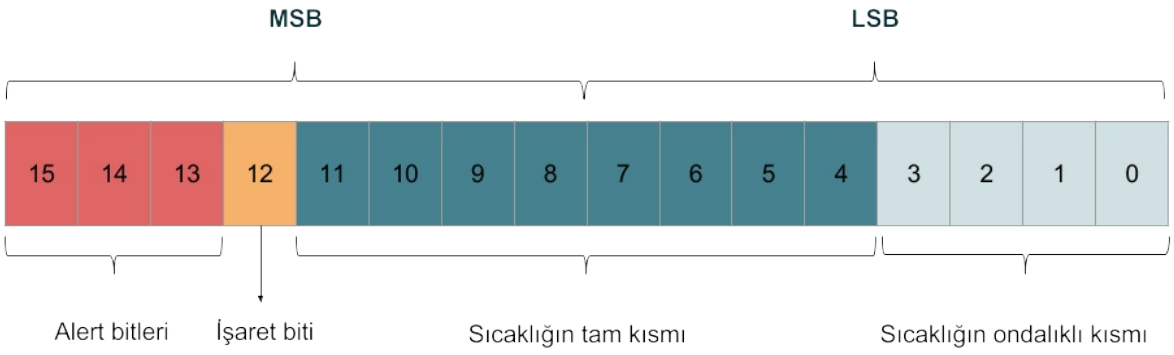
Bu 16 bit'in tamamı sıcaklık değerine karşılık gelmemektedir. Yüksek anlamlı 4 bit, alert durumu ve sıcaklığın negatif veya pozitif oluşuna ilişkindir. Sıcaklık değeri ise geriye kalan düşük anlamlı 12 bit ile ifade edilmektedir.

Daha önce bir sıcaklık sınırının aşılması veya altına düşülmesi durumunda bir alert durumu oluştuğunu söylemiştik. Aslında, 3 farklı durumda alert oluşabilmektedir. Ölçülen sıcaklık kritik bir değer, bir üst ve bir alt değer ile karşılaştırılmakta ve nihayetinde bir alert oluşmaktadır. Bu değerler için, MCP9808 içinde, *crit*, *upper* ve *lower* şeklinde isimlendirilen yazmaçlar bulunmaktadır. Ortam sıcaklık yazmacı içindeki ilk 3 bit alert durumunu çözümlemek için kullanılmaktadır. İlk 3 bit'in karşılıkları aşağıdaki gibidir.

Bit	Alert detayı
15. bit 1 ise	Ölçülen sıcaklık kritik değere eşit veya daha büyük
14. bit 1 ise	Ölçülen değer üst sınırı aşmış
13. bit 1 ise	Ölçülen değer alt sınırın altına düşmüş

12.bit ise işaret bitidir. Geriye kalan düşük anlamlı 12 bit ise sıcaklığı, sayının ikiye tümleyeni (two’s complement) şeklinde, ondalıklı bir biçimde ifade etmektedir.

Sıcaklık sabit noktalı bir sayı olarak ifade edilmektedir. En düşük anlamlı 4 bit sayının ondalıklı kısmını göstermektedir. Sıcaklık yazmacının içeriğini aşağıdaki gibi temsil edebiliriz.



MCP9808 Sıcaklık Yazmacının İçeriği

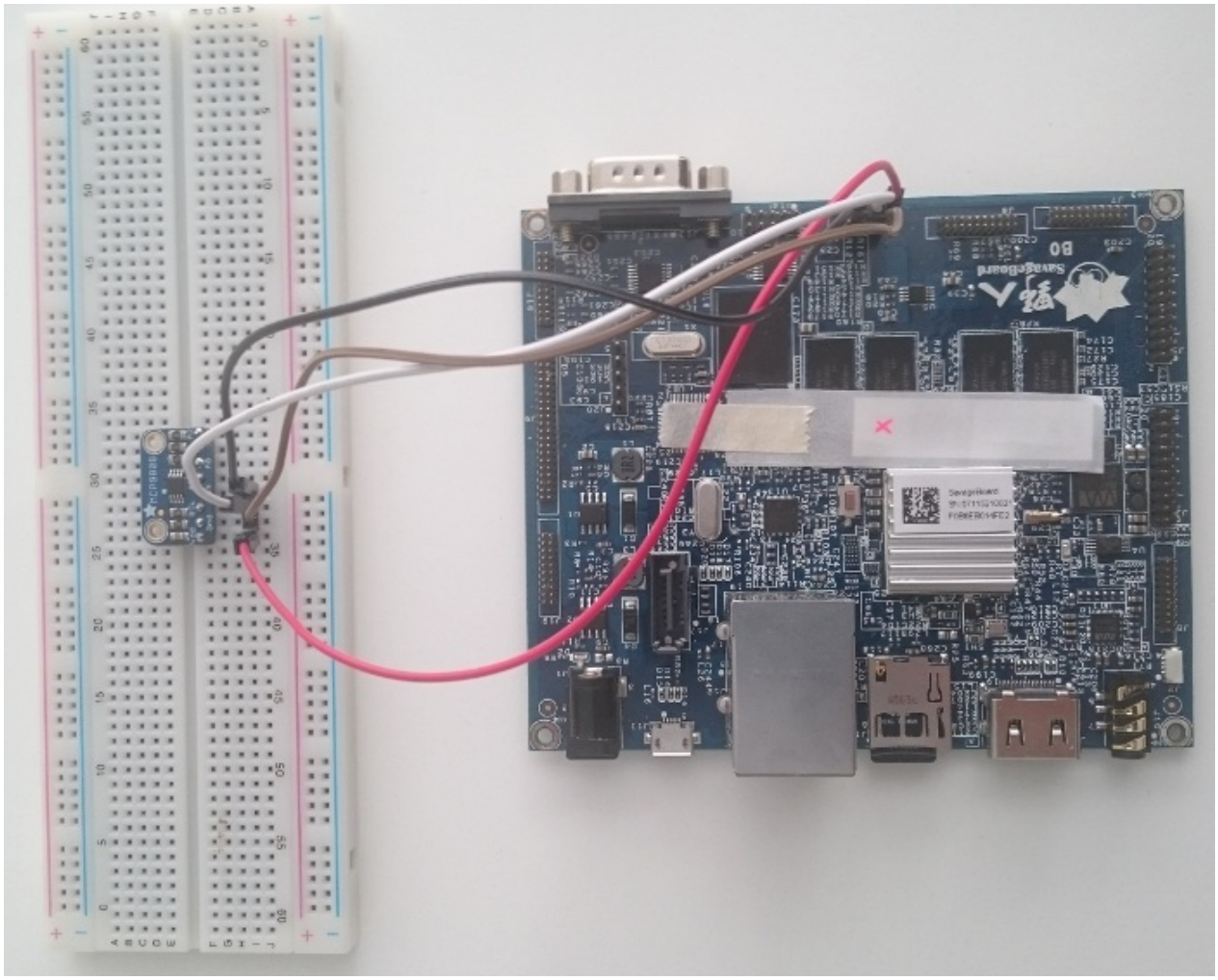
> **Not:** Daha önce öntanımlı ölçüm çözünürlüğünün 0.0625°C olduğununu söylemiştik. Ölçüm çözünürlüğü, sayısal verinin değerini değiştiren en düşük değeri göstermektedir. Sıcaklık yazmacındaki en düşük anlamlı bit, noktanın sağındaki 4. basamağı ifade etmektedir. Bu basamağın çarpan değeri 1/16 yani 0.0625 değeridir. Bu değeri daha sonraki işlemlerimizde kullanacağız.
>
> Ayrıca sıcaklığın ölçülmesi (Temperature Conversion Time) ölçüm çözünürlüğüne bağlı olarak değişmektedir. Bu değer öntanımlı durumda 250 ms'dir. Yani bu süreden daha hızlı yapılan okumalar aynı değeri üretecektir.

Ortam sıcaklık sensörünün adresi **0000 0101** yani **0x5** olarak tanımlanmıştır. Sıcaklık okumalarında bu değeri kullanacağız.

Kablolama

Adafruit MCP9808 PCB, beraberinde breadboard üzerinde çalışmaya imkan veren 8'li bir pin header'ı ile gelmektedir. Pinleri PCB'ye lehimledikten sonra breadboard üzerinde çalışabilirsiniz.

Bu aşamadan sonra SavageBoard ile MCP9808 arasındaki bağlantıyı kurmalıyız. Bu amaçla 4 adet atlama kablosuna ihtiyacımız olacak. Vdd, toprak, SCL ve SDA uçlarını karşılıklı olarak bağlamalıyız. MCP9808 pull-up dirençlerine sahip olduğu için başka bir bileşene ihtiyaç bulunmamaktadır.



Sıcaklık Sensörünün Test Edilmesi

SavageBoard ile sensör arasındaki gerekli kablolamayı yaptıktan sonra, herhangi bir kod yazmadan, sensöre I2C üzerinden erişip erişemediğimizi kontrol edebiliriz. Bu amaçla i2c-tools araçlarını kullanabiliriz.

İlk olarak, i2c-tools içindeki araçları SavageBoard hedefli derlemeliyiz. Sisteminizdeki çapraz derleyiciyi kullanarak derleme işlemini aşağıdakine benzer şekilde yapabilirsiniz. Biz derlemelerimizde **CodeSourcery** araçlarını kullanacağız.

```
$ CC=arm-none-linux-gnueabi-gcc make
```

Test amaçlı olarak **tools** dizinindeki **i2cdetect** ve **i2cget** araçlarını kullanacağız.

Bu aşamadan sonra cihaza enerji verdiğimizde /dev altında i2c portlarına ilişkin aygıt dosyası düğümlerini görebilmeliyiz.

```
# ls /dev/i2c*
/dev/i2c-0  /dev/i2c-1  /dev/i2c-2
```

I2C veri yolları üzerindeki cihazları `i2cdetect -y i2cbus` şeklinde belirlemek mümkündür. **i2cbus** değeri olarak, 0, 1, 2 gibi I2C veri yolu numaraları girilmelidir. Biz sensörümüzü SavageBoard üzerindeki 3 numaralı I2C portuna bağlamıştık. i2cdetect ile tüm veri yolları üzerindeki aygıtlar belirlenebilir. 3 numaralı veri yolu için, numaralandırma 0'dan başladığı için 2 değeri girilerek elde edilen sonuç aşağıdaki gibidir.

```
/mnt # ./i2cdetect -y 2
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- 18 -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

0x18 değerini daha önce Microchip dokümanından elde ettiğimizi hatırlayınız.

i2cdetect, bir veri yolu üzerindeki tüm muhtemel adresleri denemekte ve aldığı yanıta göre cihazları tespit etmektedir.

Bir diğer test aracı olarak ise i2cget'yi kullanabiliriz. Bu aracın en genel kullanım hali aşağıdaki gibidir:

```
i2cget [-f] [-y] I2CBUS CHIP-ADDRESS [DATA-ADDRESS [MODE]]
```

Sensör adresinin **0x18** olduğunu belirlemiştik, sıcaklık değerini okuyacağımız yazmaç adresinin de **0x5** olduğunu söylemiştik. Bu durumda i2cget aşağıdaki gibi bir sonuç üretmektedir.

```
/mnt # ./i2cget -y 2 0x18 0x5 w
0xd7c1
```

i2cget, MCP9808'i adresleyip bağlantıyı kurduktan sonra, okuma yapacağı yazmaca ilişkin adresi gönderir. MCP9808 bu sayede hangi yazmacın adreslendiğini anlayıp içeriğini göndermektedir.

i2cget, ilk okunan değeri düşük anlamlı olarak ifade etmektedir. Fakat, MCP9808 tarafından, ilk olarak yüksek anlamlı byte gönderilmektedir. Bu durumda gerçekte sıcaklık yazmacındaki değer aşağıdaki değer yani `0xc1d7` şeklinde olacaktır. Bu sayının bitsel karşılığını yazarak önceki bilgilerimize göre yorumlamaya çalışalım.

```
1100 0001 1101 0111
```

Yüksek anlamlı ilk 3 bit alert durumuyla ilişkili olduğu için göz ardı edebiliriz. 12. bit ise işaret bit'idir ve sayının pozitif olduğunu göstermektedir. İşaret biti 0 ise sayı pozitif, 1 ise negatif olarak ele alınmalıdır. Geriye kalan 12 bitin 8 biti sıcaklığın tam değeri, düşük anlamlı 4 biti ise ondalıklı değerini göstermektedir. Bu 12 bitlik sayının onluk sistemdeki karşılığını hesaplayıp sonrasında 16'ya bölerek veya 0,0625 çözünürlük değeriyle çarparak sıcaklık değerine ulaşabiliriz.

0001 1101 0111 sayısının onluk sistemdeki karşılığı 471'dir. Bu değeri 0,0625 ile çarptığımızda 29,4375 değerine ulaşmaktayız.

Sıcaklık Değerinin Yazılımsal Olarak Elde Edilmesi

Bu bölümde, daha önce **Linux Altında I2C İşlemleri** konusunda bahsettiğimiz yöntemlere ilişkin birer örnek vereceğiz.

Örnek kodları derlerken, **i2c-tools** içinden çıkan **i2c-dev.h** başlık dosyasına ihtiyacımız olacak, bu sebeple derleme işleminde bu başlık dosyasının bulunduğu dizini de derleyiciye göstermeliyiz.

Her 3 örnekte de MCP9808 adresi, sıcaklık değerini okuyacağımız yazmaç adresi ve sıcaklık çözünürlük değeri aşağıdaki sembolik sabitlerle gösterilmiştir.

```
#define MCP9808_ADDR    0x18    // MCP9808 I2C adresi
#define TEMP_REG_ADDR   0x05    // Ortam sıcaklık yazmaç adresi
#define RESOLUTION      0.0625  // Sıcaklık çözünürlüğü
```

3 örneğimizde yalnızca sıcaklık değerinin okunma kısımları farklıdır. Sıcaklık yazmacındaki değer elde edildikten sonraki yorumlama kısmı ise değişmemektedir. Daha önce `i2cget` ile elde ettiğimiz değeri inceleyerek sıcaklık değerine ulaşmıştık. Benzer işlemi burada C kodu ile yapacağız.

Her 3 uygulamada da temelde yapılan işlemler, MCP9808 ile iletişim kurmak, sonrasında sıcaklık yazmacının adresini göndermek ve periyodik olarak sıcaklık yazmacının değerini okuyarak yorumlamak şeklindedir.

İlk olarak sıcaklık yazmaç değerinin **t** isimli **int** bir değişkene çektiğimizi varsayalım. Bu aşamadan sonraki işlemler aşağıdaki gibi olacaktır.

```
int t = SICAKLIK_YAZMAÇ_DEĞERİ;
double temp;
temp = t & 0x0FFF;
temp *= RESOLUTION;
if (t & 0x1000) {
    temp -= 256;
}
printf("Sıcaklık: %.2f C\n", temp);
```

Bu kod üzerinden tekrar eski bilgilerimizi tekrarlayalım. Sıcaklık yazmacındaki yüksek anlamlı 4 bit'in sıcaklığın mutlak değeriyle alakalı olmadığını hatırlayın. 15., 14. ve 13. bitler alert bitleri, 12. bit ise işaret bitidir. Bu sebeple başlangıçta bu bitleri göz ardı edebiliriz.

Aşağıdaki kod bu duruma ilişkindir.

```
temp = t & 0x0FFF;
```

Bu değer daha sonra sıcaklık ölçüm çözünürlüğüyle çarpılarak, mutlak sıcaklık değerine erişilir.

```
temp *= RESOLUTION;
```

Bu aşamadan sonra sıcaklığın polaritesi için işaret bitine (12. bit) bakılmalıdır. 0x1000 ile bitsel and işlemi sonucu 1 çıkıyorsa sayı negatif demektir. Sıcaklığın sayının ikiye tümleyeni şeklinde tutulduğunu daha önce söylemiştik. Bu sebeple bu durumda sayıdan 256 değeri çıkarılmaktadır.

256 sayısının nereden geldiğini daha iyi anlamak için aşağıdaki nota bakabilirsiniz.

> **Not:** Bir sayının 2'ye tümleyeni alınırken, ilk önce 1'e tümleyenin alındığını daha sonra elde edilen sonucun 1 ile toplandığını hatırlayınız. 1'e tümleme işlemi için sayının 1 olan bitleri 0, 0 olanlar ise 1 yapılmalıdır.
>
> Bir byte sınırlarındaki bir sayı için konuşacak olursak. Sayının bitlerinin ters çevrilmesi neticesinde elde edilen sayı, sayının kendisinin 255 sayısı ile arasındaki farkı vermektedir. Çünkü sayının kendisiyle bitlerinin ters çevrilmiş halinin toplamı 255 sayısını verecektir.
>
> Sonrasındaki 1 ile toplama işlemi ise sayının 255 ile arasındaki farktan 1 fazlasının elde edilmesine neden olacaktır. Aynı işlem sayının 256 sayısı ile arasındaki fark ile de elde edilebilir.

```
if (t & 0x1000) {
    temp -= 256;
}
```

Bu aşamadan sonra temp değişkeni Celsius derece cinsinden sıcaklık değerini göstermektedir.

Örnek kodları sırasıyla `i2c_1.c`, `i2c_2.c` ve `i2c_3.c` adlarıyla saklayıp, çapraz derleyicinize `i2c-dev.h` dosyasının yerini göstererek derleyebilirsiniz. Bizim sistemimiz için geçerli derleme aşağıdaki gibidir.

```
$ arm-none-linux-gnueabi-gcc -o i2c_1 i2c_1.c -I i2c-tools-3.1.2/include
```

Her 3 örnekte de sıcaklık 1 saniye aralıklarla okunarak konsola basılmaktadır.

Yazma ve Okuma İşlemlerinde `read`, `write` Sistem Çağrılarının Kullanılması

Örnek kodda, I2C üzerinden okuma işlemleri read/write sistem çağrılarıyla yapılmaktadır.

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <fcntl.h>

#define MCP9808_ADDR    0x18    // MCP9808 I2C adresi
#define TEMP_REG_ADDR   0x05    // Ortam sıcaklık yazmaç adresi
#define RESOLUTION      0.0625  // Sıcaklık çözünürlüğü

void main()
{
    int i2c_fd;
    char *bus = "/dev/i2c-2";
    long funcs;
    int rc;

    if ((i2c_fd = open(bus, O_RDWR)) < 0)
    {
        printf("Failed to open the bus. \n");
        exit(1);
    }

    // İletişim kurulacak aygıtın adreslenmesi
    ioctl(i2c_fd, I2C_SLAVE, MCP9808_ADDR);

    // İçeriği okunacak yazmacın adreslenmesi
    char reg[1] = {TEMP_REG_ADDR};
    write(i2c_fd, reg, 1);
    char data[2] = {0};

    do {
        // Sıcaklık yazmaç değerinin okunması
        if(read(i2c_fd, data, 2) != 2) {
            printf("read error \n");
        }
        else {
            double temp;
            int t = (data[0] << 8) + data[1];
            temp = t & 0x0FFF;
            temp *= RESOLUTION;
            if (t & 0x1000) {
                temp -= 256;
            }
            printf("Sıcaklık: %.2f C\n", temp);
        }
        sleep(1);
    } while(1);
}
```

Yazma ve Okuma İşlemlerinde `ioctl` Çağrılarının Kullanılması

Örnek kodda, I2C üzerinden okuma işlemleri ioctl çağrısı ile yapılmıştır. Bu yöntemde okuma ve yazma işlemleri, hattın kullanımını bırakmaksızın, combined bir şekilde yapılabilmektedir.

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/i2c.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <fcntl.h>

#define MCP9808_ADDR    0x18
#define TEMP_REG_ADDR   0x05
#define RESOLUTION      0.0625

int i2c_fd;

int i2c_read16(int addr, int reg) {
    struct i2c_rdwr_ioctl_data msgset;
    struct i2c_msg iomsgs[2];
    char buf[1], rbuf[2];
    int rc;

    buf[0] = (char) reg;

    iomsgs[0].addr = iomsgs[1].addr = (unsigned) addr;
    iomsgs[0].flags = 0;        /* Yazma */
    iomsgs[0].buf = buf;
    iomsgs[0].len = 1;

    iomsgs[1].flags = I2C_M_RD; /* Okuma */
    iomsgs[1].buf = rbuf;
    iomsgs[1].len = 2;

    msgset.msgs = iomsgs;
    msgset.nmsgs = 2;

    if ( (rc = ioctl(i2c_fd, I2C_RDWR, &msgset)) < 0 )
        return -1;
    return (rbuf[0] << 8) | rbuf[1];
}

void main()
{
    char *bus = "/dev/i2c-2";
    double temp;
    int t;
    if ((i2c_fd = open(bus, O_RDWR)) < 0)
```

```
    {
        printf("Failed to open the bus. \n");
        exit(1);
    }
    do {
        t = i2c_read16(MCP9808_ADDR, TEMP_REG_ADDR);
        if (t < 0) {
            printf("read error \n");
             continue;
        }
        temp = t & 0x0FFF;
        temp *= RESOLUTION;
        if (t & 0x1000) {
            temp -= 256;
        }
        printf("Sıcaklık: %.2f C\n", temp);
        sleep(1);
    } while (1);
}
```

Yazma ve Okuma İşlemlerinde `i2c-dev.h` Fonksiyonlarının Kullanılması

Bu yöntemde, i2c-tools fonksiyonlarını kullandık. Gerekli fonksiyonlar i2c-dev.h başlık dosyasında inline olarak tanımlıdır.

Örnekteki bir noktaya dikkatinizi çekmek istiyoruz. **i2c_smbus_read_word_data** ile okunan değerin yüksek ve düşük anlamlı byte'ları swap edilmiştir.

```
rt = i2c_smbus_read_word_data(i2c_fd, TEMP_REG_ADDR);
t = rt << 8 | rt >> 8;
```

Daha önce benzer işlemi i2cget ile elde ettiğimiz değer için de yaptığımızı hatırlayınız. **i2c_smbus_read_word_data**, ilk okuduğu değeri düşük anlamlı olarak ifade etmektedir. Fakat MCP9808 üzerinden ilk olarak yüksek anlamlı byte gönderilmektedir. Bu sebeple, sıcaklık değeri yorunlanmadan önce, bir swap işlemi yapılmaktadır.

```c
#include <stdio.h>
#include <stdlib.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <fcntl.h>

#define MCP9808_ADDR    0x18
#define TEMP_REG_ADDR   0x05
#define RESOLUTION      0.0625

void main()
{
    int i2c_fd;
    char *bus = "/dev/i2c-2";
    double temp;
    int t;
    int rt;

    if ((i2c_fd = open(bus, O_RDWR)) < 0)
    {
        printf("Failed to open the bus. \n");
        exit(1);
    }

    ioctl(i2c_fd, I2C_SLAVE, MCP9808_ADDR);

    do {
        rt = i2c_smbus_read_word_data(i2c_fd, TEMP_REG_ADDR);
        t = rt << 8 | rt >> 8;
        temp = t & 0x0FFF;
        temp *= RESOLUTION;
        if (t & 0x1000){
            temp -= 256;
        }
        printf("Sıcaklık: %.2f C\n", temp);
        sleep(1);
    } while(1);
}
```

Strace Kullanımı

Nasıl Çalışır?

Unix tabanlı sistemlerde **strace** gibi bir uygulamanın varolabilmesi için gereken `ptrace` sistem çağrısı uzun yıllardır (**SVr4** ve **4.3BSD**) bulunmaktadır.

Sistem çağrısı ismini **process trace** kavramından alır. `ptrace` sistem çağrısı üzerinden bir uygulama başka bir uygulamanın durumunu takip edebildiği gibi değişiklikler yapma imkanına da sahip olmaktadır.

`ptrace` sistem çağrısı temel olarak **gdb** gibi debug uygulamalarında, **strace**, **ltrace** gibi sistem veya kütüphane çağrılarını takip uygulamalarında, *code coverage* araçlarında, çalışan yazılım koduna dokunmadan bazı hataların giderilmesinde veya güvenlik kontrollerinden geçirilmesinde kullanılır.

`ptrace` çağrısıyla bir uygulamanın kontrolü tamamen başka bir uygulamaya verilmektedir. Buradaki kontrolden kastımız, uygulamanın kullandığı tüm bellek alanına erişim, sinyallerin alınması, değiştirilmesi, file descriptor'ların yönetimi hatta uygulamanın kod segmentinin değiştirilerek yamalar yapılması dahil aklımıza gelebilecek hemen her türden tehlikeli değişikliklere izin veriliyor olmasıdır.

Bahsettiğimiz bu özelliklerinden ötürü, bir uygulamanın başka bir uygulamayı `ptrace` ile kontrol edebilmesi için, ilgili uygulamaya **sinyal** gönderme yetkisinin bulunması gerekir. Dolayısıyla özel durumlar haricinde her kullanıcının kendi sahibi olduğu diğer uygulamaları ptrace ile kontrol edebileceğini, **root** kullanıcısının da sistemdeki tüm uygulamaları kontrol edebileceğini söyleyebiliriz.

Linux Capabilities API sisteminin geliştirilmesinden sonra yukarıda koşullardan bağımsız olarak, **CAP_SYS_PTRACE** özelliği sayesinde de `ptrace` izni verilebilmektedir.

Tipik Kullanım

Uygulamanızı `strace` ile aşağıdaki biçimde başlatmanız yeterlidir:

```
$ strace ls /tmp
```

Ancak çoğu zaman çok daha önceden başlatılmış ve çalışmaya devam eden, bununla birlikte herhangi bir sorun nedeniyle ek bilgi toplamak istediğiniz durumlar oluşur. `strace` ile herhangi bir çalışan uygulamaya, `-p` parametresine `<PID>` değerini vermek suretiyle *attach* olabiliriz:

```
$ strace -p $(pidof mysqld)
Process 26829 attached - interrupt to quit
select(13, [10 12], NULL, NULL, NULL)   = 1 (in [10])
fcntl64(10, F_SETFL, O_RDWR|O_NONBLOCK) = 0
accept(10, {sa_family=AF_INET, sin_port=htons(33033), sin_addr=inet_addr("192.168.0.15
")}, [16]) = 34
fcntl64(10, F_SETFL, O_RDWR)            = 0
rt_sigaction(SIGCHLD, {SIG_DFL, [CHLD], SA_RESTART}, {SIG_DFL, [CHLD], SA_RESTART}, 8)
 = 0
getpeername(34, {sa_family=AF_INET, sin_port=htons(33033), sin_addr=inet_addr("192.168
.0.15")}, [16]) = 0
...
```

`-f` Parametresi

Strace'in uygulamanın tüm thread'lerini ve uygulamadan *fork()* edilen diğer çocuk süreçlerini takip edebilmesi için `-f` parametresi verilmelidir.

```
$ strace -f ./example

$ strace -f -p $(pidof mysqld)
```

`-e` Parametresiyle Filtreleme

Strace çıktısı zaman zaman takip için oldukça kalabalık olabilir.

Sadece belirli sistem çağrılarını takip etmek istiyorsanız `e` parametresi ile bunu yapabilirsiniz:

```
$ strace -f -e trace=open,write,close,connect,select -p 3245
```

Sadece dosya işlemleriyle ilgili sistem çağrılarını takip etmek için `-e trace=file` kullanılabilir:

```
$ strace -e trace=file 4535
```

Sadece network ile ilgili sistem çağrılarını filtrelemek için `-e trace=network` kullanılabilir:

```
$ strace -e trace=network 23232
```

Zaman Bilgisi Alma: `-tt`

Sistem çağrılarının çıktısı alınırken saniye hassasiyetinde zaman bilgisi de almak istiyorsak `-t` parametresini kullanabiliriz.

Çoğu zaman saniye hassasiyeti yeterli olmayacaktır. Mikrosaniye hassasiyetinde zaman bilgisi almak için `-tt` parametresini kullanabiliriz:

```
$ strace -tt ls /tmp
...
00:07:10.595807 openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
00:07:10.595885 getdents(3, /* 6 entries */, 32768) = 176
00:07:10.595977 getdents(3, /* 0 entries */, 32768) = 0
00:07:10.596041 close(3)
```

İstatistiki Bilgi Alma: `-c`

`-c` parametresi ile istediğimiz süre boyunca sistem çağrılarıyla ilgili istatistik toplayabiliriz:

```
$ strace -f -c -p $(pidof mysqld)
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
42.89    0.592035         275      2149       151 read
28.11    0.388024        2587       150        18 futex
27.82    0.384023        1352       284           select
 1.16    0.016001        3200         5           rt_sigtimedwait
 0.01    0.000154           0      1457           write
 0.01    0.000152          22         7           accept
 0.00    0.000007           0      9478           time
 0.00    0.000000           0       108           open
 0.00    0.000000           0       136           close
 0.00    0.000000           0        40           unlink
 0.00    0.000000           0         5           alarm
 0.00    0.000000           0         9           access
 0.00    0.000000           0        21           ioctl
 0.00    0.000000           0      2409           gettimeofday
 ................
```

`-o` İle Çıktıların Kayıt Edilmesi

`strace` uzun süreli çalıştırılıp oluşan loglar daha detaylı olarak geniş zaman aralığında incelenecekse, logların kayıt edilmesi gerekecektir.

`-o` parametresi ile logların kayıt edileceği dosyayı belirtebilirsiniz:

```
$ strace -f -o /tmp/strace.log -e trace=file ls /tmp
```

Ptrace Engelleme

Linux altında bir uygulamanın, kendisinin **root** harici kullanıcılar tarafından `ptrace` sistem çağrısı ile kontrol edilmesini engelleyebilmesine imkan verilmiştir.

Bu işlem için `prctl` özel sistem çağrısı kullanılır. Uygulama `prctl` aracılığıyla kendisi için `PR_SET_DUMPABLE` bayrağını temizleyecek olursa **root** haricindeki kullanıcıların uygulamaya sinyal gönderme hakkı olsa dahi bu uygulamayı `ptrace` ile kontrol etme şansları olmaz.

Bu özelliğin en tipik kullanımlarından biri, OpenSSH authentication agent yazılımında görülür. Böylelikle kullanıcıların parola girme aşamasında uygulamanın `ptrace` ile başka bir uygulama tarafından kontrolü engellenmiş olur.

Güvenlik

Linux kullanımının yaygınlaşmasıyla birlikte zararlı yazılımlara rastlanma sıklığı da artmaktadır. Geleneksel Linux process modelindeki `ptrace` imkan seti sebebiyle, sisteminizde kendi kullanıcınızla çalıştırdığınız herhangi bir yazılım içerisine zararlı bir kod enjekte edilmiş ise (en basit *xterm* aracından gelişmiş web tarayıcı uygulamalarına kadar), `ptrace` sistem çağrısı sayesinde çalışan diğer tüm uygulamalarınızın kontrolünün bu zararlı yazılım tarafından devralınması ve siz hiç bir şey farketmeden önemli bilgilerin kopyalanması mümkündür.

Pek çok kullanıcının farkında olmadığı bu duruma karşılık, Linux çekirdeği içerisindeki **Yama** kod adlı güvenlik modülüyle bir koruma mekanizması geliştirilmiştir. (Ayrıntılı bilgiler için: https://www.kernel.org/doc/Documentation/security/Yama.txt)

Yama modülü olan Linux çekirdeğinde, `/proc/sys/kernel/yama/ptrace_scope` dosyası üzerinden `ptrace` sistem çağrısına verilecek tepki kontrol altına alınabilmektedir. Öntanımlı olarak bu dosyada **0** değeri yazmaktadır. Dosyada yazan değer aşağıdaki tablo doğrultusunda yorumlanır:

Değer	Anlam
0	**Geleneksel davranış:** önceki bölümde anlatılanlar doğrultusunda `ptrace` yapabilme hakkının bulunduğu tüm uygulamalar kontrol edilebilir
1	**Kısıtlandırılmış `ptrace` :** sadece uygulamanın doğrudan parent process'lerine veya `PR_SET_PTRACER` opsiyonuyla uygulama tarafından izin verilen debug uygulamalarına ait `PID` değerlerinin eşleştiği uygulamalara kontrol izni verilir. Böylece `gdb program_adi` ve `strace program_adi` şeklindeki kullanımlar çalışmaya devam eder ancak çalışan bir uygulamaya sonradan *attach* olmaya izin verilmeyecektir (dolayısıyla `strace -p PID` yöntemi de çalışmayacaktır). Diğer opsiyon da özellikle KDE, Chromium, Wine gibi uygulamaların kullandığı, debug/crash handler'a ait PID değerinin `PR_SET_PTRACER` ile uygulama içerisinden belirtilmesi ve bu sayede spesifik bir uygulamanın `ptrace` yapabilmesi şeklindedir
2	**Sistem yöneticisine `ptrace` :** sadece `CAP_SYS_PTRACE` özelliği tanımlanmış uygulamalar veya `prctl` ile `PTRACE_TRACEME` opsiyonunu tanımlayan çocuk process'ler kontrol edilebilir
3	**Tamamen devre dışı:** hiç bir şart altında `ptrace` yapılmasına imkan tanınmaz. Bu özellik bir defa tanımlandığı takdirde çalışma anında tekrar değişiklik yapılamaz

Her ne kadar uygulamalar `prctl` üzerinden kendilerinin `ptrace` yapılabilmesini **root** kullanıcısı dışında devre dışı bırakabiliyor olsalar da, pek çok yazılımcı bu detayların farkında değildir. OpenSSH agent'ı gibi doğrudan güvenlikle ilgili yazılımlar bu işlemleri yapıyor olmasına karşın, sistemde çalışan tüm yazılımlardan aynı davranışı beklemek doğru olmaz. Bu nedenle sistem genelinde yazılımdan bağımsız çözümlerin üretilmesi önem taşır. Son zamanlarda bazı Linux dağıtımları (Ubuntu vb.) yukarıda tariflediğimiz `ptrace_scope` dosyasının öntanımlı değerini **1** yapmaya başlamışlardır. Böylelikle `ptrace` işlemleri kısıtlandığından sistem genelinde daha güvenli bir çalışma ortamı sağlanmaktadır.

Android kullanan sistemler de düşünüldüğünde bu gibi güvenlik konularında daha hassas olunması gerektiği açıktır.

Örnek strace gerçekleştirimi

Aşağıdaki örnek uygulamayı `ministrace.c` adıyla kaydedip

```
$ gcc -m32 -o ministrace ministrace.c
```

ile derleyebilirsiniz.

> **Not:** Örnek uygulama 32 bitlik sistemler için yazılmış olup mimari farklılıkları gözardı edilmiştir. Bu sebeple 32 bitlik bir sistemde veya `gcc-multilib` kurulu ise `-m32` parametresi ile derlemelisiniz.

```
/* ministrace.c */
#include <sys/ptrace.h>
#include <sys/reg.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int wait_for_syscall(pid_t child)
{
    int status;
    while (1) {
        ptrace(PTRACE_SYSCALL, child, 0, 0);
        waitpid(child, &status, 0);
        if (WIFSTOPPED(status) && WSTOPSIG(status) & 0x80)
            return 0;
        if (WIFEXITED(status))
            return 1;
    }
}

int do_child(int argc, char **argv)
{
    char *args [argc+1];
    memcpy(args, argv, argc * sizeof(char*));
    args[argc] = NULL;

    ptrace(PTRACE_TRACEME);
    kill(getpid(), SIGSTOP);
    return execvp(args[0], args);
}

int do_trace(pid_t child)
{
    int status, syscall, retval;
    waitpid(child, &status, 0);
    ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACESYSGOOD);
    while(1) {
        if (wait_for_syscall(child) != 0) break;

        syscall = ptrace(PTRACE_PEEKUSER, child, sizeof(long)*ORIG_EAX);
        fprintf(stderr, "syscall(%d) = ", syscall);

        if (wait_for_syscall(child) != 0) break;

        retval = ptrace(PTRACE_PEEKUSER, child, sizeof(long)*EAX);
        fprintf(stderr, "%d\n", retval);
    }
    return 0;
```

```
}

int main(int argc, char **argv)
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s prog args\n", argv[0]);
        exit(1);
    }

    pid_t child = fork();
    if (child == 0) {
        return do_child(argc-1, argv+1);
    } else {
        return do_trace(child);
    }
}
```

Uygulama derlendikten sonra herhangi bir komutu `ministrace` ile çalıştırıp çıktıyı inceleyebiliriz:

```
$ ./ministrace date
syscall(11) = 0
syscall(12) = 21843968
syscall(21) = -2
syscall(9) = 164245504
...
syscall(9) = 164241408
syscall(1) = Tue Mar  3 13:44:52 EET 2015
29
syscall(3) = 0
...
```

Örnek uygulamamızda **65** satırlık bir kod ile strace uygulamasının temel çalışma prensibi gösterilmeye çalışılmıştır. Daha gelişmiş bir örnekte sistem çağrılarının numaralarından isimlerine ulaşmak, çağrıda kullanılan parametreleri ve geri dönüş kodlarının ilgili sistem çağrısı özelinde anlamlarını göstermek mümkün olabilir.

GNU Build Sistemi Araçları

Bu bölümde uygulamaların kaynak kodlarının derlenmesi ve sisteme kurulması süreçlerinde yardımcı olacak araçlar ele alınmaktadır.

Linux sistemler üzerinde yazılım geliştirme yaparken, sıklıkla bu araçları kullanmakta olduğumuzdan, bu konulardaki farkındalığın artırılmasının sonraki süreçlerde pek çok faydası olacaktır.

Temel olarak `make` uygulaması ve `Makefile` dosyaları kullanımı anlatılacak, daha sonra kod taşınabilirliğini ve kurulum süreçlerini de hedefleyen `autoconf & automake` sistemiyle ilgili genel bilgi verilmeye çalışılacaktır.

Make

Uygulama geliştirirken sıklıkla obje dosyalarımızı yeniden ve yeniden oluşturmak zorunda kalırız. Yerine göre `gcc`, `ld`, `ar` vb. uygulamaları tekrar tekrar aynı parametrelere çağırırız. İşte `make` ugulaması, programların yeniden derlenme sürecini otomatik hale getirmek, sadece değişen kısımların yeniden derlenmesini sağlamak suretiyle zamandan kazanmak ve işlemleri her zaman otomatik olarak doğru sırada yapmak için tasarlanmıştır.

Temel Kurallar

`make` uygulaması çalıştırıldığında, bulunulan dizinde sırasıyla `GNUmakefile`, `makefile` ve `Makefile` dosyalarını arar. Alternatif olarak `-f` seçeneği ile Makefile olarak kullanacağınız dosyayı da belirlemeniz mümkün olsa da standartların dışına çıkmamakta fayda var. `make` neyi nasıl yapacağını bu dosyalardan öğrenecektir. Eğer bulunduğunuz dizinde bir `Makefile` dosyası yoksa aşağıdaki gibi bir çıktı alacaksınız demektir:

```
$ make
make: *** No targets specified and no makefile found.  Stop.
```

> Genel kabul görmüşlüğü ve göz alışkanlığı açısından dosya adı olarak alternatiflerin yerine `Makefile` isminin kullanılması önerilir

Bir `Makefile` aslında işlemlerin nasıl yapılacağını gösteren kural tanımlamalarından oluşmaktadır. Genel olarak dosyanın biçimi aşağıdaki gibidir:

```
hedef1: bağımlılıklar
<TAB> komut
<TAB> komut
<TAB> ...

hedef2: bağımlılıklar
...
```

Burada en sık yapacağımız hata `TAB` tuşuna basmayı unutmak olacaktır. `Makefile` dosyasını hazırladığınız editörden kaynaklanan bir problem de olabilir. `TAB` işlemine dikkat edilmediğinde aşağıdaki gibi bir uyarı alabilirsiniz:

```
$ make
Makefile:6: *** missing separator (did you mean TAB instead of 8 spaces?)
```

Kurallar arasında bir satır boş bırakılması GNU make için zorunlu olmamakla birlikte bazı Unix versiyonlarıyla uyumluluk için boşluk bırakılması gereklidir.

İlk satırda `hedef1` 'in oluşturulmasında etkili olan, dolayısıyla bağımlılık üreten dosyalar birbirinden boşluk ile ayrılmış olarak tek satırda listelenir. Eğer bağımlılık kısmında yer alan dosyalardan en az birinin son değiştirilme tarihi, `hedef1` 'den daha yeni ise, `hedef1` yeniden oluşturulur. Diğer durumda `hedef1` 'in yeniden oluşturulmasına gerek olmadığı anlaşılır, çünkü `hedef1` 'in bağımlı olduğu dosyalarda `hedef1` üretildikten sonra bir değişiklik olmamıştır.

> NOT: Görüldüğü üzere dosyaların son değiştirilme tarihleri üzerinden işleyen bir mekanizma bulunmaktadır. Sistem saatiniz ileri veya geri zıplarsa kurallar doğru çalışamayacaktır. Bu durumda `make clean` ile önce tam bir temizlik yapılıp yeniden süreci başlatabilirsiniz.
>
> Eğer sistem zamanındaki oynamalar nedeniyle, dosyaların son güncellenme zamanları, o anki sistem saatinden daha ileride ise, **make: warning: Clock skew detected. Your build may be incomplete** şeklinde bir uyarı alınacaktır.

Sonraki satırlarda bağımlılık yaratan bu dosyalardan `hedef1` 'in oluşturulabilmesi için gerekli komutlar yer alır. Şimdi basit bir örnek için önce yeni bir dizin oluşturup içerisinde aşağıdaki `Makefile` dosyasını oluşturalım:

```
bolgeler: marmara karadeniz ege
    cat marmara karadeniz ege | sort -u > bolgeler

marmara: istanbul bursa
    cat istanbul bursa | sort -u > marmara

karadeniz: samsun sinop
    cat samsun sinop | sort -u > karadeniz

ege: izmir aydin
    cat izmir aydin | sort -u > ege

clean:
    rm -f ege karadeniz marmara bolgeler
```

Dosyamızı hazırladıktan sonra `make` komutunu çalıştıralım:

```
$ make
make: *** No rule to make target `istanbul', needed by `marmara'.  Stop.
```

Yukarıdaki örnekte neler olduğunu anlamaya çalışalım:

1. `make` uygulamasına `Makefile` içerisindeki bir hedef kural ismini parametre olarak

vermediğimizde öntanımlı olarak dosyada bulduğu ilk hedefi gerçekleme çalışır
2. Dosyamızdaki ilk hedefin `bolgeler` olduğunu gördük
3. `bolgeler` hedefinin bağımlılıkları `marmara`, `karadeniz` ve `ege` dosyaları şeklindeymiş
4. Eğer bulunduğumuz dizinde `bolgeler` dosyası zaten mevcut ve son değiştirilme tarihi, bağımlılıkları olan `marmara`, `karadeniz` ve `ege` dosyalarının her üçünden de daha güncel olsa idi, `make` yeni bir işlem yapmaya gerek olmadığını düşünecekti. Ancak bizim dizinimizde henüz bu dosyaların hiç biri yok
5. Bu nedenle `bolgeler` hedefini gerçeklemek için öncelikle dizinde mevcut olmayan `marmara` hedefini gerçeklemek için işlemlere başlandı
6. `marmara` hedefi de benzer şekilde `istanbul` ve `bursa` dosyalarına bağımlı ve her iki dosya da sistemde yok
7. Bir önceki durumdan farklı olarak, `istanbul` dosyası dizinde mevcut olmadığı gibi bu dosyayı üretecek herhangi bir hedef de tanımlı değil.
8. Bu yüzden ***marmara*** *hedefini üretebilmek için gereken (dizinde mevcut olmayan)* ***istanbul*** *hedefini üretecek kural da yok* şeklinde bir hata mesajı ile `make` süreci sonlanmıştır

Eğer bulunduğumuz dizinde, `istanbul` ve `bursa` dosyalarını oluşturacak olursak, `make` sonrası `marmara` hedefinin üretildiğini görebileceğiz. Aynı şekilde diğer bölgeler için de `Makefile` dosyasında tanımladığımız kurallar doğrultusunda gereken dosyaları ürettiğimizde, onlar da `make` tarafından oluşturulacaktır.

Sonrasında herhangi bir il dosyasını güncellediğimizde, ona bağlı bölge tüm bölgeleri içeren dosya otomatik olarak güncellenecektir.

Şimdi de `C` dilinde yazılmış basit bir uygulamanın derlenmesi sürecine yönelik örnek `Makefile` dosyamıza bakalım:

```
CC     = gcc
CFLAGS = -O2 -Wall -pedantic
LIBS   = -lm -lnsl

all: install

ornek: ornek.o
    $(CC) $(CFLAGS) $(LIBS) -o ornek ornek.o

ornek.o: ornek.c
    $(CC) $(CFLAGS) -c ornek.c

clean:
    rm -f ornek *.o

install: ornek
    cp ornek /usr/local/bin
```

Bu örnekte hedef olarak `ornek` uygulaması derlenecektir. Uygulamanın bağımlı olduğu dosya `ornek.o` şeklinde olup bu dosya da `ornek.c` kaynak kod dosyasına bağımlıdır.

İlk satırda yer alan `CC` değişkeniyle kullanacağımız derleyiciyi belirliyoruz. Makefile dosyaları içerisinde bu şekilde değişken tanımlaması yapıp, değişkeni dosya içerisinde `$(değişken)` şeklinde kullanabiliriz. İkinci satırda ise derleyiciye vereceğimiz bazı seçenekleri `CFLAGS` değişkenine atıyoruz. Üçüncü satırda uygulamamızın linklenmesi gereken kütüphaleri `-l` parametresiyle listeledik. Ardından ilk kuralımız geliyor: `ornek` dosyası `ornek.o` dosyasına bağımlı olarak belirtilmiş ve `ornek.o` 'dan `ornek` 'in oluşturulabilmesi için gerekli komut hemen altında listelenmiştir. Değişkenlerin değerlerini yerine koyduğumuzda komutumuz `gcc -O2 -Wall -pedantic -lm -lnsl -o ornek ornek.o` şeklinde olacaktır.

İkinci kuralımız `ornek.o` 'nun nasıl oluşturulacağını belirtmektedir. `ornek.c` dosyasında bir değişiklik olduğunda `ornek.o` dosyası hemen altında listelenen komutla yeniden oluşturulur:

```
$ gcc -O2 -Wall -pedantic -c ornek.c
```

Üçüncü kuralımızda çalıştığımız dizinde nasıl temizlik yapacağımızı belirtiyoruz. `make clean` komutunu çalıştırdığımızda `ornek` dosyası ve `.o` ile biten obje dosyaları silinecektir.

Dördüncü kuralımız ise `install` şeklinde. Bu kuralda da ornek dosyasında bir değişme olduğunda `cp ornek /usr/local/bin` komutu ile dosyayı /usr/local/bin dizini altına kopyalıyoruz.

Makefile içerisindeki her bir kural `make` uygulamasına seçenek olarak verilebilir ve ayrıca işletilebilir. Yukarıdaki gibi bir Makefile dosyasına sahipsek `make ornek.o` komutuyla sadece ornek.o için verilen kuralın çalıştırılmasını sağlayabiliriz. Veya `make install` komutuyla sadece install kuralının çalışmasını sağlayabiliriz. Ancak `install` hedefi aynı zamanda `ornek` 'e bağımlı olduğundan `ornek` için girilen kurallar da çalışacaktır. Aynı şekilde `ornek` de `ornek.o` 'ya bağımlı olduğundan `ornek.o` kuralı da çalışacaktır.

Şimdi bu `Makefile` dosyasının bulunduğu yerde `ornek.c` kaynak dosyasını da hazırladığımızı varsayarak aşağıdaki çıktıları inceleyelim:

```
$ make ornek.o
gcc -O2 -Wall -pedantic -c ornek.c

$ make clean
rm -f ornek *.o

$ make
gcc -O2 -Wall -pedantic -c ornek.c
gcc -O2 -Wall -pedantic -lm -lnsl -o ornek ornek.o
cp ornek /usr/local/bin
```

Yukarıdaki Makefile örneğimize tekrar dönelim. `make clean` komutunu çalıştırdığımızda derleme sonrasında oluşan dosyalar silinmektedir. Peki, bulunduğumuz dizinde ismi **clean** olan bir dosya mevcut ise ne olur?

```
$ touch clean
$ make clean
make: `clean' is up to date.
```

Gördüğünüz gibi **clean** adında bir dosya var olduğu ve clean için bağımlılık listesi olmadığından dolayı, kuralın güncelliğini koruduğunu ve alttaki komutların çalıştırılmaması gerektiğini düşündü. İşte bu gibi durumlar için özel bir kural mevcuttur: `.PHONY`

Yukarıda anlatılan problemi giderebilmek için Makefile dosyamızın içeriğine aşağıdaki kuralı da eklemeliyiz:

```
.PHONY: clean
```

Böylelikle `make clean` komutunun, bulunulan dizinde **clean** adında bir dosya olsa bile düzgün olarak çalışmasını sağlamış olduk, bir nevi `clean` hedefini korumaya almış olduk.

Soyut Makefile Kuralları Tanımlamak

Önceki bölümde temel olarak make kullanımı üzerinde durduk. Örnek bir Makefile hazırladık. Ancak tek bir kaynak dosyasından oluşturulan bir uygulama için `make` sistemi o kadar da yararlı bir şey değil. Zaten gerçekte de en küçük uygulama bile onlarca kaynak dosyasından oluşur. Şimdi böyle bir uygulama için Makefile hazırlayalım.

Örnek: Soyut kurallar kullanılmamış Makefile

Aşağıdaki bir kısmı ortak kullanılan az sayıda kaynak dosyadan oluşan 2 adet uygulamanın derleme sürecini yöneten Makefile örneğini inceleyiniz:

```
LIBS            = -lm \
                  -lrt \
                  -lpthread \
                  $(shell pkg-config --libs openssl)

INCLUDES        = -I/usr/local/include/custom

all: server client

server: common.o server.o list.o
    $(CC) $(CFLAGS) $(LIBS) -o server common.o server.o list.o

client: common.o client.o
    $(CC) $(CFLAGS) $(LIBS) -o client common.o client.o

common.o: common.c common.h
    $(CC) $(CFLAGS) $(INCLUDES) -c common.c

server.o: server.c server.h common.h list.h
    $(CC) $(CFLAGS) $(INCLUDES) -c server.c

client.o: client.c client.h ortak.h
    $(CC) $(CFLAGS) $(INCLUDES) -c client.c

list.o: list.c list.h
    $(CC) $(CFLAGS) $(INCLUDES) -c list.c

install: client server
    mkdir -p /usr/local/bin
    cp client /usr/local/bin/
    cp server /usr/local/bin/

uninstall:
    rm -f /usr/local/bin/client
    rm -f /usr/local/bin/server

clean:
    rm -f *.o server client

.PHONY: clean
```

Kullandığımız derleyici, derleyici seçenekleri, kütüphaneler gibi değerleri değişkenlere atamakla neler kazandığımıza bir bakalım. Derleyici parametrelerini değiştirmeye karar verdiğimizde değişken kullanmıyor olsaydık 6 farklı yerde bu değişikliği el ile yapmak zorunda kalacaktır. Fakat şimdi ise sadece `CFLAGS` değişkeninin değerini değiştirmemiz yeterli olacaktır.

Ancak gene de yukarıdaki gibi bir Makefile yazmak uzun sürecek bir işlemdir. Eğer uygulamanız 60 adet **.c** dosyasından oluşuyorsa ve 60 farklı obje için tek tek kuralları yazmak zorunda kalıyorsanız bu hoş olmaz. Çünkü tüm **.o** dosyalarını üretebilmek için vereceğimiz komut aynı: `$(CC) $(CFLAGS) $(INCLUDES) -c xxx.c` Oysa biz 60 defa bu komutu tekrar yazmak zorundayız. İşte bu noktada soyut kurallar (*abstract rules*) imdadımıza yetişir.

Bir soyut kural genel olarak `*.u1` uzantılı bir dosyadan `*.u2` uzantılı bir dosyanın nasıl üretileceğini tanımlar. Kullanımı aşağıdaki gibidir:

```
.u1.u2:
    komutlar
    komutlar
...
```

Burada `u1` kaynak dosyanın uzantısı iken, `u2` hedef dosyanın uzantısıdır. Bu tür kullanımda dikkat ederseniz bağımlılık tanımlamaları yer almamaktadır. Çünkü tanımladığımız soyut genel kural için bağımlılık belirtmek çok anlamlı değildir. Bunun yerine **.u1** uzantılı bir dosyadan **.u2** uzantılı dosya üretmede istisnai olarak farklı bağımlılıkları olan kurallar da ileride vereceğimiz örnekte olduğu gibi belirtilebilir.

Soyut kurallar tanımlarken aşağıdaki özel değişkenleri kullanmak gerekecektir:

Özel Değişken	İşlevi
$<	Değiştiği zaman hedefin yeniden oluşturulması gereken bağımlılıkları gösterir
$@	Hedefi temsil eder
$^	Geçerli kural için tüm bağımlılıkları temsil eder

Bu bilgiler ışığında hemen bir örnek verelim. Uzantısı **.cpp** olan bir kaynak kodundan obje kodunu üretebilmek için aşağıdaki gibi bir kural tanımlayabiliriz:

```
.cpp.o:
    g++ -c $<
```

Şimdi konuya biraz daha açıklık getirelim. Kaynak dosyamızın adı **helper.cpp** ve amacımız **helper.o** obje dosyasını üretmek olsun. Yukarıdaki kural kaynak dosyamız için çalıştığında `.cpp.o:` satırı yüzünden helper.cpp, oluşacak helper.o için bir bağımlılık durumunu alır. Bu nedenle `$<` değişkeni helper.cpp'yi gösterir. Bu sayede helper.o dosyası üretilmiş olacaktır.

Şimdi aynı mantıkla obje dosyalarından çalıştırılabilir programımızı üretelim.

```
.o:
    g++ $^ -o $@
```

Bu biraz daha karışık çünkü çalıştırılabilir dosyamızın uzantısı olmayacak. Eğer tek bir uzantı verilmiş ise bunun birinci uzantı olduğu ve ikincinin boş olduğu düşünülür.

Soyut kurallar tanımladığımızda yapmamız gereken iki işlem daha bulunur. Bunlardan birincisi kullandığımız uzantıların neler olduğunu belirtmektir. Bu işlem için `.SUFFIXES` özel değişkeni kullanılır:

```
.SUFFIXES: .cpp .o
```

Diğer yapmamız gereken işlem ise üretilecek çalıştırılabilir dosyamızın hangi obje dosyalarına, obje dosyalarımızın ise hangi kaynak dosyalarına bağımlı olduğunu belirtmek olacaktır. İşin en güç tarafı da budur. Her zaman doğru değerleri yazmak o kadar kolay olmayabilir. Bu noktada **gcc** derleyicisinin `-M`, **g++** derleyicisinin `-MM` seçenekleriyle bağımlılıkları Makefile dosya biçimine uygun şekilde hesaplatabiliriz. Aşağıdaki ekran çıktısına bakalım:

```
$ gcc -M server.c
server.o: server.c /usr/include/stdio.h /usr/include/features.h \
  /usr/include/x86_64-linux-gnu/bits/wordsize.h \
  /usr/lib/gcc/x86_64-linux-gnu/4.7/include/stddef.h \
  /usr/include/x86_64-linux-gnu/bits/types.h \
  /usr/lib/gcc/x86_64-linux-gnu/4.7/include/stdarg.h \
  /usr/include/x86_64-linux-gnu/bits/stdio_lim.h \
  /usr/include/x86_64-linux-gnu/bits/sys_errlist.h common.h list.h
```

Görüldüğü gibi **server.o** için gerekli Makefile kuralını bizim için hatasız olarak verdi. Tek yapmamız gereken bu satırları kopyalayıp Makefile içerisine yapıştırmaktır. Ancak bağımlılıklar hesaplandığında, esasen pek sık değişmeyen sistem kütüphaneleri içindeki referans gösterdiğimiz başlık dosyalarının da eklenmiş olduğunu görüyoruz. Makefile dosyasına her bir `.o` bağımlılık listesi için bunlara yazarsak dosya bizim için iyice okunmaz hale gelecek. O yüzden genel sistem başlık dosyalarını atlayarak, Makefile dosyamızı bu yöntemler eşliğinde soyut kurallarla yeniden yazmayı deneyelim.

Örnek: Soyut kuralların kullanıldığı Makefile

```
LIBS            = -lm \
                  -lrt \
                  -lpthread \
                  $(shell pkg-config --libs openssl)

INCLUDES        = -I/usr/local/include/custom

.SUFFIXES: .c .o

.c.o:
    $(CC) $(CFLAGS) $(INCLUDES) -c $<

.o:
    $(CC) $(CFLAGS) $(LIBS) $^ -o $@

all: server client

server: common.o server.o list.o
client: common.o client.o
common.o: common.c common.h
server.o: server.c server.h common.h list.h
client.o: client.c client.h ortak.h
list.o: list.c list.h

install: client server
    mkdir -p /usr/local/bin
    cp client /usr/local/bin/
    cp server /usr/local/bin/

uninstall:
    rm -f /usr/local/bin/client
    rm -f /usr/local/bin/server

clean:
    rm -f *.o server client

.PHONY: clean
```

Makro Kütüphaneleri Kullanımı

Önceki örneğimizde tüm bağımlılık kurallarını `gcc` 'ye hesaplattığımızda çıkan uzun listeden pek memnun olmadık. El yordamıyla içlerinden standart kütüphane başlık dosyalarını çıkarıyor olmanın pek uygulanabilir bir çözüm olmadığı ortada. Üstelik dosya sayısı arttıkça Makefile dosyamız içerisindeki karmaşa da artacaktır. Bu sorunları nasıl çözebiliriz?

Çözüm yolu, işleri bizim için kolaylaştıran Makefile makroları yazmaktan veya hazır yazılmış olanları kullanmaktan geçmektedir.

İnternet üzerinde çeşitli Makefile makro kütüphaneleri bulmanız mümkün. Bunlardan bizce fonksiyon seti / kullanım kolaylığı / maksimum fayda dengesinde en iyilerinden biri, **Alper Akcan** tarafından hazırlanmış olan `Makefile.lib` makro kütüphanesidir. Kütüphaneyi https://github.com/alperakcan/libmakefile adresinden indirebilirsiniz.

`Makefile.lib`, çapraz derleme işlemlerini `CROSS_COMPILE_PREFIX` değişkeninin ayarlanması suretiyle yönetebilmektedir. Ayrıca detaylı çıktı vermesi için make sistemlerinde alışageldiğimiz haliyle *verbose* opsiyonu `V` değişkeni üzerinden `V=1` şeklinde bir atama yapmak suretiyle aktifleştirilebilmektedir.

Bu şekilde yazılmış örnek bir Makefile dosyasına bakalım:

```
target-y  = target1
target-y += target2

target1_files-y = \
    target_file_shared.c \
    target1_file_2.c \
    target1_file_3.c

target1_includes-y = \
    ./ \
    /opt/include

target1_libraries-y = \
    ./ \
    /opt/lib

target1_cflags-y = \
    -DUSER_DEFINED \
    -O2

target1_ldflags-y = \
    -luserdefined

target2_files-y = \
    target_file_shared.c \
    target2_file_2.c \
    target2_file_3.c

target2_includes-y = \
    ./ \
    /opt/include

target2_libraries-y = \
    ./ \
    /opt/lib

target2_cflags-y = \
    -DUSER_DEFINED \
    -O2

target2_ldflags-y = \
    -luserdefined

include Makefile.lib
```

Projenin github sayfasından ve `Makefile.lib` dosya içeriğinden kullanımıyla ilgili yardım alabilirsiniz.

Makefile.lib kullandığınızda, tüm bağımlılıklar sistem kütüphaneleri de dahil olarak detaylı biçimde hesaplanır. Bağımlılıkların neler olduğu ve hesaplanmasında hangi komutun kullanıldığı gibi ek bilgiler, bulunulan dizinde nokta ile başlayan (dolayısıyla ön tanımlı `ls` komutunda listelenmeyen ve göz kalabalığı oluşturmayan) bir dizin yapısı altında saklanır. Derleme süreci daha sağlıklı ve temiz bir şekilde ilerler. Aşağıdaki Makefile.lib kullanılan bir projedeki derleme zamanı çıktıları görünmektedir:

```
$ make
  DEP         /home/demirten/embedded/gateway/.gateway/base64.dep
  DEP         /home/demirten/embedded/gateway/.gateway/backend.dep
  DEP         /home/demirten/embedded/gateway/.gateway/database.dep
  DEP         /home/demirten/embedded/gateway/.gateway/common.dep
  DEP         /home/demirten/embedded/gateway/.gateway/ini.dep
  DEP         /home/demirten/embedded/gateway/.gateway/gateway.dep
  CC          /home/demirten/embedded/gateway/.gateway/gateway.o
  CC          /home/demirten/embedded/gateway/.gateway/ini.o
  CC          /home/demirten/embedded/gateway/.gateway/common.o
  CC          /home/demirten/embedded/gateway/.gateway/database.o
  CC          /home/demirten/embedded/gateway/.gateway/backend.o
  CC          /home/demirten/embedded/gateway/.gateway/base64.o
  LINK        /home/demirten/embedded/gateway/.gateway/gateway
  CP          /home/demirten/embedded/gateway/gateway

$ ls .gateway/
backend.dep      backend.o.cmd   base64.o      common.dep.cmd
database.dep     database.o.cmd  gateway.dep   gateway.o.cmd
ini.o            backend.dep.cmd base64.dep    base64.o.cmd
common.o         database.dep.cmd  gateway.dep.cmd  ini.dep
ini.o.cmd        backend.o       base64.dep.cmd  common.dep
common.o.cmd     database.o      gateway.cmd     gateway.o
ini.dep.cmd
```

Görüldüğü üzere make uygulaması çalıştırıldığında önce bağımlılıklar hesaplanmış, hesaplama sonuçları ve kullanılan komutlar hedef uygulama ismi olan `gateway` 'den yola çıkılarak `.gateway` adıyla oluşturulan dizin altında toplanmış, tüm derleme işlemleri sonucu oluşan obje dosyaları da `.gateway` altında biriktirilmeye devam edilmiş ve işlem sonucunda ana dizinde `gateway` hedef uygulaması oluşturulmuştur.

Eğer derleme sürecinde daha detaylı ekran çıktısı almak istersek, `make V=1` şeklinde komutu çalışamız yeterli olacaktır.

Daha karmaşık bir Makefile.lib örneğine bakalım (gerçek zamanlı harita render kütüphanemizin demo uygulaması kısmından alınmıştır):

```
libmakefile Örnek 2

-include ${PLATFORM}.config
```

```
include Makefile.config

MOC ?= moc

target_o-y = \
    libnavigator.o

target_a-y = \
    libnavigator.a

target-$(ENABLE_NAVIGATOR_DEMO) = \
    navigator-demo

libnavigator.o_files-y = \
    tag.h \
    navigator.c \
    ../amalgamation/libamalgamation.o

libnavigator.a_files-y = \
    libnavigator.o \

ifeq (${COMMON_POINT_TYPE}, int)
libnavigator.o_cflags-y += \
    -DNAVIGATOR_COMMON_POINT_TYPE_INT=1
endif

ifeq (${COMMON_POINT_TYPE}, double)
libnavigator.o_cflags-y += \
    -DNAVIGATOR_COMMON_POINT_TYPE_DOUBLE=1
endif

ifeq (${COMMON_BOUND_TYPE}, int)
libnavigator.o_cflags-y += \
    -DNAVIGATOR_COMMON_BOUND_TYPE_INT=1
endif

ifeq (${COMMON_BOUND_TYPE}, double)
libnavigator.o_cflags-y += \
    -DNAVIGATOR_COMMON_BOUND_TYPE_DOUBLE=1
endif

navigator-demo_depends-y = \
    libnavigator.o

navigator-demo_files-y = \
    navigator-qt.cpp \
    navigator-qt-moc.cpp \
    libnavigator.o

navigator-qt-moc.cpp: navigator-qt.h
    ${Q}$(MOC) ${navigator-demo_cflags-y} navigator-qt.h -o navigator-qt-moc.cpp

navigator-demo_cflags-y = \
```

```
    $(shell pkg-config QtGui --cflags)

navigator-demo_cflags-${ENABLE_RENDER_CAIRO} += \
    -DNAVIGATOR_ENABLE_RENDER_CAIRO=1

navigator-demo_cflags-${ENABLE_RENDER_FLOATING_SCALE} += \
    -DNAVIGATOR_ENABLE_RENDER_FLOATING_SCALE=1

navigator-demo_ldflags-y = \
    $(shell echo $(CC) | awk '{ if (/mingw/) { print "-Wl,-Bstatic -static-libgcc -L/m
ingw/Qt/lib -lmman"; } }') \
    $(shell pkg-config freetype2 --libs) \
    -lexpat \
    -lm \
    -lz \
    $(shell pkg-config QtGui --libs) \
    -lpthread

navigator-demo_ldflags-${ENABLE_RENDER_CAIRO} += \
    $(shell pkg-config cairo --libs)

navigator-demo_ldflags-${ENABLE_COMPRESS_SNAPPY} += \
    -lsnappy

navigator-demo_ldflags-${ENABLE_INPUT_TIF} += \
    -lgeotiff \
    -ltiff

navigator-demo_ldflags-${ENABLE_RENDER_PNG} += \
    $(shell pkg-config libpng --libs)

navigator-demo_ldflags-${ENABLE_RENDER_JPEG} += \
    -ljpeg

include Makefile.lib
```

İlk 2 satıra dikkatlice tekrar bakalım:

```
-include ${PLATFORM}.config
include Makefile.config
```

Satır başında yer alan **-** karakteri, ilgili dosya *include* edilmek için arandığında dizinde yer almıyorsa **make** sürecinin hata vermeyip yoluna devam etmesini sağlamak için konulmuştur. Eğer `make` çalıştırılırken `PLATFORM` değişkenine atama yapılırsa, öncelikle ilgili dosya *include* edilecektir.

Ardından her koşulda `Makefile.config` dosyasının *include* edildiğini görmekteyiz.

Şimdi **PLATFORM** değişkenin **Debian** olarak atandığı örnek bir `make` kullanımını ve `Debian.config` ile `Makefile.config` dosyalarının içeriklerini görelim:

```
$ make ENABLE_INPUT_OSM=n PLATFORM=Debian -j 8
```

Yukarıdaki kullanımda öncelikle `ENABLE_INPUT_OSM` değişkeninin değeri `n` olarak, `PLATFORM` değişkenin değeri `Debian` olarak atanmakta ve **8** paralel derleme sürecine imkan verecek şekilde uygulama başlatılmaktadır.

```
# Debian.config
ENABLE_NAVIGATOR_DEMO        ?= y

ENABLE_RENDER_COPYRIGHT      ?= y
RENDER_COPYRIGHT_COLOR       ?= 0x20ddbbcc

ENABLE_COMMON_CLIPPER        ?= y
COMMON_FILE_CACHE_SIZE       ?= 0x1400000

ENABLE_INPUT_TIF             ?= y
ENABLE_INPUT_OSM             ?= y
ENABLE_INPUT_JPEG_TURBO      ?= n
ENABLE_INPUT_TILE            ?= y

ENABLE_RENDER_CAIRO          ?= n
ENABLE_RENDER_PNG            ?= y
ENABLE_RENDER_JPEG           ?= y
ENABLE_RENDER_JPEG_TURBO     ?= n
ENABLE_RENDER_TEXT_ON_PATH   ?= y

ENABLE_COMPRESS_SNAPPY       ?= n

ENABLE_RENDER_FLOATING_SCALE ?= y

ENABLE_EXTERNAL_LOGGER       ?= n

ENABLE_DEBUGF                ?= n
ENABLE_ERRORF                ?= y
ENABLE_INFOF                 ?= y
ENABLE_TODOF                 ?= n
ENABLE_ASSERTF               ?= n

COMMON_POINT_TYPE            ?= int
COMMON_BOUND_TYPE            ?= int
```

```
# Makefile.config

ENABLE_NAVIGATOR_DEMO        ?= y

ENABLE_RENDER_COPYRIGHT      ?= y
RENDER_COPYRIGHT_COLOR       ?= 0x202a77a3

ENABLE_COMMON_CLIPPER        ?= y
COMMON_FILE_CACHE_SIZE       ?= 0x2000000

ENABLE_INPUT_TIF             ?= y
ENABLE_INPUT_OSM             ?= y
ENABLE_INPUT_JPEG_TURBO      ?= y
ENABLE_INPUT_TILE            ?= y

ENABLE_RENDER_CAIRO          ?= n
ENABLE_RENDER_PNG            ?= n
ENABLE_RENDER_JPEG           ?= y
ENABLE_RENDER_JPEG_TURBO     ?= y
ENABLE_RENDER_TEXT_ON_PATH   ?= y

ENABLE_COMPRESS_SNAPPY       ?= y

ENABLE_RENDER_FLOATING_SCALE ?= y

ENABLE_EXTERNAL_LOGGER       ?= n

ENABLE_DEBUGF                ?= n
ENABLE_ERRORF                ?= y
ENABLE_INFOF                 ?= y
ENABLE_TODOF                 ?= n
ENABLE_ASSERTF               ?= y

COMMON_POINT_TYPE            ?= double
COMMON_BOUND_TYPE            ?= double
```

Bu şekildeki konfigürasyon dosyaları yardımıyla, çeşitli değişkenlerin hem öntanımlı değerlerini atayabilmekte, hem de kullanıcı tarafından özellikle belirtilmiş ise, ilgili değeri kullanabilmekteyiz. Bunun için `?=` tanımından faydalanıyoruz. Bu tanım Makefile içerisinde, **eğer değişkene atama yapılmamış ise → ata** şeklinde işlev görmektedir

Bu yapı ile farklı konfigürasyon dosyaları kullanılabildiği gibi, konfigürasyon dosyası içerisinde `?=` şeklinde tanımlanmış değişkenleri aşağıdaki şekilde ezmek de mümkün olmaktadır:

```
$ make ENABLE_DEBUGF=y PLATFORM=Debian
```

Yukarıdaki komut, bir önceki Makefile örneğiyle birlikte değerlendirildiğinde, Debian.config dosyası içerisinde yer alan `ENABLE_DEBUGF ?= n` şeklindeki satırın geçersiz olmasını sağlayacaktır

Eğer Debian.config içerisinde bu tanım `ENABLE_DEBUGF = n` şeklinde doğrudan eşittir karakteri ile yapılmış olsaydı, öncesinde atanan değerden bağımsız olarak bu konfigürasyon dosyası işlendiğinde her zaman dosyanın içindeki atama geçerli olacaktı. Buradaki küçük detaylar dikkatle kullanıldığında, aynı kod üzerinden farklı konfigürasyonlarda derleme işleminiz kolaylaşacaktır.

Make Alternatifleri

CMake

CMake, birden çok platformu destekleyen (Linux, Apple, Windows) güçlü bir inşa aracıdır.

Geliştirilmesi büyük ölçüde **Kitware** firması tarafından yapılmaktadır.

CMake kendi kural dosyalarını işleyerek, hangi platformda çalışıyorsa o platform için doğal inşa sistemine ait kural dosyaları oluşturur (*NIX sistemler için Makefile).

Aşağıdaki örnek CMake dosyasını inceleyiniz:

```
if (${UNIX})
  set (DESKTOP $ENV{HOME})
else()
  set (DESKTOP $ENV{USERPROFILE}/Desktop)
endif()

set  (PRJ      ${DESKTOP}/common/svn )
set  (FILELIST ${PRJ}/src/source.txt )

message(STATUS "CMAKE_GENERATOR : ${CMAKE_GENERATOR}")
message(STATUS "DESKTOP         : ${DESKTOP}")
message(STATUS "PRJ             : ${PRJ}")
message(STATUS "FILELIST        : ${FILELIST}")
message(STATUS "SYSTEM_NAME     : ${CMAKE_SYSTEM_NAME}")

project(project_name)

include_directories(
  ${PRJ}/src
  ${PRJ}/includes
)

# Load SRC Variable from file
file(READ ${FILELIST} SRC)
string(REGEX REPLACE  "#.*$"  ""  SRC  ${SRC})
string(REPLACE        "\n"    ";" SRC  ${SRC})

add_executable(${PROJECT_NAME} ${SRC} )

foreach (f ${SRC})
  set_source_files_properties(${f} PROPERTIES LANGUAGE       CXX)
endforeach(f)

if (${WIN32})
  link_directories(
  )

  add_definitions(
    -DDEFINE1
  )

  target_link_libraries(
    ${PROJECT_NAME}
    wsock32.lib
  )
endif()
```

SCons

SCons, Python dili ile geliştirilmiş, birden çok platformu destekleyen diğer bir inşa aracıdır.

Konfigürasyon betikleri Python dosyalarından oluşur

C, C++ ve Fortran için doğrudan kod bağımlılık analizi desteği sunar.

Versiyon kontrol sistemlerine doğrudan destek verir (SCCS, RCS, CVS, Subversion, BitKeeper, Perforce).

Dosyaların değişimindeki kontroller son değiştirilme tarihi yerine **MD5SUM** değerleri üzerinden yapılır. Parametre vererek dosyanın değiştirilme tarihine bakacak hale de getirmek mümkündür.

Aşağıdaki örnek SCons dosyasını inceleyebilirsiniz:

```
env = Environment()
env.Append(CPPFLAGS=['-Wall','-g'])
env.Program('hello', ['hello.c', 'main.c'])
```

Rake

Ruby ile geliştirilmiş ve daha çok Ruby projelerinde kullanılan bir inşa aracıdır.

Ruby'nin DSL tanımlama noktasındaki güçlü özelliklerini kullanır.

Kurallar `Rakefile` dosyalarında tutulur.

Rakefile içerisinde Ruby dilinde görevler ve kurallar tanımlanabildiği gibi, dilinden kendisinden gelen ekstra özelliklerle örneğin çalışma zamanında yeni sınıflar dahi üretilebilir.

Aşağıdaki örnek Rakefile dosyasını inceleyebilirsiniz:

```
namespace :cake do
  desc 'make pancakes'
  task :pancake => [:flour,:milk,:egg,:baking_powder] do
     puts "sizzle"
  end
  task :butter do
    puts "cut 3 tablespoons of butter into tiny squares"
  end
  task :flour => :butter do
    puts "use hands to knead butter squares into 1{{frac|1|2}} cup flour"
  end
  task :milk do
    puts "add 1{{frac|1|4}} cup milk"
  end
  task :egg do
   puts "add 1 egg"
  end
  task :baking_powder do
   puts "add 3{{frac|1|2}} teaspoons baking powder"
  end
end
```

autoconf, automake Kullanımı

GNU build sistemi iki temel amacın gerçekleştirilebilmesi için geliştirilmiştir: Programları platformlar arası daha rahat taşınabilir hale getirmek ve kaynak koddan program kurulumlarını mümkün olduğu kadar basite indirgeyebilmek.

Taşınabilir kod yazmak gerçekten oldukça zahmetli bir iştir. Hedef mimarinin ayrıntılı olarak özelliklerinin bilinmesi çoğu zaman mümkün değildir. Bir önceki bölümde örnek olarak yazdığımız Makefile dosyasında `mkdir -p /usr/local/bin` komutunu kullanmıştık. Oysa `mkdir` komutunun `-p` seçeneği tüm Unix sistemlerinde aynı şekilde çalışmaz. Bu ve bunun gibi pek çok farklılık yüzünden her Unix sisteminde çalışabilecek bir `Makefile` yazmak oldukça güçtür. Kullanılan kütüphanelerin sistemler arasındaki farklılıkları ise apayrı bir konudur. İşte GNU build sistemi tüm bu zorlukların üstesinden gelebilmek için oluşturulmuştur.

Kdevelop gibi programlar yeni proje oluşturduğunuzda build sistemini de otomatik olarak oluşturmaktadırlar. Ancak oluşan dosyalar fazlasıyla karışık olduğundan bu bölümde çok daha basit örneklerle yapıyı anlatmaya çalışacağız. Buradaki temel bilgilerden yararlandıktan sonra Kdevelop vb. gibi programların ürettiği veya internetten indirmiş olduğunuz herhangi bir uygulamanın kaynak kodu içerisinde gezinerek farklı kullanımları inceleyebilirsiniz.

Gerekli Araçlar

Gnu build sistemi için gerekli araçlar ve kullanım alanları aşağıdaki gibidir:

1. **autoconf**: konfigürasyon için kullanılacak configure betik programını üretir. Kodun taşınabilir olmasını etkileyecek özellikleri, üzerinde çalıştığı platform için denetler. Elde ettiği değerleri, daha önceden belirtilmiş şablonlara uygun şekilde birleştirerek özelleştirilmiş Makefile, başlık dosyaları vb. oluşturur. Bu sayede programı derleyecek kullanıcı tek tek elle bu değişiklikleri yapmak zahmetinden kurtulur.
2. **automake**:, autoconf için kullanılacak Makefile şablonlarını (Makefile.in) Makefile.am dosyalarını temel alarak üretir. Automake tarafından üretilen Makefile dosyaları GNU makefile standartlarına uygun olup, kullanıcıyı elle Makefile dosyası oluşturma zahmetinden kurtarır. Autoconf'un çalışabilmesi için öncelikle automake'in düzgün olarak çalışması gereklidir.

3. **libtool**: özellikle paylaşımlı kütüphanelerin taşınabilir bir yapıda oluşturulabilmesi için gereken pek çok detayı kullanıcıdan soyutlar. Kullanımı için autoconf veya automake gerekli değildir, tek başına da kullanılabilir. Automake ise libtool'u destekler ve onunla birlikte çalışabilir.
4. **autotools**: GNU kodlama standartlarına uygun, taşınabilir kod üretmede yardımcı araçlar içeriz.

GNU build sistemi tarafından gerçekleştirilen temel görevler şunlardır:

1. Çok sayıda alt dizin içeren kaynak kodlardan uygulamaları üretebilir. Her bir dizin için ayrıca make komutunu çağırmak zahmetinden geliştiriciyi kurtarır. Bu sayede tüm kaynak kodları aynı dizinde bulundurmak yerine daha hiyerarşik bir dizin yapısı kullanabilirsiniz.
2. Konfigürasyon işlemini otomatik olarak yapar. Kullanıcıların Makefile dosyalarını düzenlemelerine gerek kalmaz.
3. Makefile dosyalarını otomatik olarak üretir. Makefile yazımı büyük projelerde sürekli tekrar gerektirir ve aynı zamanda hata yapmaya elverişli bir yapıdır. GNU build sistemi için sadece `Makefile.am` şablonunun yazımı yeterlidir. Bu sayede hata yapma olasılığı azalır ve yönetimi kolay hale gelir.
4. Hedef platform için özel testler yapabilme imkanı sunar. Makefile.am dosyasına eklenecek bir kaç satırla hedef platformda programın derlenebilesi için aranan özelliklerin var olup olmadığı kontrol edilebilir.
5. Paylaşımlı kütüphanelerin oluşturulması statik kütüphanelerin oluşturulması kadar kolay hale gelir.

GNU build sistemi için gerekli olan bu araçların sadece geliştirmenin yapıldığı sistemde kurulu olması yeterlidir. Bu programlar çalıştıktan sonra her platformda çalışabilecek betik programları üretirler. Bu sayede uygulamanızın kaynak kodunu indirip kurmak isteyen biri, `autoconf`, `automake` gibi araçları da sistemine kurmak zorunda kalmaz.

İşlemlere başlamadan önce `autoconf`, `automake` ve `libtool` paketlerini sistemimize kuralım:

```
$ sudo apt-get install autoconf automake libtool
```

Zorunlu olmamakla birlikte, `configure.ac` dosyalarımız içerisinde temel `M4` makro seti içerisinde yer almayan ancak muhtemelen kullanmak zorunda kalacağımız ek makro arşivini içeren `autoconf-archive` paketini de kurmamız yerinde olacaktır:

```
$ sudo apt-get install autoconf-archive
```

Örnek Makro: `AX_FUNC_MKDIR`

`mkdir()` fonksiyonu bazı platformlarda `_mkdir()` şeklinde kullanılmaktadır.

Bazı platfomlarda dizin erişim yetkileri için 2. argüman kullanılıyorken diğerlerinde tek argüman kullanıldığı bilinmektedir.

Aşağıkda `mkdir()` fonksiyonunun platformlar arası taşınabilirliğini bizim için kolaylaştıran `AX_FUNC_MKDIR` makro örneği verilmiştir:

```
#if HAVE_MKDIR
#  if MKDIR_TAKES_ONE_ARG
     /* MinGW32 */
#    define mkdir(a, b) mkdir(a)
#  endif
#else
#  if HAVE__MKDIR
     /* plain Windows 32 */
#    define mkdir(a, b) _mkdir(a)
#  else
#    error "Don't know how to create a directory on this system."
#  endif
#endif
```

Örnek Uygulama

Şimdi aşağıdaki örnek uygulamamız için GNU build sistemini nasıl kullanacağımızı öğrenelim.

```
#include <stdio.h>
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

int main (int argc, char *argv[])
{
        printf("Örnek Çalışıyor\n");

        if (argc == 2) {
                const char *target = argv[1];
                int rc;
#ifdef HAVE_MKDIR
#ifdef MKDIR_TAKES_ONE_ARG
                rc = mkdir(target);
#else
                rc = mkdir(target, 0755);
#endif
                if (rc == 0) {
                        printf("%s dizini oluşturuldu\n", target);
                } else {
                        fprintf(stderr, "%s dizini oluşturulamadı\n", target);
                }
#else
                fprintf(sdterr, "mkdir() desteklenmiyor\n");
#endif
        }

    return 0;
}
```

Programımızı **ornek.c** olarak kaydedelim. Şimdi programın derlenmesi işlemlerini autoconf ve automake ile yapmaya başlayalım. Bunun için öncelikle `automake` ile kullanılmak üzere aşağıdaki gibi bir `Makefile.am` dosyasını oluşturalım:

```
AUTOMAKE_OPTIONS = foreign
bin_PROGRAMS = ornek
ornek_SOURCES = ornek.c
```

`foreign` opsiyonu sayesinde ilerleyen aşamalarda GNU uygulamalarında bulunması zorunlu dosya kontrollerini devre dışı bırakıyoruz (Bu seçeneği kullanmadığımız durumdaki senaryoyu test ediniz)

Ardından aşağıdaki gibi bir `configure.ac` dosyasını oluşturalım (eski versiyonlarda dosya ismi `configure.in` şeklindeydi):

```
AC_INIT([ornek], [0.1], [ornek-bugs@yh.com.tr])
AC_CONFIG_SRCDIR([ornek.c])
AM_INIT_AUTOMAKE

AC_CONFIG_HEADERS([config.h])

AC_PROG_CC

AC_CHECK_HEADERS([sys/time.h])

AC_CHECK_FUNCS([gettimeofday])

AX_FUNC_MKDIR

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Dosyaları oluşturup kaydettikten sonra şimdi `aclocal` komutunu çalıştıralım:

```
$ aclocal
$ ls -l
-rw-r--r-- 1 demirten demirten 41928 Dec 27 01:04 aclocal.m4
drwxr-xr-x 2 demirten demirten  4096 Dec 27 01:04 autom4te.cache
-rw-r--r-- 1 demirten demirten   120 Dec 27 01:03 configure.ac
-rw-r--r-- 1 demirten demirten    45 Dec 27 00:51 Makefile.am
-rw-r--r-- 1 demirten demirten    87 Dec 27 00:26 ornek.c
```

`aclocal` çalıştıktan sonra bulunduğumuz dizinde `autom4te.cache` dizini ve `aclocal.m4` dosyası oluştu.

Sonraki adımda autoheader komutunu çalıştırıyoruz. Eğer `configure.ac` dosyasında `AC_CONFIG_HEADERS` belirtilmemişse yapılacak kontroller sonrası elde edilen değerler `config.h` dosyasına yazılmayacak olduğundan bu adımı atlayabilirsiniz:

```
$ autoheader
$ ls -l config.h.in
-rw-r--r-- 1 demirten demirten 1825 Dec 27 02:43 config.h.in
```

İşlem bitiminde `config.h` dosyası için kullanılacak `config.h.in` şablonu üretildi.

Sıradaki işlem autoconf komutuyla configure betiğini oluşturmak:

```
$ autoconf
$ ls -l configure
-rwxr-xr-x 1 demirten demirten 140829 Dec 27 01:04 configure
```

İşlem sonrasında bulununan dizinde `configure` adlı tanıdık betik uygulamasını oluşturmuş oluyoruz.

`configure` betiği de oluştuktan sonra `automake -a` komutuyla gereken diğer dosyaları oluşturuyoruz:

```
$ automake -a
configure.ac:4: installing './compile'
configure.ac:3: installing './install-sh'
configure.ac:3: installing './missing'
Makefile.am: installing './depcomp'
```

Yazılımlarda olması beklenen bazı temel dosyaların (ChangeLog, README vb.) çalışma dizinimizde bulunmaması nedeniyle uyarı mesajlarını görmekteyiz ama şu aşamada bunu dikkate almamıza gerek yok.

`automake -a` komutu sonrası dizin içeriğimiz aşağıdaki hale gelmektedir:

```
$ ls
aclocal.m4
autom4te.cache
compile -> /usr/share/automake-1.14/compile
config.h.in
configure
configure.ac
depcomp -> /usr/share/automake-1.14/depcomp
install-sh -> /usr/share/automake-1.14/install-sh
Makefile.am
Makefile.in
missing -> /usr/share/automake-1.14/missing
ornek.c
```

Bu haliyle `./configure` şeklinde betik uygulamamızı çalıştırıp sonucunu görelim:

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking whether gcc understands -c and -o together... yes
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking how to run the C preprocessor... gcc -E
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking sys/time.h usability... yes
checking sys/time.h presence... yes
checking for sys/time.h... yes
checking for gettimeofday... yes
checking for mkdir... yes
checking for _mkdir... no
checking whether mkdir takes one argument... no
checking that generated files are newer than configure... done
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
config.status: executing depfiles commands
```

İşlem bitiminde hem `Makefile` hem de `config.h` dosyası oluşacaktır:

```
/* config.h.  Generated from config.h.in by configure.  */
/* config.h.in.  Generated from configure.ac by autoheader.  */

/* Define to 1 if you have the `gettimeofday' function. */
#define HAVE_GETTIMEOFDAY 1

/* Define to 1 if you have the <inttypes.h> header file. */
#define HAVE_INTTYPES_H 1

/* Define to 1 if you have the `mkdir' function. */
#define HAVE_MKDIR 1

/* Define to 1 if you have the <sys/types.h> header file. */
#define HAVE_SYS_TYPES_H 1

/* Define to 1 if you have the <unistd.h> header file. */
#define HAVE_UNISTD_H 1

/* Define to 1 if you have the `_mkdir' function. */
/* #undef HAVE__MKDIR */

/* Define if mkdir takes only one argument. */
/* #undef MKDIR_TAKES_ONE_ARG */

/* Name of package */
#define PACKAGE "ornek"

/* Define to the address where bug reports for this package should be sent. */
#define PACKAGE_BUGREPORT "ornek-bugs@yh.com.tr"

/* Define to the full name of this package. */
#define PACKAGE_NAME "ornek"

/* Define to the full name and version of this package. */
#define PACKAGE_STRING "ornek 0.1"

/* Define to the one symbol short name of this package. */
#define PACKAGE_TARNAME "ornek"

/* Define to the home page for this package. */
#define PACKAGE_URL ""

/* Define to the version of this package. */
#define PACKAGE_VERSION "0.1"

/* Define to 1 if you have the ANSI C header files. */
#define STDC_HEADERS 1

/* Version number of package */
#define VERSION "0.1"
```

Artık Makefile dosyamız hazır olduğunda göre, `make` komutu ile derleme işlemini yapabiliriz:

```
$ make
make  all-am
make[1]: Entering directory /tmp/deneme
gcc -DHAVE_CONFIG_H -I. -g -O2 -MT ornek.o -MD -MP -MF \
    .deps/ornek.Tpo -c -o ornek.o ornek.c
mv -f .deps/ornek.Tpo .deps/ornek.Po
gcc  -g -O2   -o ornek ornek.o
make[1]: Leaving directory /tmp/deneme
```

Uygulamamızı çalıştıralım:

```
$ ./ornek dizin1
Örnek Çalışıyor
dizin1 dizini oluşturuldu
```

Uygulamanın Kurulumu

autotools bileşenleriyle yaptığımız bu çalışma, uygulamanın sadece derlenmesi sürecini değil, kurulum ve sistemden kaldırma süreçlerini de desteklemektedir. Bu noktada dilerseniz uygulamayı sisteme kurabilirsiniz. Bunun için uygulamanın kurulacağı dizinler üzerinde yazma yetkinizin bulunması gerekecektir.

Uygulamayı sistemimize kurmak için:

```
$ sudo make install
make[1]: Entering directory /tmp/deneme
/bin/mkdir -p /usr/local/bin
/usr/bin/install -c ornek /usr/local/bin
make[1]: Nothing to be done for install-data-am.
make[1]: Leaving directory /tmp/deneme
```

Sistemden kaldırmak için:

```
$ sudo make uninstall
( cd /usr/local/bin && rm -f ornek )
```

`autoreconf`

Zaman içerisinde `Makefile.am` veya `configure.ac` dosyalarında değişiklik yaptığınızda, autotools araçlarını **doğru sırada** yeniden çalıştırmanız gerekecektir.

Alternatif olarak, bu işlemi kolaylaştıran `autoreconf` komutunu `-vfi` parametresi ile kullanabiliriz. Bu şekilde değişiklik var ise gerekli bileşenler doğru sırada çalışacaktır:

```
$ autoreconf -vfi
autoreconf: Entering directory `.'
autoreconf: configure.ac: not using Gettext
autoreconf: running: aclocal --force
autoreconf: configure.ac: tracing
autoreconf: configure.ac: not using Libtool
autoreconf: running: /usr/bin/autoconf --force
autoreconf: running: /usr/bin/autoheader --force
autoreconf: running: automake --add-missing --copy --force-missing
```

Uygulamanın Dağıtıma Hazırlanması

Uygulamanızın artık hazır olduğunu düşündüğünüzde `make distcheck` komutu ile onu paket haline getirebilirsiniz (Ekran çıktısı biraz uzun olduğundan burada listelenmemiştir). Bu komut işini tamamladığında bulunduğunuz dizinde `ornek-0.1.tar.gz` adında bir dosya oluşacaktır. Artık bu dosya ile programınızın dağıtımını yapabilirsiniz.

Şimdi biraz da yaptığımız bu örneği biraz daha açıklayalım.

- Makefile.am içerisinde mantıksal bir dil kullandık. Yazdığımız hiç bir satır çalıştırılmadı.
- Diğer yandan configure.in içerisinde kullandığımız dil prosedüreldir, yazdığımız her satır çalıştırılacak bir komutu göstermektedir.
- Makefile.am dosyası içerisindeki ilk satır programın ismini belirtirken ikinci satır programı oluşturan kaynak kodları belirtmektedir.

Şimdi daha karışık olan configure.in içerisindeki komutlara sırasıyla bakalım:

- **AC_INIT** komutu configure betiği için ilklendirmeleri yapar. Parametre olarak kaynak dosyaların adlarını alır.
- **AM_INIT_AUTOMAKE** komutu, automake kullanacağımızı gösterir. Parametre olarak programın ismini ve versiyonunu alır. Eğer Makefile.in dosyalarını elle hazırlayacak olsaydık bu komutu kullanmamıza da gerek olmayacaktı.
- **AC_PROG_CC** komutu kullanılan C derleyicisinin ne olduğunu belirler.
- **AC_PROG_INSTALL** komutu BSD uyumlu install uygulamasına sahip olup olmadığımızı denetler. Eğer yoksa bu işlem için install-sh'ı kullanır.
- **AC_OUTPUT** komutu configure betik programının Makefile dosyalarını Makefile.in dosyalarından üretmesi gerektiğini belirtir.

Örneğimizdeki configure.in dosyası içerisinde yer almayan ama sıklıkla kullanacağımız bazı komutlar da şunlardır:

- **AC_PROG_RANLIB** komutuyla bir kütüphane geliştiriyorsak ranlib'in sistemde nasıl kullanılacağını öğrenebiliriz.
- **AC_PROG_CXX** komutuyla sistemdeki C++ derleyicisinin ne olduğunu öğrenebiliriz.
- **AC_PROG_YACC** ve **AC_PROG_LEX** komutlarıyla kaynak kodlarımız lex veya yacc dosyaları içeriyorsa bu uygulamaların sistemde varlığını denetleyebiliriz.

Eğer alt dizinlerde başka Makefile dosyalarımız da olacaksa bunu

```
AC_OUTPUT(Makefile         \
          dizin1/Makefile  \
          dizin2/Makefile  \
          )
```

komutlarıyla belirtebiliriz.

Dosyaların içeriğinden bahsettikten sonra şimdi de biraz önc yaptığımız örnekte çalıştırdığımız komutlardan sonra neler olduğuna tekrar bakalım.

- `aclocal` komutu çalıştıktan sonra aclocal.m4 dosyası üretilir. Bu dosya içerisinde autoconf tarafından kullanılacak olan makrolar yer almaktadır (kendi özel makrolarımızı nasıl hazırlayacağımıza ileride değinilecektir).
- `autoconf` komutuyla aclocal.m4 ve configure.ac dosyaları işlenerek configure betik programı oluşturulur.
- `automake` komutu Makefile.am dosyasını temel alan bir Makefile.in oluşturur. Ayrıca GNU kodlama standartlarına göre eksik olan dosyalar için örnek birer kopya üretir.
- `./configure` komutuyla çalıştırılan betik programı daha önceden belirtilen özellikler için sistemimizi test eder ve Makefile.in dosyasını örnek alarak Makefile dosyalarını oluşturur. AC_OUTPUT() ile belirtilen tüm dosyalardaki @FOO@ şeklindeki kayıtları FOO için elde edilen değerlerle değiştirir (örneğin C derleyicisinin ne olduğu gibi).

Konfigürasyon Başlık Dosyalarının Kullanımı

Çoğu zaman derleme anında bazı makrolar tanımlamak isteriz. `-D` seçeneği ile derleyiciye bildirilen bu değerleri programımız içerisinden kullanarak ilgili kod parçacığının çalışma şeklini değiştirebiliriz. autoconf kullandığımız bir uygulama için böylesi seçenekleri kullanmanın yolu konfigürasyon başlık dosyası, `config.h` kullanmaktan geçmektedir.

`config.h` mantığını kullanabilmemiz için test.c programımızın en başına aşağıdaki üç satırı eklemeliyiz:

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
```

Burada unutulmaması gereken önemli bir nokta, config.h dosyasının mutlaka ilk olarak include edilmesidir.

Program kaynak kodunu bu şekilde değiştirdikten sonra `configure.ac` dosyasına `AC_CONFIG_HEADERS([config.h])` satırını ve kontrol etmek istediğimiz ilgili makroları eklemeliyiz.

Automake Değişkenleri

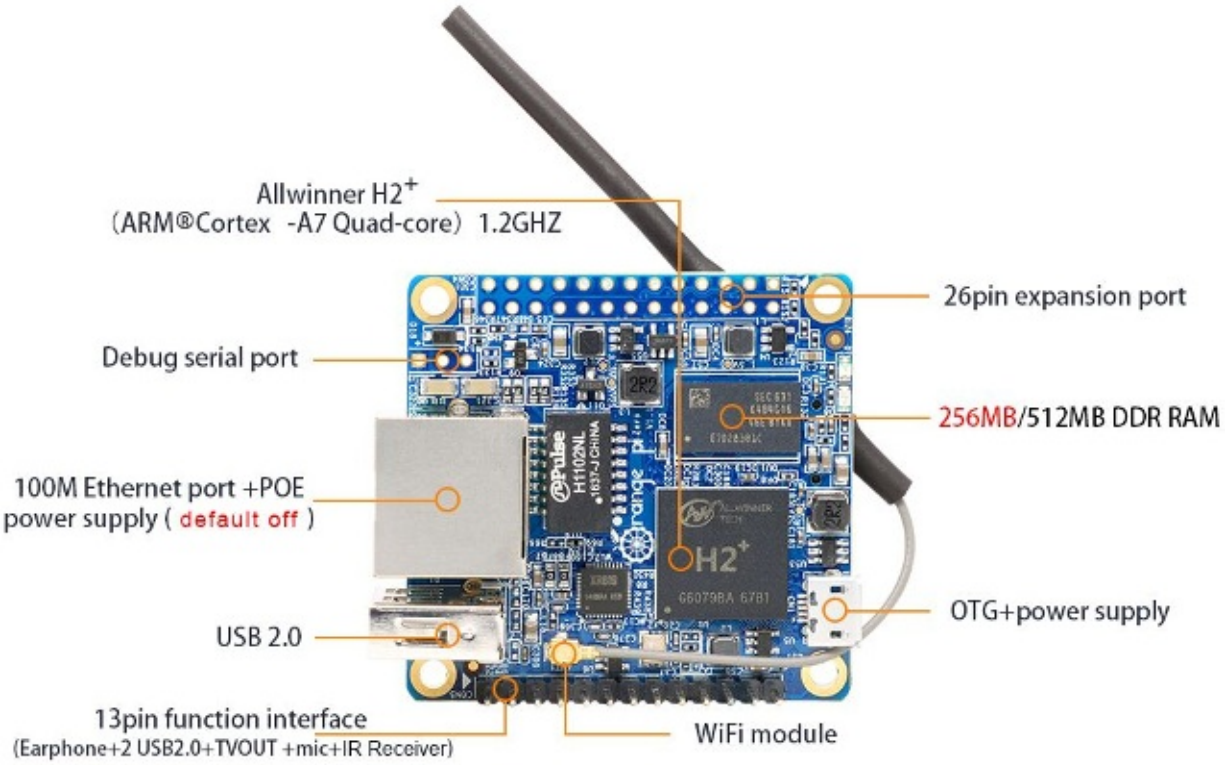
Projemiz büyüdükçe kaynak kodların bulunduğu yerler gittikçe karışmaya başlar. Bunları düzenleyebilmek amacıyla daha hiyerarşik dizin yapıları kurarız. Ancak bu defa da her bir dizin için uygun Makefile dosyalarını üretmemiz gerekecektir. Bunun için bu bölümde Makefile.am dosyalarından daha detaylı bir şekilde bahsedeceğiz.

Makefile.am dosyalarının genel biçimi `değişken = değer` şeklindedir. Ancak aynı zamanda, geleneksel Makefile mantığındaki gibi hedef ve soyut kural tanımlamalarını da destekler. Şimdi Makefile.am dosyalarında sıklıkla kullanacağımız satırlara bir bakalım.

- **INCLUDES = -I/usr/local/include -I/usr/custom/include ...:** Obje kodlarını oluştururken derleyiciye include edilen dosyaları hangi dizinlerde araması gerektiğini belirtir. Ayrıca proje kaynak kod yapısı içerisindeki bir dizin seçenek olarak verildi ise alt dizinlerde yer alan kaynak programların hepsinin bu dosyalara erişebilmelerini sağlamak amacıyla tanımlama **INCLUDES = -I$(top_srcdir)/src/libxxx** şeklinde yapılmalıdır. Buradaki `$top_srcdir` değişkeni kaynak kod yapısı içerisindeki en üst dizini tutar.
- **LDFLAGS = -L/usr/local/lib ...:** Derleyici çalıştırılabilir dosyaları üretirken ihtiyaç duyduğu kütüphaneleri hangi dizinlerde araması gerektiğini bu tanımla öğrenecektir.
- **LDADD = test.o ... $(top_builddir)/lib/libfoo.a ... -lfoo ...:** Tüm oluşacak çalıştırılabilir dosyalara linklemek istediğiniz sisteme kurulu olan ve olmayan obje dosyaları burada listelenir. Eğer listelenen obje dosyası sistemde kurulu değilse dosyanı tam adresi verilmelidir ($top_builddir/lib/libfoo.a örneğindeki gibi).
- **EXTRA_DIST = dosya1 dosya2 ...:** Kaynak kod paketinizde bulunmasını istediğiniz her türlü dosyayı burada listeleyebilirsiniz.

- **SUBDIRS = dizin1 dizin2 ...:** Bulunulan dizin için işlem yapmadan önce kuralların çalıştırılması gereken dizinlerdir. make uygulaması bulunulan dizinde işleme başlamadan önce, burada belirtilen dizinlerdeki Makefile kurallarını çalıştırır ve listelenen tüm dizinler için işlemleri bitirdikten sonra bu dizine geri döner.
- **bin_PROGRAMS = test test2 ...:** make komutu çalıştıktan sonra üretilecek ve make install komutuyla belirli bir dizin altına kopyalanacak program adları burada listelenir.
- **lib_LIBRARIES = libfoo1.a libfoo2.a ...:** make komutu çalıştıktan sonra üretilecek ve make install komutuyla belirli bir dizin altına kopyalanacak kütüphane dosyalarının adları burada listelenir.
- **check_PROGRAMS = program1 program2 ... :** make komutunun çalışması esnasında üretilmeyip, sadece make check komutuyla üretilecek, programınızın tümünü veya bir kısmını test edecek uygulamaların çalıştırılabilir dosya adları listelenir.
- **TESTS = program1 program2 ...:** make check komutu sonrasında test amaçlı çalıştırılacak dosya adlarını listeler. Çoğu durumda **TEST = $(check_PROGRAMS)** şeklinde bir tanımlama yapabilirsiniz.
- **include_HEADERS = foo1.h foo2.h ... :** /prefix/include dizini altına kurulmasını istediğiniz başlık dosyalarını burada listelemelisiniz.
- **bin_PROGRAMS** değişkeninde listelediğiniz her bir program için aşağıdaki tanımlamaları da yapmalısınız (program kelimesi yerine programın adını yazmalısınız):
- **program_SOURCES = test.c test1.c test2.c test.h test1.h test2.h ...:** Automake programı burada belirtmiş olduğunuz dosya adları için, C, C++ ve Fortran dillerine özgün soyut Makefile kurallarını oluşturur. Eğer başka bir dil kullanılıyorsa gerekli kuralları siz vermelisiniz.
- **program_LDADD = $(top_builddir)/lib/libfoo.a -lnsl ...:** Burada programınıza linklenmesi gereken kütüphaneleri listelemelisiniz.
- **program_LDFLAGS = -L/dizin1 ...:** program_LDADD ile belirttiğiniz kütüphanelerin hangi dizinlerde aranması gerektiği burada listelenir.
- **program_DEPENDENCIES = dep1 dep2 ... :** Programınızın derlenebilmesi için bağımlı olduğu diğer hedefleri burada listelemelisiniz.

Orange Pi Zero



Orange Pi Zero, düşük maliyetli, 256 ve 512MB DDR RAM seçenekleri bulunan açık kaynaklı bir tek kart bilgisayardır. 256MB RAM bulunduran versiyonu 7$ üzerinden satılmaktadır.

Bu bölümde kısaca cihazın teknik özelliklerini inceleyecek sonrasında cihazı açabilmek için gerekli araçları nasıl edinebileceğimize bakacağız. Cihazı açmak için Android, Ubuntu veya Raspbian gibi bir dağıtım kullanmak yerine önyükleyici, kernel ve dosya sistemi gibi gerekli araçları kendimiz oluşturacağız. Bu sayede ilk olarak asgari düzeyde işimizi görecek bir sistem ile cihaza login olmayı hedeflemekteyiz.

Orange Pi Zero Teknik Özellikleri

Cihazın temel teknik özelliklerini aşağıdaki gibi listeleyebiliriz.

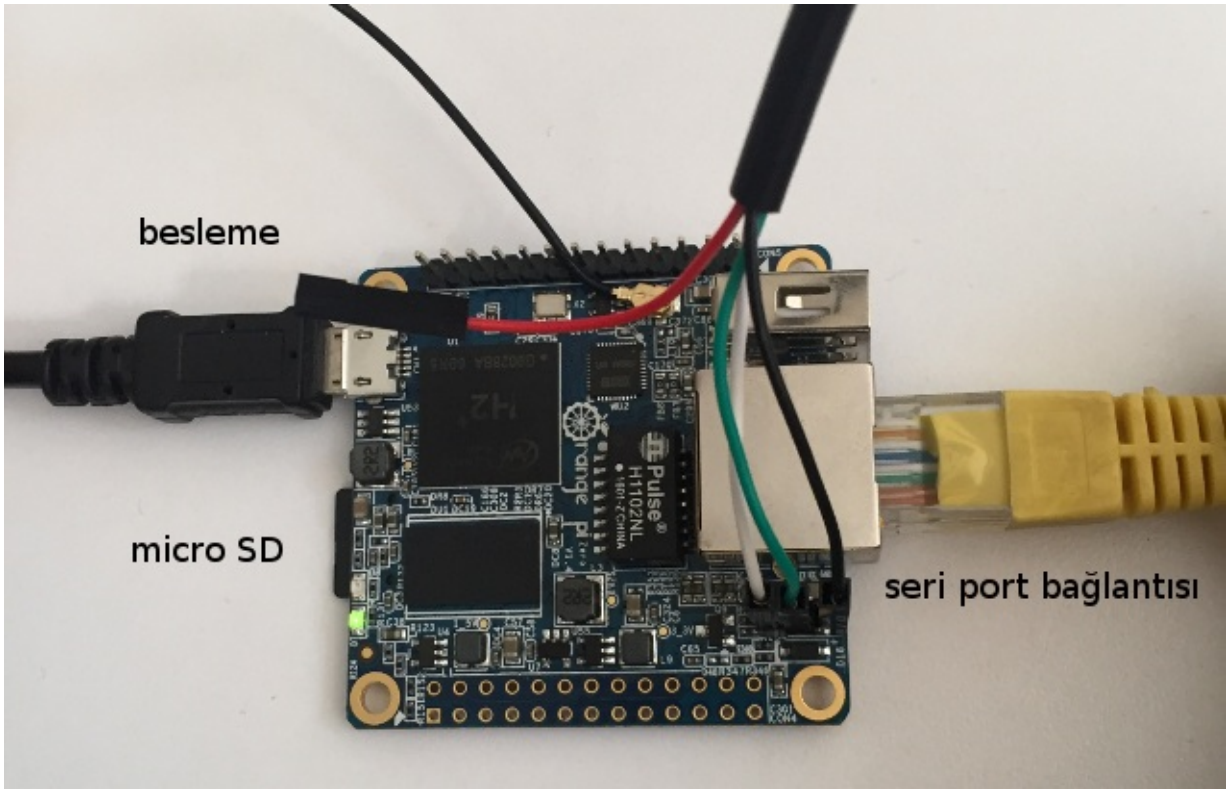
Özellik	Açıklama
CPU	H2 Quad-core Cortex-A7
GPU	Mali400MP2 GPU @600MHz
RAM	256MB/512MB DDR3 SDRAM (GPU paylaşımlı)
Depolama	micro SD (maksimum 64GB)
Network	10/100M Ethernet
WIFI	XR819, IEEE 802.11

Cihaz depolama ortamı olarak micro SD kullanmakta ve USB OTG üzerinden beslenmektedir. Cihaza enerji verildiğinde ilk olarak işlemci yongasındaki dahili ROM bootloader çalışmaktadır. Bu alt seviye önyükleyici sonrasında micro SD üzerindeki daha gelişkin bir önyükleyiciyi çalıştırmaktadır. Derleyeceğimiz önyükleyici, kernel ve dosya sistemini micro SD üzerine uygun şekilde yerleştireceğiz.

Seri Konsol Bağlantısı

Cihazla birlikte bir seri konsol kablosu gelmemektedir. Kabloyu nasıl edinebileceğinizle ilgili, bir sonraki Raspberry Pi bölümündeki seri konsol bağlantı başlığına bakabilirsiniz.

Cihaz ile seri konsol bağlantısı kurabilmek için kart üzerinde ethernet soketinin yanında 3 adet pin bulunmaktadır. Doğru bağlantı şekli aşağıda verilmiştir. Aşağıdaki şekil referans alınırsa cihaz üzerinde en soldaki pin Tx, ortadaki Rx ve en sağdaki uç ise toprak ucudur. Dönüştürücü kabloda sıralama içten dışa doğru beyaz, yeşil ve siyah şeklinde olmalıdır.


besleme
micro SD
seri port bağlantısı

Geliştirme Ortamının Hazırlanması

Bu başlık altında, cihazı açabilmek için gerekli olan araçları nasıl edinebileceğimize ve üretebileceğimize bakacağız. Derleme işlemini kendi bilgisayarımızda yapacağımız için ilk olarak bir çapraz derleyiciye ihtiyacımız olacak. Ayrıca, daha sonra oluşturacağımız önyükleyici, kernel gibi dosyaları kendi bilgisayarımızda uygun bir dizin hiyerarşisi altında toplamak ve sonrasında buradan SD karta yazmak faydalı olacaktır.

Biz kendi sistemimizde ilk olarak aşağıdaki gibi bir dizin yapısı oluşturduk. İhtiyaç duydukça yeni dizinler ekleyeceğiz.

```
/opt/orangepi
├── downloads
├── RootFS
└── toolchain
```

Geliştirme sürecimizde çapraz derleyici olarak `Linaro` firmasının derleyici setini kullanacağız. Siz de hard float desteği olan başka bir derleyici seti kullanabilirsiniz. Bu durumda vereceğimiz örneklerdeki derleyici öneklerini derleyicinizin öneki ile değiştirmelisiniz.

Derleyici setini aşağıdaki gibi indirip sisteminize kurabilirsiniz.

```
$ cd /opt/orangepi/downloads/
$ wget https://releases.linaro.org/components/toolchain/binaries/4.9-2016.02/\
       arm-linux-gnueabihf/gcc-linaro-4.9-2016.02-x86_64_arm-linux-gnueabihf.tar.xz
$ tar xf gcc-linaro-4.9-2016.02-x86_64_arm-linux-gnueabihf.tar.xz -C ../toolchain/
```

Bu işlem sonrasında `/opt/orangepi/toolchain` dizini altında geliştirme araçlarını içeren bir dizin oluşmuş olmalıdır. Oldukça uzun bir isme sahip bu dizine bir sembolik link oluşturmak daha sonra bu dizini `PATH` çevre değişkenine eklerken işimizi kolaylaştıracaktır. Bu işlemler aşağıdaki gibi yapılabilir.

```
$ cd ../toolchain/
$ ln -s gcc-linaro-4.9-2016.02-x86_64_arm-linux-gnueabihf/ linaro-arm-linux-gnueabihf
$ export PATH=$PWD/linaro-arm-linux-gnueabihf/bin:$PATH
```

Bu aşamadan sonra bulunduğumuz terminal üzerinde çapraz derleyiciyi kullanabiliriz, aşağıdaki gibi test edebilirsiniz.

```
$ arm-linux-gnueabihf-gcc -v
```

Her seferinde çapraz derleyici dizini yolunu `PATH` değişkenine eklemekle uğraşmamak için aşağıdaki satırı `ls ~/.bashrc` dosyanıza ekleyebilirsiniz.

```
export PATH=/opt/orangepi/toolchain/linaro-arm-linux-gnueabihf/bin:$PATH
```

Bu aşamadan sonra önyükleyici ve kernel derleme aşamalarına geçebiliriz.

U-boot Derleme Süreci

Derleme işleminden önce sisteminizde aşağıdaki paketlerin kurulu olduğundan emin olunuz. Gerekli paketleri Debian tabanlı bir sistemde aşağıdaki gibi kurabilirsiniz.

```
$ sudo apt install device-tree-compiler
$ sudo apt-get install libfdt-dev
$ sudo apt-get install swig libpython-dev
```

U-boot kodunu aşağıdaki gibi indirip derleyebilirsiniz.

```
cd /opt/orangepi
$ git clone git://git.denx.de/u-boot.git
```

U-boot kodu artık ana geliştirme dizinimiz altında `u-boot` isimli bir dizinde bulunmaktadır. `u-boot/configs` dizini altına bakılacak olursa çok sayıda ayar dosyası görülecektir. Biz cihazımız için uygun olan `orangepi_zero_defconfig` dosyasını kullanacağız. Bu dosya üzerinde yalnız u-boot delay süresini güncelleyeceğiz. U-boot, işletim sistemi çekirdeğini yüklemeden önce belli süre beklemektedir, bu aşamada seri konsol üzerinden bir tuşa basılırsa u-boot komut satırına düşülmektedir. Öngörülen olarak 2 sn olan bu süreyi 5 sn olarak değiştireceğiz. Bu işlem için make içindeki `menuconfig` kuralı kullanılabilir.

```
$ make orangepi_zero_defconfig
$ make menuconfig
```

`make menuconfig` sonrasında aşağıdaki gibi bir değişiklik yapılıp değişiklik saklanabilir.

```
(5) delay in seconds before automatically booting
```

Bu aşamadan sonra U-boot kodunu derleyebiliriz.

```
$ make CROSS_COMPILE=arm-linux-gnueabihf-
```

Bu işlem sonunda aşağıdaki dosyalar oluşacaktır.

Dosya	Açıklama
u-boot.bin	U-boot önyükleyicisi
spl/sunxi-spl.bin	U-boot'dan önce çalışan ikincil önyükleyici (secondary program loader)
u-boot-sunxi-with-spl.bin	spl ve U-boot önyükleyicilerinin ard arda eklendiği dosya

`u-boot.bin` ve `sunxi-spl.bin` dosyalarını ayrı ayrı SD kart üzerine atmak yerine `u-boot-sunxi-with-spl.bin` dosyasını kullanacağız.

Artık kernel derleme aşamasına geçebiliriz.

Kernel Derleme Süreci

Orange Pi Zero için kernel kodunun kararlı son hali aşağıdaki gibi klonlanabilir.

```
$ cd /opt/orangepi
$ git clone --depth 1 git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stabl
e.git
```

Kernel kodunu aşağıdaki gibi derleyebiliriz, öngörülen ayar dosyası üzerinde bu aşamada bir değişiklik yapmayacağız.

```
$ cd linux-stable
$ make ARCH=arm clean
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- sunxi_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage -j4
```

Bu işlem sonunda `arch/arm/boot/` altında kernel imaj dosyası `zImage` oluşmuş olmalıdır.

Kernel modülleri ise ayrı derlenmelidir. Daha sonra ayar dosyası üzerinde değişiklik yapılarak istenilen özellikler modül olarak derlenebilir veya tüm özellikler kernel koduna dahil edilebilir. Kernel modülleri aşağıdaki gibi derlenebilir.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules -j4
```

Bu işlem sonunda `.ko` uzantılı kernel modülleri oluşacaktır. Bu modüller hedef dosya sisteminde uygun bir dizin altında bulunmalıdır. Cihaz için dosya sistemini hazırladığımız bölümde kernel modüllerinin nasıl kopyalanması gerektiğine bakacağız.

Son olarak kartın donanımına ait parametreleri barındıran `.dtb` dosyasını üretmeliyiz. Bu işlem aşağıdaki gibi yapılabilir.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- dtbs -j4
```

Bu işlem sonunda `arch/arm/boot/dts/sun8i-h2-plus-orangepi-zero.dtb` dosyası oluşacaktır. Bu dosyayı da kernel imajı ve modüllerle birlikte SD kart üzerinde `/boot` dizini altına taşıyacağız.

Bu aşamadan sonra dosya sistemini oluşturmaya geçebiliriz.

> NOT: Dokümanı hazırladığımız sırada OrangePi Zero ethernet ve wifi modülleri mainline kernel (4.13) içerisinde desteklenmiyordu. Konu ile ilgili ayrıntı için sonraki bölümü inceleyiniz.

Wifi Desteği - Problemli Senaryo Örneği

Orange Pi Zero 7$'lık fiyatı ile her ne kadar çok cazip görünse de, elbette ucuz etin yahnisi durumu burada da söz konusu oluyor.

Linux mainline kernel 4.13'te ne ethernet nede wifi desteklenmiyor. Daha önceki sürümlerle çalışan, çok da iyi olmayan bir ethernet ve wifi desteği mevcut olmakla birlikte, yeni kernel'lar ile birlikte kullanılan subsystem'lerdeki değişikliklere göre gereken güncellemeleri yapmadıkları için, mainline kernel'dan çıkartılmış. Problemin yakın zamanda çözüleceği belirtilmiş ancak an itibariyle sorunun devam ettiğini görüyoruz.

Bu yazıyı okuduğunuz sırada kernel versiyonu muhtemelen 4.13'ten yüksek ve burada bahsettiğimiz problem tamamen çözülmüş olacak. Bununla birlikte özellikle gömülü sistemler ile çalışırken zaman zaman karşılaşabileceğiniz bu tarz durumlara örnek teşkil etmesi amacıyla, wifi modülünü çalıştırmak için neler yaptığımız aktarmaya çalışacağız.

```
/home/demirten/embedded/orange/xradio/rx.c: In function ‘xradio_rx_cb’:
/home/demirten/embedded/orange/xradio/rx.c:205:16: error: ‘RX_FLAG_HT’ undeclared (fir
st use in this function)
   hdr->flag |= RX_FLAG_HT;
                ^~~~~~~~~~
/home/demirten/embedded/orange/xradio/rx.c:205:16: note: each undeclared identifier is
 reported only once
for each function it appears in
```

...

...

```
--- ./arch/arm/boot/dts/sun8i-h2-plus-orangepi-zero.dts.orig    2017-09-08 20:15:25.33
8333936 +0300
+++ ./arch/arm/boot/dts/sun8i-h2-plus-orangepi-zero.dts    2017-09-08 20:25:25.2678114
62 +0300
@@ -56,7 +56,7 @@

     aliases {
         serial0 = &uart0;
-        ethernet1 = &xr819;
+        ethernet1 = &xr819wifi;
     };

     chosen {
@@ -78,6 +78,16 @@
         };
     };
```

```
+    vdd_wifi: vdd_wifi {
+        compatible = "regulator-fixed";
+        regulator-name = "wifi";
+        regulator-min-microvolt = <1800000>;
+        regulator-max-microvolt = <1800000>;
+        gpio = <&pio 0 20 GPIO_ACTIVE_HIGH>;
+        startup-delay-us = <70000>;
+        enable-active-high;
+    };
+
     reg_vcc_wifi: reg_vcc_wifi {
         compatible = "regulator-fixed";
         regulator-min-microvolt = <3300000>;
@@ -115,19 +125,22 @@
 &mmc1 {
     pinctrl-names = "default";
     pinctrl-0 = <&mmc1_pins_a>;
+    vqmmc-supply = <&vdd_wifi>;
     vmmc-supply = <&reg_vcc_wifi>;
     mmc-pwrseq = <&wifi_pwrseq>;
     bus-width = <4>;
     non-removable;
     status = "okay";

-    /*
-     * Explicitly define the sdio device, so that we can add an ethernet
-     * alias for it (which e.g. makes u-boot set a mac-address).
-     */
-    xr819: sdio_wifi@1 {
-        reg = <1>;
-    };
+        xr819wifi: xr819wifi@1 {
+                reg = <1>;
+                compatible = "xradio";
+                pinctrl-names = "default";
+                interrupt-parent = <&pio>;
+                interrupts = <6 10 IRQ_TYPE_EDGE_RISING>;
+                interrupt-names = "host-wake";
+                local-mac-address = [dc 44 6d c0 ff ee];
+        };
 };
```

...

```
# modprobe xradio_wlan

xradio_wlan: loading out-of-tree module taints kernel.
  xradio_wlan mmc1:0001:1:    Input buffers: 30 x 1632 bytes
    Hardware: 7.9
    WSM firmware ver: 8, build: 43, api: 1060, cap: 0x0003
  xradio_wlan mmc1:0001:1: Firmware Label:XR_C01.08.0043 Jun  6 2016 20:41:04
```

Dosya Sisteminin Hazırlanması

Bu bölümde busybox ile temel bir dosya sisteminin nasıl hazırlandığına bakacağız, benzer anlatım bir sonraki konu olan Raspberry Pi bölümünde de bulunmaktadır. Dosya sistemi hazırlanmasıyla ilgili daha detaylı bilgiler için ise *Kök Dosya Sistemi Oluşturma* başlıklı bölümü inceleyebilirsiniz.

Biz burada konunun bütünlüğünü korumak için kullandığımız derleme seti ile temel bir dosya sistemini `/opt/orangepi/RootFS/` altında oluşturacağız.

Busybox

`Busybox` kodlarını aşağıdaki gibi indirip derleyebiliriz. İstenirse öngörülen ayar dosyası `make defconfig` işlemi sonrasında `make menuconfig` ile özelleştirilebilir.

```
$ cd /opt/orangepi
$ git clone git://busybox.net/busybox.git
$ cd busybox/
$ make defconfig
$ make CROSS_COMPILE=arm-linux-gnueabihf-
```

Bu işlem sonucunda `busybox` dosyası oluşacaktır. Bu aşamadan sonra gerekli sembolik linkler oluşturulmadır. Bu işlem için Makefile dosyası `install` isimli bir hedef barındırmaktadır. Bu işlemden önce dosya sistemini kendi bilgisayarımızda nerede oluşturacağımıza karar vermeliyiz. Biz bu amaçla `/opt/orangepi/RootFS` dizinini kullanacağız. İlk olarak bu dizin altındaki dosyaların sahibi olarak kendi kullanıcımızı tanımlayalım.

```
$ sudo chown -R $USER /opt/orangepi/RootFS/
```

Derleme işleminden sonra `make install` dediğimizde `_install` adlı bir dizin altında, ihtiyaç duyduğumuz sembolik linkleri barındıran, `/bin` , `/sbin` ve `/usr` dizinleri oluşacaktır. `CONFIG_PREFIX` değişkenini kullanarak bu dizinleri, `_install` dizini yerine, RootFS altında aşağıdaki gibi oluşturabiliriz.

```
$ make CROSS_COMPILE=arm-linux-gnueabihf- CONFIG_PREFIX=/opt/orangepi/RootFS install
```

Bu işlem sonucunda RootFS dizininde aşağıdaki dosyalar oluşmuş olmalıdır.

```
drwxrwxr-x 2 serkan serkan 4096 Ağu 17 18:24 bin
lrwxrwxrwx 1 serkan serkan   11 Ağu 17 18:24 linuxrc -> bin/busybox
drwxrwxr-x 2 serkan serkan 4096 Ağu 17 18:24 sbin
drwxrwxr-x 4 serkan serkan 4096 Ağu 17 18:24 usr
```

Temel Kütüphanelerin Taşınması

Cihaz üzerinde çalışacak busybox ve diğer uygulamalar `libc` ve `libm` gibi birtakım temel kütüphanelere ihtiyaç duymaktadır. Bu kütüphaneleri kullandığımız derleyici setinden temin edeceğiz.

Derleyici, derleme esanasında kullandığı hedef sisteme ait kütüphane ve başlık dosyalarını sysroot denilen bir dizin yapısında saklamaktadır. Bu dizin yapısını aşağıdaki gibi öğrenebiliriz.

```
$ arm-linux-gnueabihf-gcc --print-sysroot
```

Bu dizinden yapacağımız kopyalama işlemlerini basitleştirmek için bu dizini gösteren bir çevre değişkenini aşağıdaki gibi tanımlayabiliriz.

```
SYSROOT=$(arm-linux-gnueabihf-gcc --print-sysroot)
```

`RootFS` altında `/lib` dizinini oluşturduktan sonra ilk planda ihtiyaç duyduğumuz kütüphaneleri aşağıdaki gibi kopyalayabiliriz.

```
$ mkdir -p /opt/orangepi/RootFS/lib
$ cp $SYSROOT/lib/libc.so.6 /opt/orangepi/RootFS/lib/
$ cp $SYSROOT/lib/libm.so.6 /opt/orangepi/RootFS/lib/
$ cp cp $SYSROOT/lib/ld-linux-armhf.so.3 /opt/orangepi/RootFS/lib/
```

Kernel ve DTB Dosyasının Taşınması

Daha önce derlediğimiz kernel imajını ve dtb dosyasını `RootFS` altında `/boot` dizinine aşağıdaki gibi taşıyabiliriz.

```
$ mkdir /opt/orangepi/RootFS/boot
$ cd /opt/orangepi/RootFS
$ cp ../linux-stable/arch/arm/boot/zImage boot/
$ cp ../linux-stable/arch/arm/boot/dts/sun8i-h2-plus-orangepi-zero.dtb boot/
```

Kernel Modüllerinin Taşınması

Kernel modülleri dosya sistemi üzerinde `/lib/modules/kernel-version` şeklinde uygun dizin yapısında bulunmalıdır. Kopyalama işlemini aşağıdaki gibi yapabiliriz.

```
$ cd /opt/orangepi/linux-stable/
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- INSTALL_MOD_PATH=../RootFS modules_
install
```

Bu işlem sonrasında `RootFS` altında `lib/modules/4.13.0` gibi bir dizin oluşmalıdır.

Açılış Betiğinin Hazırlanması

İşletim sistemi çekirdeği açılış sürecini tamamladığında, dosya sistemi üzerinde bir takım öntanımlı dizinlere bakarak, `init` isimli bir uygulamayı aramaktadır. İlk çalışan uygulama olan `init` `/etc/inittab` isimli bir dosyayı okuyarak gerekli işlemleri yapmaktadır.

İlk olarak `RootFS` altında dosya sistemi için temel dizinleri aşağıdaki gibi oluşturalım.

```
$ cd /opt/orangepi/RootFS
$ mkdir etc dev proc sys tmp var
```

Basit bir `inittab` dosyasının içeriği aşağıdaki gibi olabilir.

```
# /etc/inittab

::sysinit:/etc/rcS

::askfirst:-/bin/sh

::shutdown:/bin/sync
::shutdown:/bin/umount -a -r
```

`inittab` içerisinde `sysinit` ile başlayan satırda `rcS` açılış betiğinin çalıştırıldığını görüyoruz. Açılışa ilişkin temel işlemler bu betik içinde yapılmaktadır. Temel bir açılış betiği aşağıdaki gibi olabilir.

```
#!/bin/sh

export PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/bin

mount -t proc   proc   /proc
mount -t sysfs  sysfs  /sys
mount -t tmpfs -o mode=1777 tmpfs  /tmp

mkdir /dev/pts
mount -t devpts -o gid=5,mode=620 devpts /dev/pts

mkdir /dev/shm
mount -t tmpfs -o mode=0777 tmpfs /dev/shm

mount -t tmpfs -o mode=0755,nosuid,nodev tmpfs /var

ifconfig lo 127.0.0.1 up
route add -net 127.0.0.0 netmask 255.0.0.0 gw 127.0.0.1 lo

mount -o remount -o rw /
```

Son durumda RootFS dizinimizin içeriği aşağıdaki gibi olmalıdır.

```
$ tree  -L 1 RootFS/
RootFS/
├── bin
├── boot
├── dev
├── etc
├── lib
├── linuxrc -> bin/busybox
├── proc
├── sbin
├── sys
├── tmp
├── usr
└── var
```

Bu aşamadan sonra SD kartın hazırlanması kısmına geçebiliriz.

SD Kartın Hazırlanması

Bu başlık altında açılabilir bir SD kartı nasıl hazırlayabileceğimize bakacağız. SD kartın organizasyonu aşağıdaki gibi olmalıdır.

Başlangıç	Boyut	Kullanım
0	8KB	Bölümlendirme tablosunun da bulunduğu kullanılmayan alan
8	24KB	Başlangıç SPL yükleyicisi
32	512KB	U-boot
544	128KB	U-boot çevre değişkenleri
672	352KB	Saklı tutulmuş alan (reserved)
1024	-	Bölümlendirme ve dosya sistemi için ayrılan alan

Yukarıdaki tabloya dikkat edilecek olursa bölümlendirmenin ilk 1MB'lık alandan sonra başladığı görülmektedir. Cihaza enerji verildiğinde ilk çalışan kod olan ROM boot kodu SD kartın 8KB adresinden başlayan alanda başlangıç yükleyicisi SPL'i aramaktadır. SPL daha sonra asıl önyükleyici olan U-boot'u yüklemektedir.

SD kart üzerinde yukarıdaki organizasyonu sağlamak için ilk olarak SD kartı geliştirme bilgisayarımıza takalım. SD kartı taktığımızda `/dev` altında karta ilişkin bir dosya düğümü oluşmalıdır. `dmesg` çıktısından bu isme ulaşabiliriz. Bizim sistemimizde aldığımız sonuç aşağıdaki gibidir.

```
$ dmesg | tail
...
[12513.272757] mmcblk0: mmc0:1234 SA16G 14.4 GiB
[12513.275031]  mmcblk0: p1 p2
```

Bu durumda `/dev` altında `mmcblk0` adında bir dosya düğümü oluşmuş olmalı.

```
$ ls /dev/mmcblk0
/dev/mmcblk0
```

SD kartı harici bir SD kart okuyucu ile sisteminize bağladı iseniz sisteminizde bu dosya düğümü `/dev/sdb` veya `/dev/sdc` şeklinde oluşabilir. Dosya düğümü doğru bir şekilde belirlemek oldukça önemlidir. Yanlış bir dosya düğümü ile çalışmak sisteminize zarar verebilir. Dosya düğümünü belirledikten sonra bu düğüme ilişkin bir çevre değişkeni aşağıdaki gibi oluşturulabilir.

```
$ export CARD=/dev/mmcblk0
```

Daha sonra SD kart üzerinde bir bölüm oluşturduğumuzda bu bölüme ilişkin dosya düğümünün adında partition'ı ifade eden `p` harfi geçecektir. Bu duruma ilişkin aşağıdaki gibi bir çevre değişkeni daha tanımlanabilir.

```
$ export p=p
```

SD kart üzerinde ilk olarak üzerinde dosya sistemini oluşturacağımız bir bölüm (partition) oluşturalım. Bu bölüm SD kartın ilk 1MB'lık alanından sonra başlamalıdır. Bölümlendirme için Linux altında `fdisk` veya `gparted` benzeri araçlar kullanılabilir. Bu işlem öncesinde SD kart mount edilmiş ise ilk olarak dosya sisteminden bağı koparılmalıdır. Bu amaçla `umount` komutu kullanılabilir. Örneğin:

```
$ sudo umount /dev/mmcblk0p1
```

fdisk ile SD kart üzerinde bir bölüm aşağıdaki gibi oluşturulabilir.

```
$ sudo fdisk ${CARD}
```

Sonrasında aşağıdaki komutlar girilmelidir. Sonuçta 32 MB'lık bir bölüm oluşturulacaktır. Farklı boyutta bir bölüm oluşturmak istiyorsanız aşağıdaki örnekte `+32M` yazan kısmı ihtiyacınıza göre değiştirebilirsiniz.

```
Command (m for help): o                                 # Yeni bir bölümlendirme tablos
u oluşturuyoruz
Command (m for help): n                                 # Yeni bir bölüm oluşturuyoruz
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p):                                     # Enter ile geç (Birincil bölüm
 oluşturuluyor)
Using default response p
Partition number (1-4, default 1):                      # Enter ile geç (Bölüm numarası
 1 olarak veriliyor)
Using default value 1
First sector (2048-15523839, default 2048):             # Enter ile geç (2MB sonrasında
 ilk bölüm başlıyor)
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-15523839, default 15523839): +32M       #
+32M ile 32MB'lık
                                                                                    #
bir bölüm oluşturuluyor

Command (m for help): w                                 # w ile değişiklikler kaydedili
yor
The partition table has been altered!
```

Artık üzerinde dosya sistemi oluşturabileceğimiz bir bölümümüz bulunmakta. Bu bölüme aşağıdaki gibi bir `ext4` dosya sistemi oluşturabiliriz. Bu bölümlendirmemize ilişkin aşağıdaki gibi bir dosya düğümü oluşmuş olmalıdır.

```
$ ls ${CARD}${p}1
/dev/mmcblk0p1
```

Bu bölüm üzerinde `ext4` dosya sistemini aşağıdaki gibi oluşturabiliriz.

```
$ sudo mkfs.ext4 ${CARD}${p}1
```

Bu aşamadan sonra SD kartı dosya sistemimize mount ederek üzerine yazabiliriz. Geliştirme dizinimiz altında disk isimli bir dizin oluşturup SD kartı buraya mount edebiliriz.

```
$ cd /opt/orangepi
$ mkdir disk
$ sudo mount  ${CARD}${p}1 disk
```

Artık SD kartı üzerindeki dosya sistemine yazabilir durumdayız. Kendi sistemimizde oluşturduğumuz dosya sistemini SD kart üzerine aşağıdaki gibi atabiliriz.

```
$ sudo cp -a RootFS/* disk/
```

Bu işlem sonrasında SD kart üzerine yazma yapıldığından emin olalım ve SD kartı umount edelim.

```
$ sync
$ umount disk
```

Geriye yalnız önyükleyicileri SD kart üzerine yazma kısmı kaldı. U-boot kodunu derlediğimiz kısımda da belirttiğimiz gibi SD kart üzerine SPL ve U-boot önyükleyicilerini içeren `u-boot-sunxi-with-spl.bin` dosyasını atacağız. Öncesinde bölümlendirme tablosununun bulunduğu alan dokunmadan, 1KB ileriden başlayarak ilk 1MB'lık alanı aşağıdaki gibi temizleyelim.

```
$ sudo dd if=/dev/zero of=${CARD} bs=1k count=1023 seek=1
```

Bu aşamadan sonra önyükleyici çiftini SD kart üzerine, 8KB ileriden başlayacak şekilde, yazabiliriz.

```
$ sudo dd if=u-boot/u-boot-sunxi-with-spl.bin of=${CARD} bs=1024 seek=8
```

Artık SD kartı cihaza takıp cihaza enerji verebiliriz

Cihazın Açılması

Daha önce seri konsol bağlantısının anlatıldığı kısımda gösterildiği gibi cihaz ile seri port bağlantısı oluşturulduktan ve SD kartı cihaza taktıktan sonra cihaza OTG üzerinden enerji verebiliriz. Bu aşamadan sonra `gtkterm` veya `minicom` gibi bir seri konsol emülatörü üzerinden cihazın konsol çıktısı izlenebilir.

U-boot kodunu derlerken otomatik boot öncesindeki bekleme süresini 5 sn olarak belirlemiştik. Bu süre içinde bir tuşa basarak U-boot komut satırına düşmeliyiz. U-boot komut satırında `printenv` dediğimizde çevre değişkenleri ve değerlerini görebilmeliyiz. İlk olarak kernel komut satır argümanlarını tutan `bootargs` değişkenine uygun değeri verelim.

```
=> setenv bootargs console=ttyS0,115200 earlyprintk root=/dev/mmcblk0p1 ro rootwait
```

Bu sayede kernel'a kök dosya sisteminin yerini ve konsolu belirtiyoruz. Sonrasında SD kart üzerindeki dosya sisteminde bulunan kernel ve dtb dosyalarını bellekte uygun bir alana çekmeliyiz. `bdinfo` ile karta ilişkin bazı bilgilere erişilebilir.

```
=> bdinfo
arch_number = 0x00000000
boot_params = 0x40000100
DRAM bank   = 0x00000000
-> start    = 0x40000000
-> size     = 0x10000000
...
```

Kernel ve dtb dosyasını start ile belirtilen adresin sonrasına güvenli bir alana sırasıyla aşağıdaki gibi çekebiliriz.

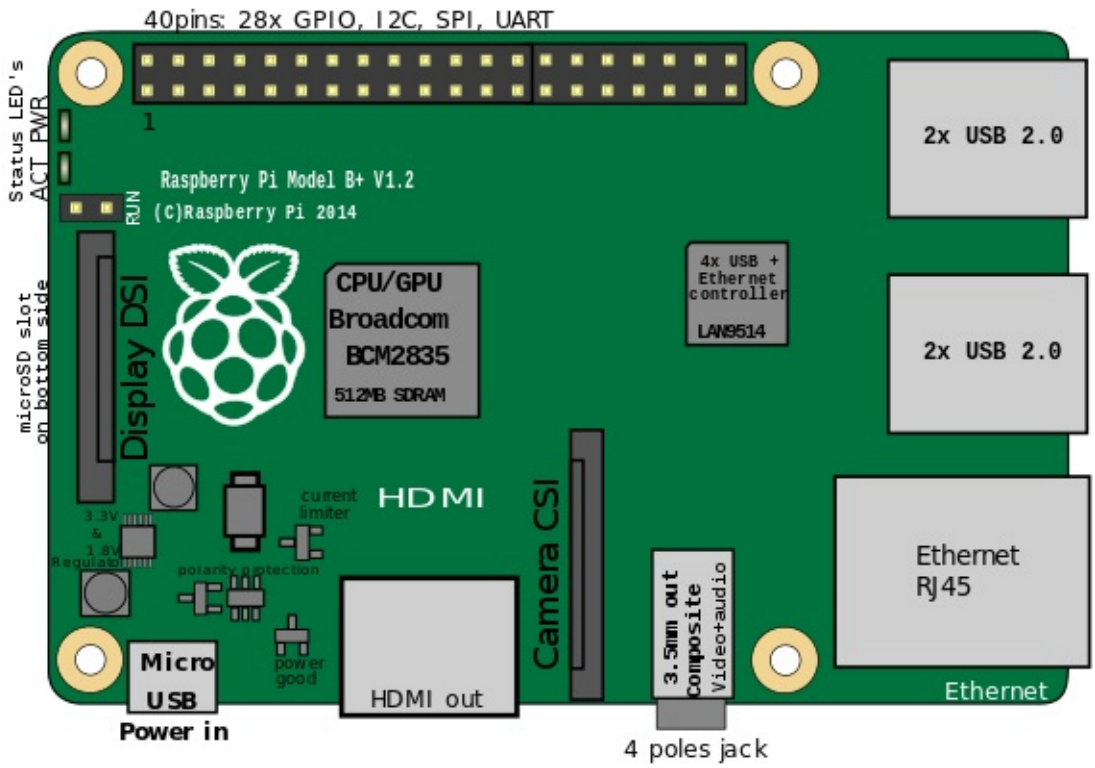
```
=> ext2load mmc 0 0x46000000 boot/zImage
=> ext2load mmc 0 0x49000000 boot/sun8i-h2-plus-orangepi-zero.dtb
```

Kernel'ı boot etmek için `bootz` komutunu kullanacağız. bootz komutuna sırasıyla kernel, initrd ve dtb adresleri geçirilmektedir. Biz initrd kullanmadığımız için bu seçeneği `-` ile geçeceğiz.

```
=> bootz 0x46000000 - 0x49000000
```

Bu işlem sonunda konsolda `Please press Enter to activate this console.` yazısı görülmelidir. Bir tuşa basıldığında kabuk komut satırına düşülecektir. Bu aşamadan sonra cihaz üzerinde çalışılmaya hazır durumdadır.

Raspberry Pi



Raspberry Pi, *Raspberry Pi Foundation* tarafından eğitim amacıyla geliştirilmiş, tek kart, mini bilgisayar serisidir. *Model A*, *Model B* ve eMMC flash'a sahip *Compute Module* olmak üzere türleri bulunmaktadır. Model A ve Model B serileri (Model B, Model B+, Model Pi 2) sırasıyla $25-$35 aralığında fiyatlardan satılmaktadır.

Bu bölümdeki incelemelerimizi ilk olarak 32 bitlik işlemciye sahip Raspberry Pi 2 - Model B üzerinde yapacak sonrasında ayrı bir başlık altında 64 bitlik işlemciye sahip daha üst bir model olan Raspberry Pi 3 Model B'ye bakacağız. Bölüm içerisinde temel olarak cihazların teknik özelliklerini, açılış sürecini, gerekli araçları nasıl edinebileceğimizi ve cihaz üzerinde işletim sistemini nasıl çalıştırabileceğimizi inceleyeceğiz.

Raspberry Pi 2 Teknik Özellikleri

Cihazın temel teknik özelliklerini aşağıdaki gibi listeleyebiliriz.

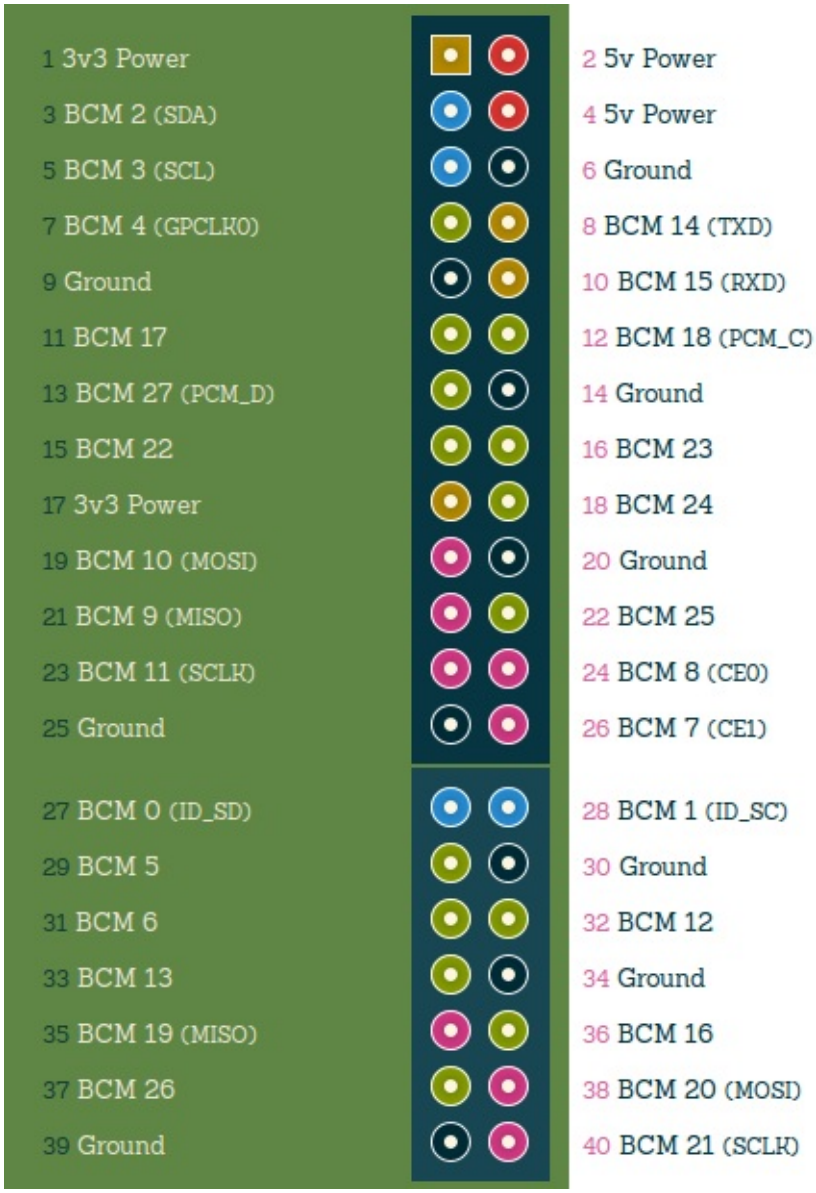
Özellik	Açıklama
SoC	Broadcom BCM2836 (CPU, GPU, DSP, SDRAM, and single USB port)
CPU	900 MHz quad-core ARM Cortex A7 (ARMv7 instruction set)
GPU	Broadcom VideoCore IV @ 250 MHz
RAM	1GB (GPU ve CPU paylaşımlı)
Video çıkışı	HDMI, composite video
Hafıza	MicroSD
Giriş/Çıkış	17 GPIO plus specific functions
Diğer	OpenGL ES 2.0, 1080p30 h.264/MPEG-4 AVC high-profile decoder ve encoder

Broadcom tarafından üretilen **BCM2836** SoC (System-on-Chip), bünyesinde temel olarak grafik işlemci (GPU), genel amaçlı işlemci (ARM CPU), ROM (Read-Only-Memory) ve SDRAM barındırmaktadır. Grafik işlemci olarak yine Broadcom tarafından üretilen **VideoCore IV** kullanılırken, genel amaçlı işlemci olarak 900 MHz quad-core ARM Cortex A7 kullanılmıştır.

> **Not:** İşlemci için **BCM2836** yerine **BCM2709** kodu da kullanılabilmektedir. BCM2836 işlemci ailesini gösterirken BCM2709 işlemcinin gerçek kodunu göstermektedir. Linux tarafından */proc/cpuinfo* çıktısında ikinci kodun listelendiğini görmekteyiz.

Cihazın dış dünya ile iletişimini sağlayan **GPIO** (General Purpose Input/Output) pin dizilimi aşağıdaki gibidir. Bazı pinlerin hem GPIO hem de başka bir amaçla kullanılabildiğine dikkat ediniz.

> **Not:** Resimde yer alan boşluğun altındaki bölümde, **Model B+** ile gelen **14** yeni GPIO bilgisi verilmiştir. Daha eski modeller için ise üst kısmı aynen geçerlidir.



Seri Konsol Bağlantısı

Raspberry cihazınız ile birlikte seri konsol kablosu gelmemektedir. Yurt içinde bu kabloyu temin etmekte güçlük çekebilirsiniz veya pahalı olabilir. AliExpress üzerinden tanesini 1-2$ aralığında getirtebilirsiniz. Arama yaparken *PL2303HX USB to UART TTL* anahtar sözcüklerini kullanabilir veya şu link üzerinden gidebilirsiniz.

Konsol bağlantısı için cihaz üzerindeki 6 (Ground - Siyah), 8 (Transmit - Beyaz) ve 10 (Receive - Yeşil) nolu pinleri kullanmalısınız. Aşağıda doğru bir bağlantı örneği gösterilmiştir:


HDMI

Açılış Süreci

Raspberry Pi için ARM işlemci ana işlemci olarak değil daha çok bir yardımcı işlemci (coprocessor) olarak tasarlanmıştır. Ana işlemci ise VideoCore grafik işlemcisidir. Bu sebeple cihazın açılış süreci gömülü cihazların genel olarak izlediği yoldan bir miktar ayrışmaktadır.

Tipik bir ARM işlemcili cihaz (kitabımızda incelediğimiz diğer örnek cihazlar dahil olmak üzere) açılış sürecinde genel olarak aşağıdaki adımları izlemektedir:

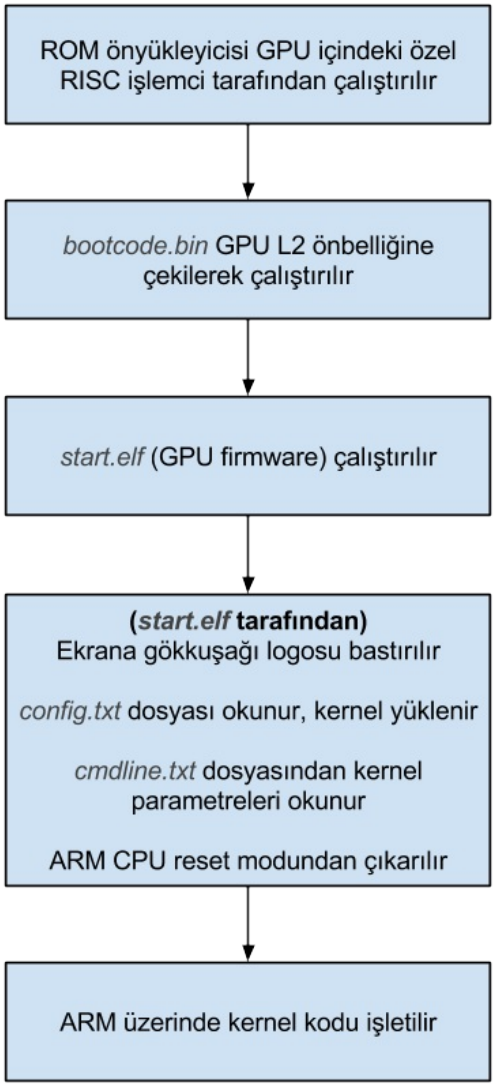
- Cihaza enerji verildikten sonra ARM işlemci çalışır.
- ARM işlemci, ilk olarak, bünyesindeki ROM bellekteki ilk önyükleyici (ROM bootloader) çalıştırır.
- ROM bootloader kendinden sonraki işletim sistemini yüklemekten sorumlu önyükleyiciyi yükler.
- İşletim sistemi belleğe yüklenir ve ARM işlemci tarafından çalıştırır.

Cihaza enerji verilmesinden işletim sisteminin yüklenmesine kadar geçen tüm aşamalarda ARM işlemcinin rol aldığını görmekteyiz. Raspberry Pi için ise açılış süreci temel olarak aşağıdaki gibidir.

- Cihaza enerji verildiğinde ARM işlemci reset modundadır, yani aktif olarak çalışmamaktadır.
- ROM bellekteki ilk önyükleyici (first-stage bootloader, ROM bootloader) VideoCore GPU içindeki küçük bir RISC (Reduced Instruction Set Computer) işlemci tarafından çalıştırılır.
- İlk önyükleyici tarafından SD karttaki FAT32 boot bölümü mount edilir ve ikinci önyükleyici (second-stage bootloader) **bootcode.bin**, GPU içindeki L2 cache'e çekilerek, çalıştırılır. Bu aşamada ARM işlemci hala reset durumundadır.
- İkinci önyükleyici tarafından aslında GPU firmware'i olan son önyükleyici (third-stage bootloader) **start.elf** yüklenir ve çalıştırılır. Bu aşamada ekrana gökkuşağı logosu bastırılmaktadır.
- *start.elf* tarafından SD kart boot bölümündeki **config.txt** dosyası okunur. Bu ayar dosyası ile bazı sistem ayarları değiştirilebilmekte (ARM ve GPU frekansları, RAM paylaşımları) ayrıca önyükleyiciye müdahale edilebilmektedir.
- SD kart boot bölümünde dtb (bcm2709-rpi-2-b.dtb) ve kernel (kernel.img) dosyalarının bulunması durumunda *start.elf* tarafından sırasıyla 0×100 ve 0×8000 adreslerine yüklenmektedir. Ayrıca **cmdline.txt** dosyasının bulunması halinde dosya içeriğindeki komut satırı argümanları çekirdeğe geçirilmektedir.
- Son olarak ARM işlemci reset durumundan çıkarılır ve işletim sistemini çalıştırması

sağlanır.

Cihaza enerji verilmesinden, işletim sisteminin çalıştırılmasına kadar geçen süreç aşağıdaki diagramda özetlenmiştir.



Bu sürece diğer cihazların açılış sürecinden aşina olduğumuz **u-boot** önyükleyicisinin karışmadığını görmekteyiz, *kernel* GPU firmware'i *start.elf* tarafından yüklenmekte ve ardında ARM işlemci reset durumundan çıkarılmaktadır. *config.txt* dosyasına müdahale ederek *start.elf* tarafından işletim sistemi çekirdeğinin değil *u-boot* önyükleyicisinin yüklenmesini sağlayabilir, bu sayede sistemimizi u-boot üzerinden açabiliriz. Bu işlemin nasıl yapıldığına daha sonra değineceğiz.

Gerekli Araçların Elde Edilmesi

Bu başlık altında, cihazı açabilmek için gerekli olan araçları nasıl edinebileceğimize ve üretebileceğimize bakacağız.

Geliştirme Araçları

Geliştirme araçları için birden fazla seçenek bulunmasına karşın, doğrudan donanım sağlayıcının önerdiği BCM2708 hedefli çapraz derleyicileri aşağıdaki gibi edinebilirsiniz:

```
$ git clone https://github.com/raspberrypi/tools
```

tools/arm-bcm2708/arm-bcm2708hardfp-linux-gnueabi/bin dizin yolunu PATH çevre değişkenine ekleyerek, **arm-bcm2708hardfp-linux-gnueabi** önekine sahip hard float desteği olan 32 bitlik derleyiciyi kullanabilirsiniz.

```
$ cd tools
$ export PATH=$PWD/arm-bcm2708/arm-bcm2708hardfp-linux-gnueabi/bin/:$PATH
```

GPU Önyükleyicileri

Cihazın açılış sürecini incelerken *bootcode.bin* ve *start.elf* olmak üzere iki adet önyükleyicinin gerekli olduğunu görmüştük. Raspberry Pi için önyükleyici ve GPU firmware dosyalarının derlenmiş hallerini aşağıdaki gibi edinebilirsiniz, bu dosyalara ilişkin kaynak kod erişimi verilmemektedir.

```
$ git clone https://github.com/raspberrypi/firmware
```

boot dizininde önyükleyicilerle beraber örnek çekirdek imajı da bulunmaktadır.

> **Not:** firmware git arşivi 4 GB'ın üzerinde olup çoğunlukla `boot` dizini altındaki bir kaç küçük dosyaya ihtiyaç duyacağımızdan, web üzerinden https://github.com/raspberrypi/firmware/tree/master/boot adresinden de bu dosyaları hızlıca indirebilirsiniz.

Kernel Derleme Süreci

Kaynak Kodun Edinilmesi

Raspberry Pi için kernel patch'leri https://github.com/raspberrypi/linux adresinde yayınlanmaktadır.

Kernel kodunu aşağıdaki gibi `git` ile klonlayarak indirebilirsiniz:

```
$ git clone https://github.com/raspberrypi/linux
```

Yukarıdaki *clone* işlemi tüm kernel kaynak kodunu geçmiş commit tarihçesi ile birlikte indireceğinden 1 GB'ın üzerinde download gerektirecektir. Alternatif olarak commit tarihçesinden vazgeçip son halini daha küçük bir download gerekecek şekilde aşağıdaki gibi de indirmeniz mümkündür. Ancak bu şekilde indirme işlemi yaptığınızda sonradan `git pull` yapamazsanız.

```
$ git clone --depth=1 https://github.com/raspberrypi/linux
```

Derleme

Derleme sürecine başlamadan önce kullanacağınız toolchain'in `PATH` ortam değişkeni içerisinde yer aldığından emin olunuz. Eğer değilse öncelikle toolchain ana dizinine geçip aşağıdaki komutları uygulayınız:

```
$ cd tools
$ export PATH=$PWD/arm-bcm2708/arm-bcm2708hardfp-linux-gnueabi/bin/:$PATH
```

Sonraki adımda Raspberry Pi 2 board'umuz için kernel kodu içerisinden çıkan öntanımlı konfigürasyonu uygulayalım:

```
$ make ARCH=arm bcm2709_defconfig
```

Kernel kodu ile birlikte gelen öntanımlı konfigürasyon dosyasında ihtiyaç duyulmayacak pek çok bileşen bulunmaktadır (Kullanmayacağımız network protokolleri, aygıt sürücüleri, dosya sistemleri gibi). Bu noktada mevcut kernel konfigürasyonu üzerinde ek iyileştirme çalışmaları için `menuconfig` aracını aşağıdaki gibi çalıştırıp konfigürasyonu iyileştirebilirsiniz:

```
$ make ARCH=arm menuconfig
```

Konfigürasyon süreci tamamlandıktan sonra derleme aşamasına geçebilirsiniz. Öncelikle sıkıştırılmış kernel imaj dosyası olan `zImage` 'i üretelim:

```
$ make ARCH=arm CROSS_COMPILE=arm-bcm2708hardfp-linux-gnueabi- -j4 zImage
```

İşlem bitiminde aşağıda dosya yolunda `arch/arm/boot/zImage` dosyası oluşacaktır:

Sonrasında Flattened Device Tree modeli için kullanılacak olan **dtb** dosyasını üretmek amacıyla aşağıdaki komutu çalıştırın:

```
$ make ARCH=arm CROSS_COMPILE=arm-bcm2708hardfp-linux-gnueabi- -j4 dtbs
```

İşlem tamamlandığında `arch/arm/boot/dts/bcm2709-rpi-2-b.dtb` dosyası oluşacaktır.

Sıradaki işlem konfigürasyon sürecinde modül olarak derlenmesi seçilen kernel modüllerinin derlenmesidir. Bunun için aşağıdaki komutu çalıştırın:

```
$ make ARCH=arm CROSS_COMPILE=arm-bcm2708hardfp-linux-gnueabi- -j4 modules
```

Modüllerin Kök Dosya Sistemine Aktarılması

Derlenmiş olan modüllerin uygun bir dizin hiyerarşisi altında toplanması gereklidir.

İlerleyen bölümde Raspberry Pi 2 board'umuz için kendi kök dosya sistemimizi `/opt/rpi` dizini altında oluşturacağız. Kernel modülleri ise çalıştığı sistemde `/lib/modules` dizini altında kernel versiyon numarası adıyla oluşturulan dizin ve alt dizinlerinde yer alacaktır.

Modülleri uygun dizin yapısıyla çıkartmak için gereken `make` hedefi `modules_install` şeklindedir. `INSTALL_MOD_PATH` değişkenini geliştireceğimiz kök dosya sistemini barındıracak olan ana dizini gösterecek şekilde ayarlamak suretiyle, modüllerin kurulumunu gerçekleştirebiliriz. Bunun için kullanacağımız komut aşağıdaki gibi olacaktır:

```
$ make ARCH=arm CROSS_COMPILE=arm-bcm2708hardfp-linux-gnueabi- \
INSTALL_MOD_PATH=/opt/rpi modules_install
```

İşlem bitiminde `/opt/rpi/lib/modules/4.4.13-v7+` ve `/opt/rpi/lib/firmware` dizin yapıları oluşacaktır. Derlemiş olduğunuz kernel versiyonuna göre ilkinin ismi değişkenlik gösterebilir. Derleme yaptığınız kernel içerisine dahil etmiş olduğunuz ve kernel kodu ile dağıtıma uygun

firmware bileşenleri içeren modüllere ait *firmware* dosyaları da kendilerine özgü dizin hiyerarşisi içerisinde oluşturulacaktır.

İşlemler tamamlandıktan sonra `/opt/rpi/lib/modules/4.4.13-v7+` ve `/opt/rpi/lib/firmware` dizinlerini incelemenizi öneririz.

U-boot Derleme Süreci

Kaynak Kodun Edinilmesi

U-Boot kodunu aşağıdaki gibi indirip derleyebilirsiniz.

```
$ git clone git://git.denx.de/u-boot.git
```

Yukarıdaki örnekte u-boot kodu mainline repo'dan çekildi. Alternatif olarak u-boot mainline kodu içerisinde Raspberry Pi desteği üzerinde çalışan **Stephen Warren**'in repo'sunu da kullanabilirsiniz. Yeni geliştirilen özellikleri henüz mainline koda entegre edilmeden denemeniz gerektiğinde de burayı kullanabilirsiniz:

```
$ git clone git://github.com/swarren/u-boot.git
$ cd u-boot
$ git checkout -b rpi_dev origin/rpi_dev
```

Derleme

Kaynak kodu indirmiş olduğunuz dizine geçtikten sonra aşağıdaki gibi öntanımlı konfigürasyonu baz alarak derleme yapabilirsiniz:

```
$ make rpi_2_defconfig
$ make CROSS_COMPILE=arm-bcm2708hardfp-linux-gnueabi- -j2
```

> **Not:** Çapraz derleyicinizin yol ifadesinin PATH çevre değişkeninde tanımlı olduğundan emin olunuz. Başka bir derleyici kullanıyorsanız **CROSS_COMPILE** çevre değişkenini, derleyicinizin önekini gösterecek şekilde değiştirmelisiniz.

İşlem tamamlandığında `u-boot.bin` dosyası ana dizinde oluşacaktır.

Dosya Sisteminin Hazırlanması

Bu bölümde Raspberry Pi board'umuz için **busybox** ile basit bir dosya sistemi hazırlanmasına yönelik çalışmalar anlatılacaktır. Dosya sistemi hazırlanmasıyla ilgili daha detaylı bilgiler için *Kök Dosya Sistemi Oluşturma* başlıklı bölümün de incelenmesini öneririz.

Busybox

Öncelikle güncel *busybox* kaynak kodlarını indirip diskimizde bir dizine açmalıyız:

```
$ wget http://www.busybox.net/downloads/busybox-1.23.2.tar.bz2
$ tar xf busybox-1.23.2.tar.bz2
$ cd busybox-1.23.2
```

Ardından öntanımlı konfigürasyonunu oluşturup, `PATH` ortam değişkenimiz içerisinde daha önce ayarlanmış olan toolchain önekini kullanarak çapraz derleme işlemimizi gerçekleştirebiliriz:

```
$ make defconfig
$ make CROSS_COMPILE=arm-bcm2708hardfp-linux-gnueabi- -j4
```

Derleme işlemi tamamlandıktan sonra `busybox` uygulaması ana dizinde oluşacaktır. Daha önce anlatıldığı gibi busybox uygulamasını içerdiği diğer uygulamalara ait linkleri de oluşturmak suretiyle kullanmamız gereklidir. Bu linkleri oluşturmak için `make install` komutunu verebiliriz ancak bu durumda ilgili linkler bulunduğumuz yerde `_install` adlı bir dizinde üretilecektir. Bunun yerine örneğin `/opt/rpi` şeklinde genel bir dizin açıp orayı kök dosya sistemi olarak hazırlamak daha yerinde bir karar olacaktır. Aşağıdaki gibi `CONFIG_PREFIX` değişkenine atama yapmak suretiyle linklerin ilgili dizinde oluşturulması sağlanabilir:

```
$ sudo mkdir /opt/rpi
$ sudo chown -R $USER /opt/rpi

# Yukarıdaki 2 komutla /opt/rpi dizinini oluşturup yazma yetkilerini
# kendi kullanıcımıza vermiş olduk. Bu amaçla herhangi başka bir dizini de kullanabili
rsiniz

$ make CROSS_COMPILE=arm-bcm2708hardfp-linux-gnueabi- CONFIG_PREFIX=/opt/rpi install
```

İşlem başarıyla tamamlandıktan sonra `/opt/rpi` altında aşağıdaki dosya ve dizinlerin oluştuğu görülecektir:

```
$ ls -l /opt/rpi/
total 8
drwxr-xr-x 2 demirten demirten 4096 Jun 23 15:59 bin
lrwxrwxrwx 1 demirten demirten   11 Jun 23 15:59 linuxrc -> bin/busybox
drwxr-xr-x 2 demirten demirten 4096 Jun 23 15:59 sbin
drwxr-xr-x 4 demirten demirten   27 Jun 23 15:59 usr
```

Yukarıdaki dizin yapısı kök dosya sistemimizin temelini oluşturmaktadır. Ancak bir kaç ek dokunuşa daha ihtiyaç olacaktır.

Sistemimizi yukarıdaki kök dosya sistemiyle açmaya çalışacak olursa, kernel tarafında aşağıdakine benzer bir hata alırız:

```
...
[    2.492921] devtmpfs: error mounting -2
...
```

Yukarıdaki hata mesajında `devtmpfs` dosya sisteminin **mount** edilemediği söylenmektedir. Konuyla ilgili daha ayrıntılı bilgiye *Devtmpfs Dosya Sistemi* bölümümüzden erişebilirsiniz. Bu sürecin doğru işlemesini sağlamak için hazırladığımız yeni kök dosya sistemi içerisinde `/dev` dizinini aşağıdaki komutla oluşturmalıyız:

```
$ mkdir /opt/rpi/dev
```

Kütüphanelerin Taşınması

Busybox sayesinde tek bir uygulama ile pek çok komuta birden sahip olduk, ancak busybox uygulamasının çalışabilmesi için gereken `libc.so.6`, `libm.so.6` ve `libcrypt.so.1` kütüphaneleri ile paylaşımlı kütüphane kullanan tüm uygulamalar için gerekli olan `ld-linux.so.3` kütüphanesi de kök dosya sistemimizde `/lib` dizini altında yer almalıdır.

Bunun için öncelikle `/lib` dizinimizi oluşturalım:

```
$ mkdir /opt/rpi/lib
```

Ardından ilgili kütüphanelerin kullandığımız toolchain içerisinden çıkan versiyonlarını bu dizin altına kopyalayalım. Bu noktada *Çapraz Derleme ve Gerekli Ekipmanlar* başlığından da faydalanabilirsiniz.

Raspberry Pi için daha önce edindiğimiz `tools` dizini altındaki `arm-bcm2708/arm-bcm2708hardfp-linux-gnueabi` toolchain'ini kullanmaktaydık. Burası toolchain ana dizinidir (Raspberry Pi için `tools` arşivinden 4 adet toolchain çıkmaktadır, biz burada örneklerimizde kullandığımız versiyon ile ilerliyoruz, diğer toolchain'leri kullandı iseniz sonraki komutları da bu doğrultuda değiştirmeniz gerekecektir).

Toolchain ana dizini altında `arm-bcm2708hardfp-linux-gnueabi --> lib` dizin yapısı içerisinde ihtiyaç duyduğumuz kütüphaneler bulunmaktadır. Toolchain ana dizininde iken aşağıdaki komutlarla bunları kök dosya sistemimize kopyalayalım:

```
$ cd arm-bcm2708hardfp-linux-gnueabi/lib
$ cp libc.so.6 libm.so.6 libcrypt.so.1 ld-linux.so.3 /opt/rpi/lib/
```

Artık hem **busybox** uygulamamız hem de uygulamanın çalışması için gereken tüm bileşenler hazır. Şimdi temel açılış ve kapanış betiklerimizi hazırlayalım.

Açılış Betiğinin Hazırlanması

Linux sistemlerde açılış sürecini yöneten SysV Init uygulamasının küçük bir versiyonu busybox içerisinden çıkmakta ve `/sbin/init` şeklinde kök dosya sistemi üzerinde linki üretilmektedir.

`init` uygulaması gömülü sistemimizde de ilk çalışacak uygulama olup, çalışmaya başladığında `/etc/inittab` dosyasını okumaktadır. Bu dosya mevcut değilse, kendisi öntanımlı ayarlarıyla açılacaktır. Açılış ve kapanış sürecini tam kontrolümüz altına almak için `/etc/inittab` dosyasını oluşturmalıyız. Dosyanın bir örneğine busybox kaynak kod dizini altında `examples/inittab` şeklinde erişebilirsiniz.

Bizim kullanacağımız temel `inittab` dosyası aşağıdaki gibi olacaktır:

```
::sysinit:/etc/acilis
::askfirst:-/bin/sh
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/etc/kapanis
```

Yukarıdaki dosya `init` uygulaması tarafından okunduğunda şunlar anlaşılır:

- Sistemde `init` tarafından çalıştırılacak ilk uygulama `/etc/acilis` olacaktır
- Sistemin çekirdek tarafında ayarlanmış olan konsol aygıtı üzerinde, *askfirst* yöntemiyle `/bin/sh` kabuğu çalıştırılacaktır
- Sisteme doğrudan bağlı bir klavye olması halinde CTRL-ALT-DEL tuş kombinasyonu

kullanıldığında `/sbin/reboot` uygulaması çalıştırılacaktır
- Sistem kapanış sürecine geçtiğinde `/etc/kapanis` uygulaması çalıştırılacaktır

`/etc/acilis` betiğinin küçük bir örneği aşağıdaki gibidir:

```
#! /bin/sh

echo "Sistem aciliyor..."

# Proc dosya sistemini mount edelim
mount -t proc none /proc

# Sysfs dosya sistemini mount edelim
mount -t sysfs none /sys

# Opsiyonel olarak /tmp dizinini bellek üzerinde
# tmpfs dosya sistemiyle oluşturalım
mount -t tmpfs none /tmp

# devtmpfs /dev üzerine overwrite edeceğinden
# mount etmeden önce dizini oluşturalım
mkdir -p /dev/pts && mount -t devpts none /dev/pts
```

Yukarıdaki açılış betiğine, kullanacağınız diğer servis ve uygulamaları başlatacak komutları da ekleyerek nihai sisteminizi üretebilirsiniz.

Kapanış betiğinde ise çalışan uygulamalarınıza `TERM` sinyali yollayıp biraz beklemek, halan kapanmadı iseler `KILL` sinyali göndererek uygulamaları sonlandırmak ve son adımda yazılabilir olarak kullandığınız disk bölümlerini unmount etmeli veya read-only (salt okunur) modda yeniden mount (remount) etmelisiniz. Yazılabilir modda mount edilmiş olan bir bölümü read-only remount etmek, Linux çekirdeği tarafından fiziksel olarak diske yazılmak için bekletilen tüm verilerin yazılmasını tetikler, böylelikle kapanışta veri kaybı yaşamamış olursunuz. Bu işlemi yapmamanız halinde, cihazın kontrolsüz kapatılmasındakine benzer sorunlarla karşılaşırsınız. Aşağıda basit bir `/etc/kapanis` betik örneği verilmiştir:

```
#! /bin/sh

echo "Sistem kapaniyor"

# Tüm uygulamalara TERM sinyali gönder
kill -s SIGTERM -1

# 3 saniye bekle
sleep 3

# Halen uygulama kaldı ise KILL sinyali gönder
kill -s SIGKILL -1

# Tüm mount edilmiş bölümlerin unmount edilmesini sağlayalım
# -r parametresi, umount işlemi başarısız olursa
# read-only modda remount edilmesini sağlar
umount -a -r
```

`quiet` Parametresi

Çekirdek boot parametreleri arasına (bir sonraki bölümde `cmdline.txt` dosyasına yazacağımız) `quiet` parametresini eklemeniz halinde açılış sırasında sistem konsoluna çok daha az mesaj çıkartılır.

Bu işlem açılış süresinin de bir miktar kısalmasını sağlamaktadır. Sistemi ilk ayağa kaldırırken tüm çekirdek açılış mesajlarını konsolda görmek yerinde olacaktır. Ancak sistem kararlı hale geldikten sonra `quiet` parametresini eklemek suretiyle süreci hızlandırmanız önerilir. Bu parametre ile açılışta göremediğiniz mesajlara sonraki bir anda `dmesg` komut çıktısından gene erişebilirsiniz.

Bu parametrenin mevcut çekirdek konfigürasyonumuzdaki etkisini test etmek için şöyle bir yöntem izleyebiliriz. Eğer kullanıcı kipinde çalıştırılan ilk uygulama (`init`) üzerinden yukarıdaki örnekte çalıştırdığımız `/etc/acilis` betiğimizde ilk iş olarak çekirdek mesajları tampon bölgesine bir mesaj gönderecek olursa, `quiet` parametresinin olduğu ve olmadığı açılışları karşılaştırabiliriz.

Çekirdek mesajlarının tutulduğu tampon alanına, `/dev/kmsg` özel aygıt dosyasına yazmak suretiyle biz de bir mesaj koyabiliriz. Yukarıdaki örneğini verdiğimiz `/etc/acilis` betiği içerisinde diğer komutları çalıştırmadan önce aşağıdaki satırı ekleyelim:

```
echo "quiet test mesaji" > /dev/kmsg
```

Ardından `quiet` parametresinin olduğu ve olmadığı durumlarda, `dmesg | grep "test mesaji"` komutu ile mesajların çıkartılma zamanlarına bakarak karşılaştıralım:

`quiet` olmadığında:

```
dmesg | grep "test mesaji"
[    2.698438] quiet test mesaji
```

`quiet` olduğu durumda:

```
dmesg | grep "test mesaji"
[    1.615022] quiet test mesaji
```

Görüldüğü üzere 1 saniyenin üzerinde bir zaman kazancı söz konusudur.

> Kullandığınız çekirdek konfigürasyonu, GPU tarafına ayrılan bellek (16 MB olduğunda GPU'daki bazı fonksiyonlar devre dışı kalmaktadır, Sistem Konfigürasyonu sayfasında detayına bakabilirsiniz), ARM CPU overclock ve turbo parametrelerinin durumuna göre bu süreler değişkenlik gösterebilir. Yaptığımız testlerde kullanıcı kipinde uygulama çalıştırmaya minimum 1.1 saniye içerisinde gelebildiğimizi gördük.

Cihazın Açılması

Raspberry Pi 2 yalnız microSD üzerinden açılabilmektedir. Bu sebeple SD kart üzerinde, biri boot dosyaları diğeri ise dosya sistemi için gerekli, iki adet bölümlendirme (partition) bulunmalıdır. FAT32 formatlı boot bölümlendirmesinde önyükleyiciler, çekirdek ve gerekli diğer dosyalar bulunurken, diğer bölüm cihazın tanıyabileceği formatta bir dosya sistemini içermektedir.

SD kart üzerine gerekli dosyaları kendimiz yazabileceğimiz gibi Raspberry Pi desteği olan bir dağıtımı da kullanabiliriz. Şimdi sırayla bu yöntemlere bakalım.

SD Karta Gerekli Dosyaların Kopyalanması

İlk olarak, SD kart üzerinde boot dosyalarını alabilecek büyüklükte (FAT32 dosya sisteminde minimum disk bölümü 32MB olabilir bununla birlikte boot yükleyici FAT16 ile de çalışmaktadır) bir FAT32 bölümü ve dosya sistemi için ext4 bölümü oluşturmalıyız. Bu amaçla Linux altında *fdisk* veya *gparted* gibi uygulamaları kullanabilirsiniz.

Linux altında SD kartları `/dev/mmcblk0` , `/dev/mmcblk1` vb. şeklinde tanınmaktadır. İlk tanına SD kart üzerinde oluşturularak birinci disk bölümünün aygıt ismi de `/dev/mmcblk0p1` şeklinde olacaktır (0 değil 1'den başladığına dikkat ediniz).

Disk bölümlendirmek genel olarak konumuzun dışında olsa da, örnek olması açısından aşağıda disk bölümlendirme tablosu (partition table) **tamamen boş olan** ve `/dev/mmcblk0` şeklinde algılanmış bir SD kart için gereken `fdisk` komutlarının pipe yöntemiyle verilip FAT32 türünde formatlanması gösterilmektedir:

```
$ echo "n
p
1

+64M

t
c
w
" | sudo fdisk /dev/mmcblk0 && sudo mkfs.vfat /dev/mmcblk0p1
```

FAT32 boot bölümünde asgari aşağıdaki dosyalara ihtiyaç duymaktayız.

Dosya	Görevi
bootcode.bin	Alt seviye önyükleyici
start.elf	GPU firmware
fixup.dat	SDRAM'in GPU ve CPU arasında bölümlendirilmesinde kullanılıyor
kernel.img	Kernel imajı
config.txt	*start.elf* tarafından okunan ayar dosyası
cmdline.txt	Kernel'a geçirilecek komut satırı argümanlarını barındıran dosya
bcm2709-rpi-2-b.dtb	Kernel derleme aşamasında üretilmiş olan Device Tree Block dosyası

Daha önce de söylediğimiz gibi *config.txt* dosyası ile sistem ayarları değiştirilebilmekte (ARM ve GPU frekansları, RAM paylaşımları) ayrıca önyükleyiciye müdahale edilebilmektedir. Bu dosyanın içeriğini *BIOS* ayarlarına benzetebiliriz. *config.txt* içinde yüklenecek çekirdeğe ilişkin herhangi bir atıf bulunmadığı sürece *start.elf*, *kernel.img* dosyasını yüklemekte ve ARM işlemciyi reset modundan çıkarmaktadır. Çekirdek imajı olarak önyükleyicilerle beraber gelen *kernel.img* dosyasını kullanabileceğiniz gibi daha önce derlediğimiz *zImage* dosyasını da *kernel.img* olarak kopyalayıp kullanabilirsiniz.

Örnek bir `config.txt` içeriği aşağıdaki gibidir:

```
kernel=kernel.img
```

Örnek bir `cmdline.txt` dosyasının içeriği ise aşağıdaki gibidir:

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200 console=tty1 root=/dev/mmcblk0p2 rootfstyp
e=ext4 elevator=deadline rootwait
```

> **ÖNEMLİ NOT:** Raspberry Pi için diğer kaynaklarda da yukarıdaki gibi bir örnek `cmdline.txt` dosyası ile karşılaşacaksınız. Burada verilen çekirdek açılış parametrelerinde `console` bölümünün 2 defa yer aldığına dikkat ediniz. Örneğimizde `console` öncelikle seri konsola atanmış, sonra bir de `tty1` aygıtı üzerine atanmış. Bu şekilde boot edilen bir çekirdek konsola gidecek mesajları her iki aygıta da gönderir (örneğimizde seri port ve HDMI üzerindeki tty konsolu) ancak sonuncu `console` parametresi olarak hangi aygıt belirtilmişse sadece oradan girdi alır. Yani yukarıdaki argümanlarla boot edecek olursanız konsol çıktıları hem seri portta hem de bağlı ise HDMI monitorde görünecek, Linux açıldıktan sonra ise seri konsoldan giriş yapamayacak, sistemdeki USB portuna takacağınız bir USB klavye ile giriş yapmak durumunda kalacaksınız. Giriş işlemleri için de seri konsolu kullanmak istiyorsanız `cmdline.txt` içerisinde sıralamayı uygun şekilde değiştirebilir veya `console=tty1` bölümünü tamamen kaldırabilirsiniz.

Temel bir dosya sisteminin nasıl oluşturulduğunu kitabımızın *Kök Dosya Sistemi Oluşturma* bölümünde bulabilirsiniz.

Kök Dosya Sisteminin SD Karta Aktarımı

Cihazı SD kart üzerindeki bir kök dosya sistemi ile açmak için, önceki aşamalarda hazırlamış olduğumuz kök dosya sistemini SD kartın ilgili bölümüne kopyalamamız gerekiyor.

Bunun için eğer henüz yoksa `fdisk` vb. bir disk bölümlendirme aracı ile SD kart üzerinde *Linux* tipinde (83) ikinci bir disk bölümü oluşturmalı, bu bölümü kullanmak istediğimiz dosya sisteminin türüne göre biçimlendirmeli ve sonrasında ilgili bölümü sistemimize *mount* edip hazırlamış olduğumuz kök dosya sistemini bu alana kopyalamalıyız.

Disk bölümünün `/dev/mmcblk0p2` ismiyle oluşturduğumuzu ve kök dosya sistemini `/opt/rpi` dizininde oluşturduğumuzu, geçici mount dizini olarak da `/mnt/tmp` 'yi kullanacağımızı varsayarsak komutlarımız şu şekilde olacaktır:

```
$ sudo mkfs.ext4 /dev/mmcblk0p2
$ sudo mount /dev/mmcblk0p2 /mnt/tmp
$ sudo cp -a /opt/rpi/* /mnt/tmp/
$ sudo umount /mnt/tmp
```

Artık sistemimiz SD kart üzerindeki kök dosya sistemi üzerinden açılmak için hazır.

Cihazın U-Boot Üzerinden Açılması

config.txt içine aşağıdaki gibi bir giriş eklediğimizde *start.elf* kernel yerine u-boot'u yükleyecek ve sonrasına ARM işlemciyi çalıştıracaktır. Öncesinde, SD kartın boot bölümüne daha önce derlediğimiz *zImage* ve *u-boot.bin* dosyalarını kopyalamalıyız.

```
kernel=u-boot.bin
```

u-boot komut satırına düştükten sonra çekirdek ve ilgili dtb dosyasını belleğin güvenli bir alanına çekip, çekirdeğe kök dosya sistemi ve konsolla ilgili değerleri geçirdikten sonra aşağıdaki gibi çalıştırabiliriz.

```
mmc dev 0
fatload mmc 0:1 0x01000000 zImage
fatload mmc 0:1 0x02000000 bcm2709-rpi-2-b.dtb
setenv bootargs earlyprintk console=ttyAMA0 console=tty1 root=/dev/mmcblk0p2 rootwait
bootz 0x01000000 - 0x02000000
```

> **DİKKAT:** 20 Haziran 2015 itibariyle Raspberry Pi 2 modeli için hazırlanmış u-boot uygulaması ile SD kart üzerinden okuma işlemi **80 KB/s** gibi oldukça düşük bir hızda çalışmaktadır. Bu nedenle çekirdek imajının yüklenmesi 40-50 saniye arasında zaman almakta olup yakın gelecekte sorunun giderilmiş olması beklenmektedir.

U-Boot Ortam Değişkenlerinin Düzenlenmesi

Bir önceki örneğimizde u-boot konsolu üzerinden komutlar girerek sistemin açılması gösterildi. Eğer her defasında komutları girmek yerine öntanımlı açılışın bu şekilde gerçekleşmesini istiyorsak, u-boot ortam değişkenlerini düzenleyerek non-interactive açılış sürecini yönlendirebiliriz.

Buraya kadar olan çalışmalarınızda dikkatinizi çekti ise, u-boot ilk açılırken aşağıdaki gibi mesajlar çıkmaktaydı:

```
reading uboot.env

** Unable to read "uboot.env" from mmc0:1 **
Using default environment
```

Görüleceği üzere ilk çalıştırmada `uboot.env` adlı bir dosya aranmakta ve bulunamadığı için öntanımlı ortam değişkenlerinin kullanılacağına dair bilgi verilmektedir.

`uboot.env` dosyasının formatı basit olmakla birlikte, bir metin dosyası olarak hazırlanması mümkün değildir (NULL değerler ve padding yapılması gerekecektir)

Bununla birlikte u-boot kodu içerisindeki `tools` dizini altında `mkenvimage` adlı yardımcı bir uygulama çıkmaktadır. Bu uygulama ile istenen `uboot.env` dosyasını oluşturabiliriz.

Öncelikle istediğimiz ortam değişkenlerini bir metin dosyasını yazmalıyız. Bunun için aşağıdaki satırları `/tmp/uboot.env.txt` adlı yeni bir geçici dosya oluşturup içerisine kopyalayın:

```
arch=arm
baudrate=115200
board=rpi_2
board_name=rpi_2
bootargs=earlyprintk console=ttyAMA0 console=tty1 root=/dev/mmcblk0p2 rootwait
bootcmd=run mmcboot
mmcboot=setenv devnum 0; mmc dev 0; fatload mmc 0:1 ${kernel_addr} zImage; fatload mmc
 0:1 ${fdt_addr} ${fdtfile}; bootz ${kernel_addr} - ${fdt_addr}
bootdelay=2
cpu=armv7
fdt_addr=0x2000000
fdtfile=bcm2709-rpi-2-b.dtb
kernel_addr=0x1000000
loadaddr=0x00200000
scriptaddr=0x00000000
soc=bcm283x
stderr=serial,lcd
stdin=serial,lcd
stdout=serial,lcd
usbethaddr=b8:27:eb:4c:66:85
vendor=raspberrypi
```

Ardından `mkenvimage` komutunu u-boot kaynak kod ana dizininde bulunduğunuzu varsayacak olursak aşağıdaki gibi çalıştırın:

```
$ tools/mkenvimage -s 0x4000 -o /tmp/uboot.env /tmp/uboot.env.txt
```

Örneğimizdeki `-s` parametresi oluşacak dosyanın boyutunu belirlememizi sağlamaktadır. Dosyanın 4 KB ve katları şeklinde olması beklenir. Biz burada 16 KB (0x4000) şeklinde belirtmiş olduk. Ortam değişkenlerimiz bu kadar yer kaplamıyor, kalan alan `0xFF` ile doldurulacak ve uygun *checksum* değeri hesaplanacaktır.

İşlem bitiminde oluşan `uboot.env` dosyasını SD kartımızın u-boot'u barındıran ilk bölümüne kopyaladığımızda bir sonraki açılışta yeni değerlerin aktifleştiğini (`printenv` komutuyla u-boot ekranında teyit edilebilir) göreceksiniz.

Cihazın Bir Dağıtım Üzerinden Açılması

Raspberry Pi, Raspbian, Ubuntu MATE, Ubuntu Snappy Core, OpenELEC ve RISC OS olmak üzere birçok sürüm tarafından desteklenmektedir. Biz incelememizde Debian tabanlı Raspbian dağıtımını kullanacağız. Aşağıdaki bağlantıdan Rasbian veya başka bir dağıtımı indirebilirsiniz.

```
https://www.raspberrypi.org/downloads/
```

Sonrasında elde ettiğimiz *2015-05-05-raspbian-wheezy.zip* dosyasını açmalı ve işletim sistemi imajını SD karta yazmalıyız. Sisteminizde SD karta ilişkin dosya düğümünü belirledikten sonra yazma işlemini *dd* ile yapabilirsiniz. Sistemimiz için örnek bir yazma aşağıdaki gibidir.

> **Not:** sudo fdisk -l ile sisteminizdeki bölümlendirme tablosunu listeleyebilirsiniz.

Yazma işleminden önce SD kartınızın dosya sisteminize mount edilmemiş olduğundan emin olunuz.

```
dd bs=4M if=2015-05-05-raspbian-wheezy.img of=/dev/sdb
```

> **Not:** dd ile yapılan kopyalama işleminin ilerleyişini izlemek için başka bir terminalde aşağıdaki komutu girebilirsiniz.
>
> pkill -USR1 -n -x dd

Bu işlem sonrasında SD kartınızda daha önce bizim elle kopyaladığımız dosyaları da içeren bir boot bölümü ve dosya sistemine ilişkin ikinci bir bölüm oluşmuş olmalıdır. SD kartı cihaza takıp çalıştırdığımızda işletim sistemi açılmalıdır.

Benzer işlem Raspberry Pi işletim sistemi yükleyicisi NOOBS ile de yapılabilmektedir. NOOBS, Raspbian dağıtımının içermekte ayrıca internet üzerinden indirdiği başka dağıtımları da cihaz üzerine kurabilmektedir. NOOBS yükleyicisini yine aşağıdaki adresten indirip, .zip dosyasını açtıktan sonra bir önceki adımda yaptığımız gibi *dd* ile SD karta yazabilirsiniz.

```
https://www.raspberrypi.org/downloads/
```

SD kartı cihaza takıp enerji verdiğinizde cihaz recovery modda açılacaktır. Cihazı HDMI üzerinden bir ekrana bağladıktan sonra, GUI üzerinden istediğiniz dağıtımı seçebilir ve cihaza yükleyebilirsiniz.

NFS Root Çalışma

Kök dosya sisteminin cihazın üzerindeki bir depolama biriminde değil de ağ üzerindeki bir NFS paylaşımında yer aldığı senaryo NFS Root olarak adlandırılmakta olup, *NFS Root Çalışma* başlıklı bölümde ayrıntılarıyla işlenmiştir.

Bu bölümde Raspberry Pi 2 özelinde NFS Root çalışma detaylarını inceleyeceğiz.

NFS Root çalışma yöntemleriyle ilgili destekler (`CONFIG_ROOT_NFS=y`)kernel imajı içerisine statik olarak eklenmiş olmalıdır.

Start.elf Üzerinden NFS Root Açılışı

Raspberry Pi cihazlarında GPU tarafından yüklenen *bootcode.bin* ve *start.elf* adlı boot yükleyiciler bulunmaktadır. Geleneksel *u-boot* yükleyicisi ile karşılaştırıldığında oldukça farklı olan bu yükleyiciler, nfs root çalışma senaryosu için benzer imkanlar sağlamaktadır.

Cihazla birlikte gelen *start.elf* boot yükleyicisi, `cmdline.txt` adlı bir dosya olması halinde burada yer alan kernel commandline parametrelerini `ATAGS` yöntemiyle çalıştırılacak kernel'a iletmektedir (ARM Mimarisinde Açılış Süreci bölümüne bakabilirsiniz).

NFS üzerinden kök dosya sistemini bağlayabilmek için `cmdline.txt` dosyasına aşağıdaki satırın girilmesi yeterli olacaktır:

```
console=ttyAMA0,115200 root=/dev/nfs rw nfsroot=192.168.0.5:/opt/rpi ip=192.168.0.6:::
::eth0 rootwait
```

Eğer DHCP üzerinden ip alınacak ise, `ip=dhcp` şeklinde de belirtebilirsiniz. Bu senaryonun çalışması için kernel içerisinde `CONFIG_IP_PNP_DHCP=y` şeklinde seçilmiş olmalıdır.

U-Boot Üzerinden NFS Root Açılışı

Raspberry Pi 2 üzerinde u-boot çalıştırmak, Haziran 2015 itibariyle halen çeşitli önemli sorunlara yol açmaktadır. U-boot içerisinde ethernet kartının çalışır hale getirilmesi her zaman mümkün olmamakta ve cihazı resetleyip yeniden denemek veya u-boot içindeyken dhcp komutuyla ip almayı denemek gerekmektedir. Kernel imajını ağdan yüklemek yerine SD karttan okumak istediğimizde okuma performansı çok düşük olmaktadır.

Bahsettiğimiz olumsuzluklara rağmen u-boot'un kullanım yaygınlığı ve önümüzdeki aylarda bu problemlerin giderilme olasılığı düşünülerek ayrıntılı olarak incelenmiştir.

U-boot üzerinden açılışta en kritik nokta, *start.elf* örneğinde olduğu gibi doğru command line parametreleri ile kernel imajının çalışmaya başlamasını sağlayabilmektir.

Kernel imajını ve Flattened Device Tree (FDT) tanım dosyasını yükleme yöntemimize göre süreç bir miktar değişmektedir. Aşağıda bu yöntemlerden ağ üzerinden yükleme ve sd kart üzerinden yükleme seçenekleri incelenmiştir.

Kernel ve FDT'nin Ağ Üzerinden Yüklenmesi

Kullanılacak kernel imajı ve fdt dosyası TFTP sunucu ana dizinine konulmalıdır.

U-boot komut satırında ethernet'i kullanmaya başlayabilmek için öncelikle aşağıdaki komut verilmelidir:

```
usb start
```

Ardından u-boot'un standart ortam değişkenleri üzerinden ip atama yöntemiyle cihazımızın ve TFTP sunucunun ip adresi belirtilmelidir. Örneğimizde cihaz ip adresi 192.168.0.6, sunucu ip adresi ise 192.168.0.5 şeklindedir.

```
setenv ipaddr 192.168.0.6
setenv serverip 192.168.0.5
```

Ardından `zImage` kernel imaj dosyasını TFTP sunucudan `0x1000000` gibi bir adrese çekelim:

```
tftp 0x1000000 zImage
```

Aynı şekilde cihazımız için TFTP sunucu ana dizinine koyduğumuz `bcm2709-rpi-2-b.dtb` FDT dosyasını `0x2000000` gibi bir adrese çekelim:

```
tftp 0x2000000 bcm2709-rpi-2-b.dtb
```

Kernel ve FDT imajları bellekte ve adreslerini biliyoruz. Sonraki adımda command parametrelerini hazırlayabilmek için `bootargs` u-boot değişkenini aşağıdaki gibi düzenlemeliyiz:

```
setenv bootargs console=ttyAMA0,115200 root=/dev/nfs nfsroot=192.168.0.5:/opt/rpi ip=1
92.168.0.6
```

Son olarak `bootz` komutuna kernel imajının ve FDT imajının bellekteki adreslerini parametre olarak verip Linux açılış sürecini başlatalım:

```
bootz 0x1000000 - 0x2000000
```

Özet olarak aşağıdaki gibi çıktılar almış olmalıyız:

```
U-Boot> setenv ipaddr 192.168.0.6
U-Boot> setenv serverip 192.168.0.5
U-Boot> tftp 0x1000000 zImage
Waiting for Ethernet connection... done.
Using sms0 device
TFTP from server 192.168.0.5; our IP address is 192.168.0.6
Filename 'zImage'.
Load address: 0x1000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #
         945.3 KiB/s
done
Bytes transferred = 3819416 (3a4798 hex)
U-Boot> tftp 0x2000000 bcm2709-rpi-2-b.dtb
Waiting for Ethernet connection... done.
Using sms0 device
TFTP from server 192.168.0.5; our IP address is 192.168.0.6
Filename 'bcm2709-rpi-2-b.dtb'.
Load address: 0x2000000
Loading: #
         861.3 KiB/s
done
Bytes transferred = 9703 (25e7 hex)
U-Boot> setenv bootargs console=ttyAMA0,115200 root=/dev/nfs nfsroot=172.16.2.64:/opt/
rpi,vers=3 ip=192.168.0.6
U-Boot> bootz 0x1000000 - 0x2000000
## Flattened Device Tree blob at 02000000
   Booting using the fdt blob at 0x2000000
   Loading Kernel Image ... OK
   Loading Device Tree to 07b43000, end 07b485e6 ... OK

Starting kernel ...
Uncompressing Linux... done, booting the kernel.
...
[    5.008131] smsc95xx 1-1.1:1.0 eth0: link up, 100Mbps, full-duplex, lpa 0xC1E1
[   10.243929] IP-Config: Complete:
[   10.247276]      device=eth0, hwaddr=66:82:1a:78:a9:2f, ipaddr=192.168.0.6, mask=25
5.255.255.0, gw=172.16.2.254
[   10.257516]      host=192.168.0.6, domain=, nis-domain=(none)
[   10.263368]      bootserver=192.168.0.5, rootserver=192.168.0.5, rootpath=
[   10.282394] VFS: Mounted root (nfs filesystem) readonly on device 0:14.
[   10.289601] devtmpfs: mounted
[   10.293252] Freeing unused kernel memory: 388K (8072d000 - 8078e000)
Sistem aciliyor...
[   10.534755] random: nonblocking pool is initialized

Please press Enter to activate this console.
```

Kernel ve FDT'nin SD Kart Üzerinden Yüklenmesi

Raspberry Board Konfigürasyonu

Raspberry Pi cihazınız açılırken boot yükleyici uygulama tarafından SD kart üzerindeki birinci FAT bölümünde `config.txt` adlı dosya aranır ve bu dosyada yer alan tanımlara göre cihazın çalışma biçimine dair bir takım özellikler ayarlanır.

Bu süreci PC mimarisindeki BIOS yazılımları ile benzeştirebiliriz. BIOS'lardan farklı olarak konfigürasyon için ekran üzerinde bir arayüz sunulamadığı için, tüm ayarlar `config.txt` adındaki dosya üzerinde yapılır.

Değişikliklerin aktif olması için sistemin yeniden başlatılması gereklidir. `config.txt` dosyası **GPU** tarafından çalıştırılan boot yükleyici içerisinden işlenmektedir. Dolayısıyla **CPU** ve Linux tarafı henüz başlamadan önce sistemle ilgili önemli ayarlamaların yapılabilmesi sağlanmaktadır.

Örnek olarak GPU ve CPU arasında belleğin hangi oranlarda paylaştırılacağı bu dosyadan ayarlanabilir. Bunun için `gpu_mem` konfigürasyon anahtar sözcüğü kullanılır. Eğer 1 GB'lık bir Raspberry Board'unuz var ve GPU tarafına 128 MB ayırmak istiyorsanız `config.txt` içerisine aşağıdaki satırı eklemelisiniz:

```
gpu_mem=128
```

Yukarıdaki satır boot yükleyici tarafından işlendiğinde 1 GB belleğin 128 MB'ı GPU kullanımına ayrılır, kalan 872 MB ise CPU (ve Linux) tarafından kullanılabilecek hale gelir. 1 GB bellekli modellerde GPU için ayırabileceğiniz minimum alan **16 MB**, maksimum ise **944 MB** olabilir.

Benzer şekilde CPU saat frekansı, GPU saat frekansı, SDRAM frekansı, H264 Video Decoder lisansları vb. ayarlanabilir. Aşağıdaki bölümlerde başlıca konfigürasyon bloklarını inceleyeceğiz.

Bellek Konfigürasyonu

`gpu_mem`

Sistemde yer alan GPU'nun belleğin ne kadarını kullanacağını bu şekilde belirtebiliriz. Raspberry Pi modelleri arasında bellek farklılıkları bulunduğundan 256 MB, 512 MB ve 1024 MB belleğe sahip Raspberry Pi modellerinde GPU için ayrılabilecek minimum ve maksimum değerler aşağıdaki tablodaki gibidir:

Raspberry Pi Modeli	Minimum GPU Belleği	Maksimum GPU Belleği
256 MB	16 MB	192 MB
512 MB	16 MB	448 MB
1024 MB	16 MB	944 MB

Bazen aynı SD kart imajını her 3 model Raspberry Pi cihazında kullanmak isteyebilirsiniz. Bu senaryoda farklı modellere yönelik GPU bellek miktarını tanımlayabilmek için, 256 MB bellekli modellerde `gpu_mem_256`, 512 MB bellekli modellerde `gpu_mem_512` ve 1024 MB bellekli modellerde `gpu_mem_1024` parametrelerini kullanabilirsiniz.

ÖNEMLİ: `gpu_mem=16` şeklindeki bir tanımla GPU tarafına minimum bellek ayırmanız gerekirse, standart `start.elf` ve `fixup.dat` dosyaları ile sistem boot edilememektedir. Bunun için her iki firmware dosyasının **c**ut-**d**own (kırpılmış) versiyonları kullanılmalıdır. Raspberry Pi Firmware arşivininden `start_cd.elf` ve `fixup_cd.dat` dosyalarını da SD kartınızın ilk FAT bölümüne kopyalamak suretiyle açılışı gerçekleştirebilirsiniz. GPU tarafına 16 MB bellek ayrıldığında, GPU'nun bazı özellikleri de devre dışı bırakılmakta ve daha hızlı bir açılış gerçekleşmektedir. Ancak bu açılış için yukarıda bahsettiğimiz dosyaların da SD kart üzerinde bulunması zorunludur.

`disable_l2cache`

Öntanımlı değeri **0** olup, bu değeri **1** yapmanız halinde ARM CPU'nun GPU üzerindeki Level 2 Cache alanına erişimini engelleyebilirsiniz. Özellikle GPU yoğun uygulamalarda bu özelliğin kullanılması tercih edilebilir.

`config.txt` üzerinden GPU Level 2 Cache devre dışı bırakıldığında mevcut Linux çekirdek imajınızla ARM CPU tarafında boot gerçekleştiremezsiniz. Boot edebilmek için Linux çekirdeğinin de GPU Level 2 Cache alanını kullanmayacağını bilecek şekilde derlenmiş olması gereklidir. Çekirdek konfigürasyonunda `CONFIG_BCM2708_NOL2CACHE=y` yapıp yeniden derleyerek boot işlemini gerçekleştirebilirsiniz.

`disable_pvt`

Öntanımlı değeri **0** olup, **1** yapılması halinde her 500 milisaniyede bir DRAM sıcaklığının kontrol edilmesi fonksiyonunu devre dışı bırakmaktadır. Her bir kontrol ortalama 16µs zaman almaktadır. Eğer ortam şartlarından eminseniz az da olsa performansı artırmak için bu özelliği kullanmak isteyebilirsiniz.

Grafik Ekran ve Monitor Konfigürasyonları

`display_rotate`

Ekranı döndürmek veya yatay/dikey oryantasyonunu değiştirmek için kullanılır. Aşağıdaki tabloda display_rotate değerlerine karşılık gerçekleşen durum bilgileri yer almaktadır:

Display_rotate değeri	Sonuç
0	Herhangi bir dönüşüm uygulanmaz
1	Saat yönünde 90 derece döndür
2	Saat yönünde 180 derece döndür
3	Saat yönünde 270 derece döndür
0x10000	Görüntünün yatay katlanmış hali
0x20000	Görüntünün dikey katlanmış hali

> **Not:** 90 ve 270 derece döndürme işlemlerinde bir miktar ek GPU belleği gerektiğinden, GPU için **16 MB** bellek ayırmanız halinde çalışmayacaktır. Bunun için GPU bölümüne minimum **32 MB** bellek ayırmalısınız.

`hdmi_safe`

Bu değerin **1** yapılması halinde HDMI monitorlerde sık karşılaşılan otomatik algılama sorunlarına karşılık çeşitli workaround'ler devreye alınır. Eğer HDMI ekranında görüntü alamıyorsanız bu opsiyonun değerini 1 yapıp test etmeniz önerilir.

`hdmi_safe=1` durumu aşağıdaki konfigürasyona eşittir:

```
hdmi_force_hotplug=1
hdmi_ignore_edid=0xa5000080
config_hdmi_boost=4
hdmi_group=2
hdmi_mode=4
disable_overscan=0
overscan_left=24
overscan_right=24
overscan_top=24
overscan_bottom=24
```

Yukarıda yer alan her bir konfigürasyon anahtar deyiminin detaylarına buradan erişebilirsiniz.

`disable_splash`

Bu değerin 1 yapılması durumunda, açılış sürecindeki test sırasında gökkuşağı logosunun gösterimi engellenir.

Bazı HDMI monitorlerde algılama süreci uzun sürdüğünden bu logoyu hiç görmemiş de olabilirsiniz. Normalde cihazınız açılmaya başladığında ilk olarak HDMI üzerinde bu logoyu göstermektedir.

`bootcode_delay`

Değer girilmesi halinde `bootcode.bin` yükleyicisi `start.elf` ikinci aşama boot yükleyici yüklemeden önce girilen değer kadar saniye boyunca bekler.

Bu opsiyon Raspberry Pi cihazı ile HDMI monitorün aynı güç kaynağından beslendiği ve HDMI monitor tarafında ilklendirme işlemlerinin uzun sürdüğü senaryolarda faydalı olmaktadır.

Overclock Parametreleri

Raspberry Pi, overclock işlemleri için çok sayıda konfigürasyon opsiyonu sunmaktadır. ARM CPU frekansı, GPU içerisinde ayrı ayrı h264 bloğu, 3D bloğu, SDRAM frekansı vb. ayarlanabilmektedir. İlgilenenler detaylı bilgiye buradan ulaşabilir.

Önceleri Raspberry tarafından önerilen overclock fonksiyonlarının dışında bir parametre kullanıldığında cihaz garanti dışı kalıyordu. Daha sonradan `force_turbo` parametresi ile aktive edebileceğiniz turbo mode geliştirildi. Turbo mode aktif cihazlarda çalışma frekansları sistem meşgul olduğunda artırılmakta, sistem sıcaklığı 85°C üzerine çıktığında ise düşürülmektedir. Bu şekildeki kullanım garanti kapsamına dahildir.

Codec Lisans Aktivasyonu

Raspberry Pi board'unun güçlü olduğu yanlardan biri de donanım destekli h264 encode ve decode işlemidir. Bu özelliğin aktivasyonu her bir board için lisans satın alınarak yapılmaktadır.

Halihazırda **MPEG-2** ve **VC-1** olmak üzere 2 tip codec satın alınabilmektedir. VC-1 codec'i sadece Windows Media Video (WMV) dosyalarını oynatmak istiyorsanız gereklidir.

Satın alma işlemleri için http://swag.raspberrypi.org/collections/software adresini ziyaret etmelisiniz. Lisanslar CPU seri numaralarına göre yapıldığından, satın alma işleminden önce sisteminizi boot edip `/proc/cpuinfo` dosyasından seri numarasını öğrenmelisiniz:

```
# cat /proc/cpuinfo
...
Hardware        : BCM2709
Revision        : a01041
Serial          : 00000000634c6685
```

Temmuz 2015 itibariyle MPEG-2 codec lisansı cihaz başına 2£ (yaklaşık 8,5 TL) civarındadır.

Lisansınız geldikten sonra `config.txt` içerisinde bu lisansın ilgili olduğu CPU seri numarasını belirtmeniz gerekiyor. Örnek olarak yukarıdaki CPU için MPEG-2 lisansı alınmışsa girmemiz gereken parametre şu şekilde olacaktır:

```
decode_MPG2=0x634c6685
```

Eğer VC-1 codec lisansı almış olsaydık bu durumda aşağıdaki parametreyi girmeliydik:

```
decode_WVC1=0x634c6685
```

Bazen aynı SD kart imajını birden fazla üründe kullanmanız gerekebilir. Ancak sözkonusu cihazlar için ayrı ayrı codec lisansları alınmış ise, her bir cihazdaki `config.txt` dosyasındaki ilgili parametrelerin düzenlenmesi gereklidir.

Raspberry Pi bu durumu kolaylaştırmak için yukarıdaki konfigürasyon parametrelerine virgül ile ayrılmış şekilde **8** adede kadar CPU seri numarası yazılmasına imkan vermiştir. Aşağıda 3 farklı CPU seri numarasının yer aldığı bir örneği görebilirsiniz:

```
decode_MPG2=0x634c6685,0x1234567,0x99889922
```

Her ne kadar birden çok seri numarası yazımı mümkün olsa da, sahada büyük ölçekli bir kullanım içi 8 adet oldukça düşük bir sayı olarak kalmaktadır. Böyle bir ürün içerisinde kullanmak istiyorsanız lisans yönetimi için yazılım tarafında süreci kolaylaştırıcı ek mekanizmalar geliştirmeyi düşünmelisiniz.

Raspberry Pi 3

Bu başlık altında 64 bitlik işlemciye sahip Raspberry Pi 3 Model B için, 64 bit hedefli, u-boot ve Linux çekirdeğini nasıl derleyebileceğimize bakacağız.

Raspberry Pi 3 ve Raspberry Pi 2 için geliştirme süreçleri oldukça benzerdir. Sadece u-boot, kernel default ayarları ve seri portun kullanımına ilişkin bir takım farklılıklar bulunmaktadır. Yeri geldiğinde bu farklılıklara değineceğiz. Raspberry Pi 3 ile Raspberry Pi 2 fiziksel seri konsol bağlantıları aynıdır.

Raspberry Pi 3 Teknik Özellikleri

Cihazın temel teknik özelliklerini aşağıdaki gibi listeleyebiliriz.

Özellik	Açıklama
SoC	Broadcom BCM2837
CPU	64-bit quad-core ARM Cortex-A53, 1.2GHz
GPU	Broadcom Dual Core VideoCore IV
RAM	1GB LPDDR2 (900 MHz) (GPU ve CPU paylaşımlı)
Video çıkışı	HDMI and RCA, plus 1 CSI camera connector
Hafıza	MicroSD
Giriş/Çıkış	17 GPIO plus specific functions
Network	10/100 Ethernet, 802.11n WiFi and Bluetooth 4.1

Raspberry Pi 2 ve 3 arasındaki temel özelliklere ilişkin karşılaştırma ise aşağıdaki gibidir:

Raspberry Pi 2	Raspberry Pi 3
Processor: 32-bit quad-core ARM Cortex-A7	Processor: 64-bit quad-core ARM Cortex-A53
Clock frequency: 1000 MHz	Clock frequency: 1200 MHz
RAM: 1024 MB	RAM: 1024 MB
Wi-Fi : Yok	Wi-Fi : Var
Bluetooth : Yok	Bluetooth : Var, 4.1
Power supply : 5v 2A	Power supply : 5v 2.5A
Storage: MicroSD card	Storage: MicroSD card
Network adapter: Ethernet network card	Network adapter: Ethernet network card
USB ports: 4	USB ports: 4
2.5A USB: No	2.5A USB: Yes

Geliştirme Araçlarının Elde Edilmesi

Gerekli araç ve üretilen dosyaları uygun bir dizin yapısı altında tutmak geliştirme sürecinde faydalı olacaktır. Örnek olarak kendi sistemimizde aşağıdaki dizin yapısını oluşturduk. Daha sonra linux ve u-boot dizinlerini yine bu dizin yapısı altına ekleyeceğiz.

```
/opt/rpi3
└── toolchain
```

Geliştirme ana dizinini **DEVELOPMENT_DIR** isimli bir çevre değişkeninde saklayabiliriz.

```
$ export DEVELOPMENT_DIR=/opt/rpi3
```

DEVELOPMENT_DIR çevre değişkeni ile belirlediğiniz başka bir dizini göstererek geliştirmelerinizi o şekilde yapabilirsiniz.

İlk olarak çapraz derleyiciyi nasıl edinebileceğimize bakalım. 64 bit hedefli **Linaro** çapraz derleme araçlarını aşağıdaki gibi sisteminize indirip **toolchain** dizini altına açabilirsiniz.

```
$ cd $DEVELOPMENT_DIR/toolchain

$ wget -c https://releases.linaro.org/components/toolchain/binaries/latest-7/\
           aarch64-linux-gnu/gcc-linaro-7.3.1-2018.05-x86_64_aarch64-linux-gnu.tar.xz

$ tar xvf gcc-linaro-7.3.1-2018.05-x86_64_aarch64-linux-gnu.tar.xz --strip-components=
1
```

Artık toolchain dizini altında ihtiyaç duyduğumuz çapraz derleme araçları bulunmaktadır. **PATH** çevre değişkenine çapraz derleyicinin konumunu aşağıdaki gibi ekleyebiliriz.

```
$ export PATH=$DEVELOPMENT_DIR/toolchain/bin:$PATH
```

Bu aşamadan sonra bulunduğumuz terminal üzerinde çapraz derleyiciyi kullanabiliriz, aşağıdaki gibi test edebilirsiniz.

```
$ aarch64-linux-gnu-gcc -v
```

Kernel Derleme Süreci

Raspberry Pi için kernel kaynak kodunu aşağıdaki gibi indirebiliriz.

```
$ cd $DEVELOPMENT_DIR
$ git clone --depth=1 https://github.com/raspberrypi/linux
```

Çapraz derleyicinin PATH değişkeninde tanımlı olduğundan emin olduktan sonra derleme işlemine geçilebilir.

```
$ export PATH=$DEVELOPMENT_DIR/toolchain/bin:$PATH
```

Raspberry Pi 3 için öntanımlı konfig dosyası **bcmrpi3_defconfig** olarak isimlendirilmiştir. Kernel konfigürasyonu ve derleme işlemleri aşağıdaki gibidir:

```
$ cd linux/
$ make ARCH=arm64 bcmrpi3_defconfig
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j4
```

Bu işlem sonucunda **arch/arm64** dizini altında, uygun alt dizinler içerisinde, kernel imajı ve gerekli dtb dosyası oluşacaktır. Kernel imajı ve dtb dosyasının konumları aşağıdaki gibi olacaktır.

```
arch/arm64/boot/Image
arch/arm64/boot/dts/broadcom/bcm2837-rpi-3-b.dtb
```

dtb dosyası olarak **bcm2837-rpi-3-b.dtb** dosyasını kullanacağız. Raspberry Pi 3 üzerinde Broadcom **BCM2837** SoC kulllanıldığını hatırlayınız.

Derleme işlemi sonunda oluşan kernel imajının **uncompressed** olduğuna dikkat ediniz. u-boot üzerinden kernel'ı yüklerken sıkıştırılmamış bu imajı kullanacağız.

U-boot Derleme Süreci

U-boot kaynak kodunu aşağıdaki gibi indirebiliriz.

```
$ cd $DEVELOPMENT_DIR
$ git clone git://git.denx.de/u-boot.git
```

Bu aşamadan sonra u-boot kodunu rp3 için konfigure edip derleyebiliriz.

```
$ cd u-boot/
$ make rpi_3_defconfig
$ make CROSS_COMPILE=aarch64-linux-gnu- -j2
```

Derleme sonunda ana dizinde **u-boot.bin** dosyası oluşacaktır.

Bu aşamadan sonra daha önceki bölümlerde anlatıldığı gibi gerekli boot dosyaları ve dosya sistemini içeren bir SD kart hazırlanmalıdır.

SD Kart Boot Bölümlendirmesi

Raspberrp Pi 2'de olduğu gibi yine gerekli araçlar **https://github.com/raspberrypi/firmware/tree/master/boot** adresinden indirilebilir. Bu adresten **bootcode.bin**, **fixup.dat** ve **start.elf** dosyalarını indirebilirsiniz. Raspberry Pi 2 ve 3 arasında bu dosyalarla ilgili bir farklılık bulunmamaktadır. Bu dosyalarla birlikte, derlediğimiz kernel imajını, dtb ve u-boot dosyalarını boot bölümlendirmesine kopyalamalıyız. Derlenen dosyalar **Image**, **bcm2837-rpi-3-b.dtb** ve **u-boot.bin** şeklinde isimlendirilmiş olacaktır.

Cihaz direkt kernel üzerinden açılabildiği gibi öncesinde u-boot önyükleyicisini çalıştırmak da mümkündür. Burada u-boot üzerinden cihazı nasıl açabileceğimizi inceleyeceğiz. u-boot.bin dosyasını SD karta **kernel8.img** olarak kopyalayacağız. Bu isim değişikliğinin nedenine

aşağıda değindik.

Tüm kopyalama işlemleri sonunda SD kartın boot bölümlendirmesinin içeriği aşağıdaki gibi olmalıdır.

```
├── bcm2837-rpi-3-b.dtb
├── bootcode.bin
├── config.txt
├── fixup.dat
├── Image
├── kernel8.img (u-boot.bin dosyası)
└── start.elf
```

Bu noktada cihazın nasıl açıldığını hatırlamak faydalı olacaktır. Cihaza enerji verildiğinde ilk olarak ROM bellekteki alt seviye önyükleyici, GPU içindeki, bir RISC işlemci tarafından çalıştırılır. Bu önyükleyici tarafından SD kartın FAT bölümlendirmesi mount edildikten sonra **bootcode.bin** çalıştırılır. bootcode.bin yükleyicisi de yine bir önyükleyici olan **start.elf** isimli GPU firmware'ini çalıştırır. Bu aşamaya kadar süreç dışarıdan bir müdahale olmaksızın gerçekleşmektedir. start.elf çalıştıktan sonra config.txt içeriğini okumaktadır. Bu sayede **config.txt** dosyası üzerinde değişiklik yapılarak start.elf yükleyicisinin çalışmasına müdahale edilebilmektedir.

Çalıştırılacak olan kernel imajı config.txt içerisinde **kernel=Image** şeklinde gösterilebilir. Raspberry Pi 2 için, bir önceki bölümde incelediğimiz gibi config.txt dosyasına, **kernel=u-boot.bin** şeklinde bir giriş eklenirse ilk olarak kernel yerine u-boot çalıştırılacaktır. Raspberry Pi 3 için ise bu yöntem işe yaramamaktadır. Raspberry Pi 2 ve 3 için default kernel imaj adı kernel7.img'dir. Raspberry Pi 3 önyükleyicisi, start.elf, **kernel8.img** isimli bir dosya görmesi durumunda öncelikli olarak bu dosyayı 64 bitlik modda yüklemektedir.

Seri Konsol

Raspberry Pi 3, PL011 ve mini olmak üzere iki adet UART'a sahiptir. Bu sebeple daha önce incelediğimiz Raspberry Pi 2 modeline göre config.txt ve kernel komut satır argümanlarında bazı farklılıklar bulunmaktadır.

Pi 3 modelinde PL011 default durumda Bluetooth modülü için kullanılmaktadır. Bu sebeple seri konsol için mini UART'ı kullanacağız. Linux tarafından bu aygıt **ttyS0** olarak tanınmaktadır.

> **Not:** Pi 2 modeli için seri konsol olarak PL011 kullanılmakta ve bu aygıt Linux tarafından **ttyAMA0** olarak isimlendirilmektedir. Pi 2 için seri konsol olarak ttyAMA0 kullandığımızı hatırlayınız.

Cihaz açılırken default durumda mini UART aktif değildir. Bu sebeple **config.txt** dosyasına aşağıdaki satır eklenmelidir.

```
enable_uart=1
```

Bu aşamadan sonra cihaza enerji verdiğimizde ilk olarak u-boot çalışacaktır. Bu sırada klavyeden bir tuşa basarak u-boot komut satırına geçebilirsiniz.

U-boot Komut Satırı

u-boot komut satırında ilk olarak kernel imaj ve dtb dosyasını SD karttan belleğin güvenli alanlarına çekelim.

```
U-Boot> fatload mmc 0 ${kernel_addr_r} Image
U-Boot> fatload mmc 0 ${fdt_addr_r} bcm2837-rpi-3-b.dtb
```

kernel_addr_r ve **fdt_addr_r** değişkenlerine uygun adresler atanmış durumdadır. Daha sonra kernel komut satırı argümanlarını **bootargs** değişkeninde tutabiliriz.

```
U-Boot> setenv bootargs 8250.nr_uarts=1 root=/dev/mmcblk0p2 rootwait\
                console=tty1 console=ttyS0,115200
```

Seri konsol için ttyS0'ı kullanıyoruz ayrıca kernel'a **8250.nr_uarts=1** ile kullanılacak UART sayısını geçiriyoruz.

Bu aşamadan sonra uncompressed kernel imajını **booti** komutu ile aşağıdaki gibi çalıştırabiliriz.

```
U-Boot> booti ${kernel_addr_r} - ${fdt_addr_r}
```

initrd imajı kullanmadığımız için booti komutuna 2. argüman olarak - değerini geçirdik. Sonrasında komut satırndan boot diyerek Linux çekirdeğini çalıştırabilirsiniz.

```
U-Boot> boot
```

Bu işlemleri sürekli tekrarlamamak için, cihazın açılması bölümünde anlatıldığı gibi, bir **uboot.env** dosyası hazırlanabilir veya bu ayarları **saveenv** komutu ile SD kart üzerinde saklayabiliriz. Burada ikinci yöntemi kullanacağız. Yukarıdaki işlemler aşağıdaki gibi kalıcı hale getirilebilir.

```
U-Boot> setenv bootargs "8250.nr_uarts=1 root=/dev/mmcblk0p2 rootwait console=tty1\
                console=ttyS0,115200"

U-Boot> setenv bootcmd "fatload mmc 0 ${kernel_addr_r} Image;\
                fatload mmc 0 ${fdt_addr_r} bcm2837-rpi-3-b.dtb;\
                booti ${kernel_addr_r} - ${fdt_addr_r}"

U-Boot> saveenv
```

Cihaz sonraki açılışlarda u-boot önyükleyicisi **bootcmd** değişkeni ile gösterilen işlemleri otomatik olarak yapacaktır.

Board Spesifik Kılavuzlar

Bu bölümde özellikle kişisel kullanım için uygun örnek board'lar hakkında çeşitli notlara yer vereceğiz.

Bazı ürünlerde belirli bir kernel repository'sini kullanmak, çeşitli yamalar uygulamak vb. gerekebilmektedir. Kullandığınız board'a özgü ek kılavuz burada yer alıyorsa incelemenizi öneririz.

Hawkboard

Hawkboard, TI (*Texas Instruments*) firmasının geliştirdiği, OMAP ARM9 (*OMAP-L138*) işlemcisini kullanan açık bir donanımdır.

Temel olarak aşağıdaki bileşenlere sahiptir.

İşlemci	**ARMV5/ARM926**
DSP	C674x Floating Point
RAM	128 MB DDR
NAND	128 MB
MMC/SD	Var
Ethernet	Var
VGA	Var
Konsol	Var

> **Not:** Hawkboard projesi, kartın bazı donanım problemleri nedeniyle sonlandırılmış durumdadır. Fakat kartın NAND, Ethernet ve VGA gibi bileşenlere sahip olması, bu tür sistemleri tanıma sürecinde bizim için kullanışlı bir çalışma ortamı oluşturacaktır.

Bu bölümde, cihaz için önyükleyici ve işletim sistemi çekirdeğinin nasıl derlendiğinden, sonrasında cihazı kendi bilgisayarımız üzerinden (*Host*) nasıl açabileceğimize (*Recovery*) ve temel bazı kullanım senaryolarına bakacağız.

İlk olarak, önyükleyici ve çekirdeği nasıl edinip derleyebileceğimize bakalım.

> **Not:** Derleme sürecinde *CodeSourcery* geliştirme araçlarını kullanacağız. Bu aşamada çapraz derleyicinizin yol ifadesinin *PATH* çevre değişkeninde tanımlı olduğundan emin olunuz.
>
> Ayrıca başka bir derleyici kullanıyorsanız, önyükleyici ve çekirdek derleme aşamalarında, derleyicinize ait öneki (*prefix*) kullanmalısınız. *CodeSourcery* için derleyici öneki *arm-none-linux-gnueabi-* şeklindedir.

Önyükleyici (*Bootloader*)

Hawkboard, *U-Boot* önyükleyicisini kullanmaktadır.

```
wget -c http://hawkboard.googlecode.com/files/u-boot-omapl1_v1.tar.bz2
tar xf u-boot-omapl1_v1.tar.bz2
cd u-boot-omapl1
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- distclean
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- omapl_hawkboard_config
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

Bu aşamada ana dizinde *u-boot* dosyası oluşmuş olmalıdır.

Çekirdek (*Kernel*)

```
wget -c http://hawkboard.googlecode.com/files/linux-omapl1_ver1.tar.bz2
tar xf linux-omapl1_ver1.tar.bz2
cd linux-omapl1
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- distclean
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- omapl138_hawkboard_defconfig
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- uImage
```

Bu aşamada ana dizinden itibaren *arch/arm/boot* dizininde, *U-Boot* tarafından yüklenebilecek, çekirdek imajı *uImage* oluşmuş olmalıdır.

Şimdi kartı sıfırdan nasıl açabileceğimize bakalım.

Kartın Kurtarılması (*Board Recovery*)

Kart üzerinde, açılış sürecine ilişkin alt seviye bir yazılım olmaması, bu yazılımın bozulmuş olması durumununda veya bu süreç *bypass* edilmek istendiğinde, kartın dışarıdan açılması tek veya alternatif bir seçenek olarak karşımıza çıkmaktadır.

Kart kurtarma senaryolarında genel olarak cihaza SD kart, USB veya seri port üzerinden ilk olarak önyükleyicinin atılarak çalıştırılması hedeflenmektedir. Cihaz üzerinde önyükleyici çalıştırıldıktan sonra seri terminal üzerinden cihazla alt seviye bir iletişim kurulabilir ve sonraki adımlara buradan devam edilebilir.

Hawkboard seri port üzerinden (*UART Recovery*) açılabilmektedir. Öncesinde, daha önce derlediğimiz önyükleyici dosyasını (*u-boot*) seri port için uygun şekilde imzalamalıyız. Bu imzalama ve önyükleyiciyi cihaza atma işlemleri için TI tarafından hazırlanan *AIS Generator / UART Host Tool* araçlarını kullanacağız. Gerekli araçları sisteminize kurmak için ilk olarak aşağıdaki *zip* dosyasını indirebilirsiniz.

> **Not:** TI tarafından sağlanan bu araçlar, maalesef Linux karşılıkları olmayan, .NET Framework bağımlılığı olan pencereli uygulamalardır. Linux altında bu programları kullanabilmek için sisteminizde *mono*, *mono-vbnc* ve *wine* paketleri bulunmalıdır.

```
wget -c http://www-s.ti.com/sc/techlit/sprab41.zip
```

zip dosyasından *AISgen_d800k008_Install_v1.13.exe* kurulum dosyası çıkacaktır. Kurulum sonrasında, sırasıyla imzalama ve cihaza önyükleyici atmak için *AISgen_d800k008.exe* ve *UartHost.exe* uygulamalarını kullanacağız.

İlk olarak, imzalama işlemini nasıl yaptığımıza bakalım.

> **Not:** İşlemci yongasındaki ROM önyükleyici dışarıdan okuyacağı bir sonraki önyükleyiciyi, TI tarafından geliştirilen, Application Image Script (AIS) formatında beklemektedir. Formatın detaylarına TI dokümanlarında mevcuttur.

Seri Port Hedefli İmzalama

AISgen imzalama uygulamasında *General* ve *DDR* sekmelerine aşağıdaki değerleri vermeliyiz. *DDR* sekmesinin görünür olması için *General* sekmesinde, *Configure DDR* seçeneği işaretlenmelidir.

General:

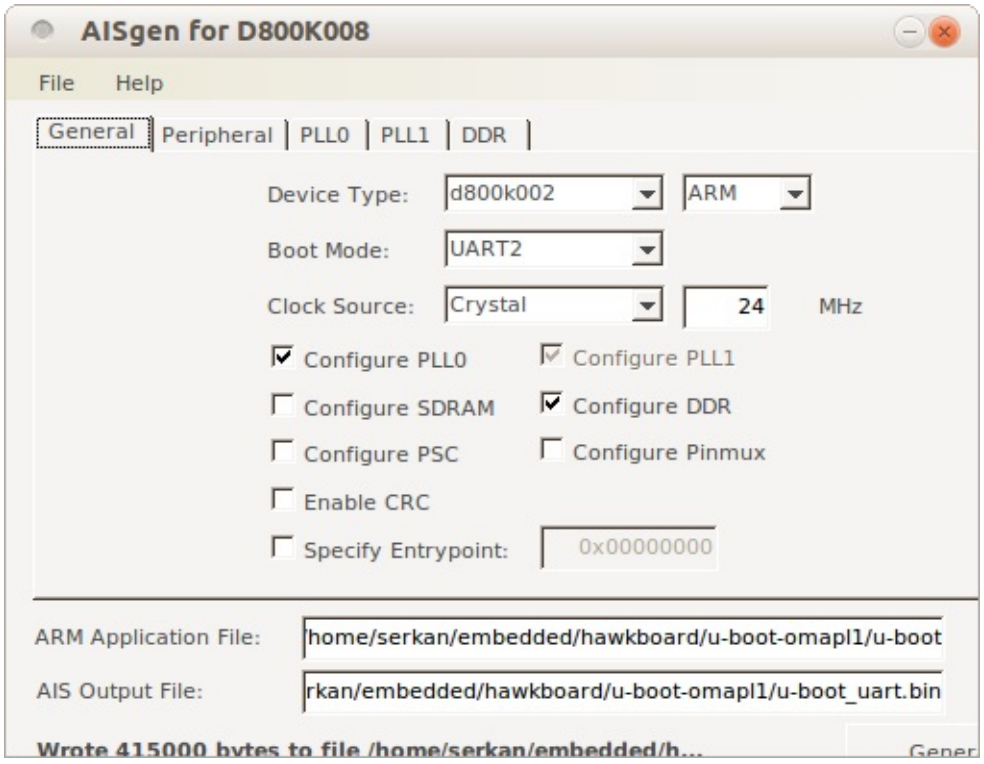
Özellik	Değer
Device Type	d800k002 - ARM
Boot Mode	UART2

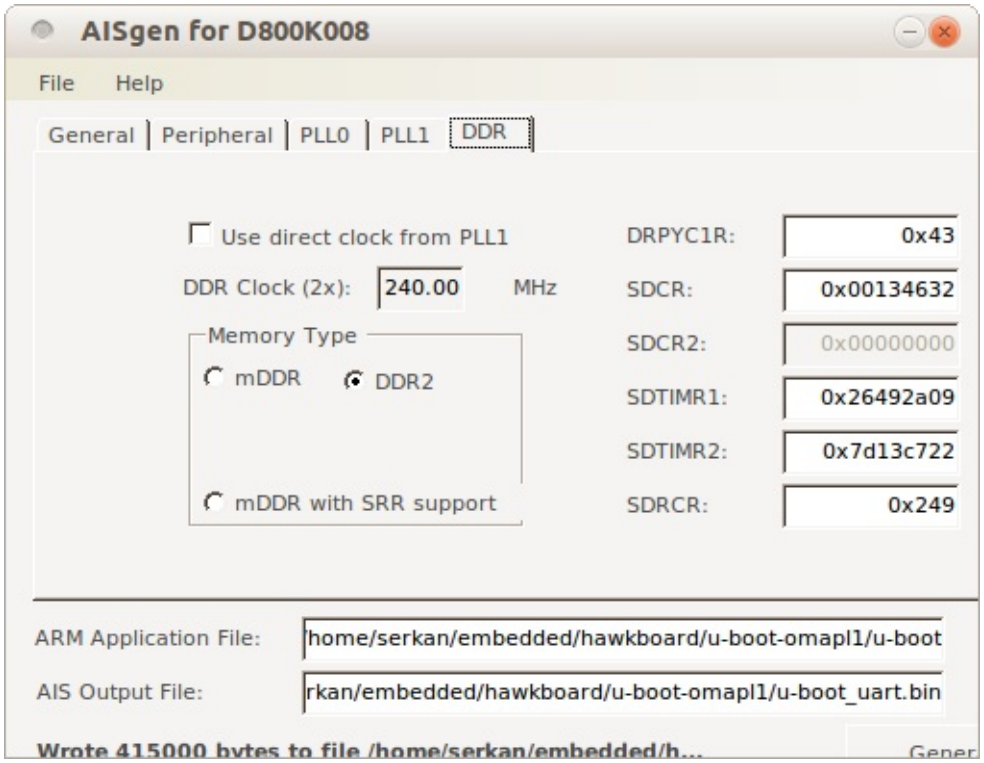
DDR:

Özellik	Değer
Memory Type	DDR2
DRPYC1R	0x43
SDCR	0x00134632
SDCR2	-
SDTIMR1	0x26492a09
SDTIMR2	0x7d13c722
SDRCR	0x249

> **Not:** Bu aşamada yukarıdaki bu değerlerin ne anlama geldiğini bilmek zorunda değiliz.

Linux altında aldığımız örnek ekran görüntüleri aşağıdaki gibidir.





Sonrasında, *ARM Application File* bölümüne daha önce derlediğiniz *u-boot* dosyasını, *AIS Output File* bölümünü de oluşturulacak olan dosya ismini vererek imzalama işlemini gerçekleştirebilirsiniz. Biz imzalı dosyamıza *u-boot_uart.bin* ismini verdik.

Şimdi imzalanmış önyükleyici dosyasını Hawkboard'a nasıl atacağımıza bakalım.

Seri Port Üzerinden Önyükleyicinin Atılması

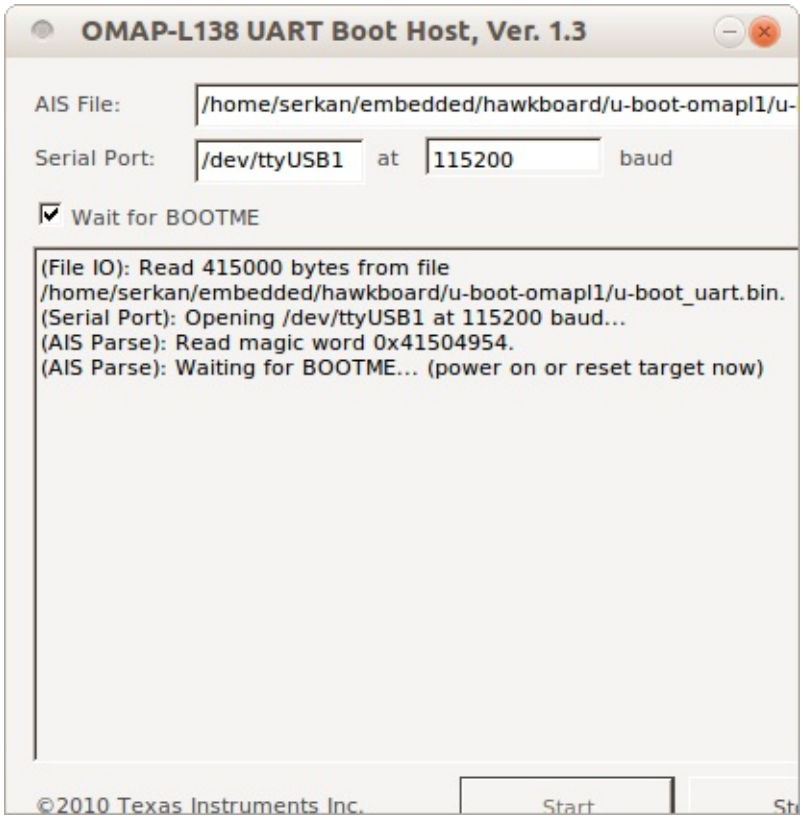
Bu amaçla *UartHost.exe* uygulamasını kullanacağız. Fakat öncesinde cihaz üzerindeki açılış sürecini kontrol eden anahtarları (*Boot Switches*) uygun şekilde ayarlamalıyız. Üretim aşamasında işlemci içindeki dahili bir hafızaya kodlanan önyükleyici (*ROM Bootloader*) bu anahtarların durumuna göre sonraki önyükleyiciyi NAND veya seri port üzerinden okumaya çalışmaktadır. Cihazı seri port üzerinden açabilmek için boot anahtarları aşağıdaki gibi ayarlanmalıdır.

> **Not:** Boot anahtarları, kart üzerinde DIP switch şeklinde bulunmaktadır. Reset butonunun yakınında bulunan bu anahtarlar *SW1* olarak isimlendirilmiştir.

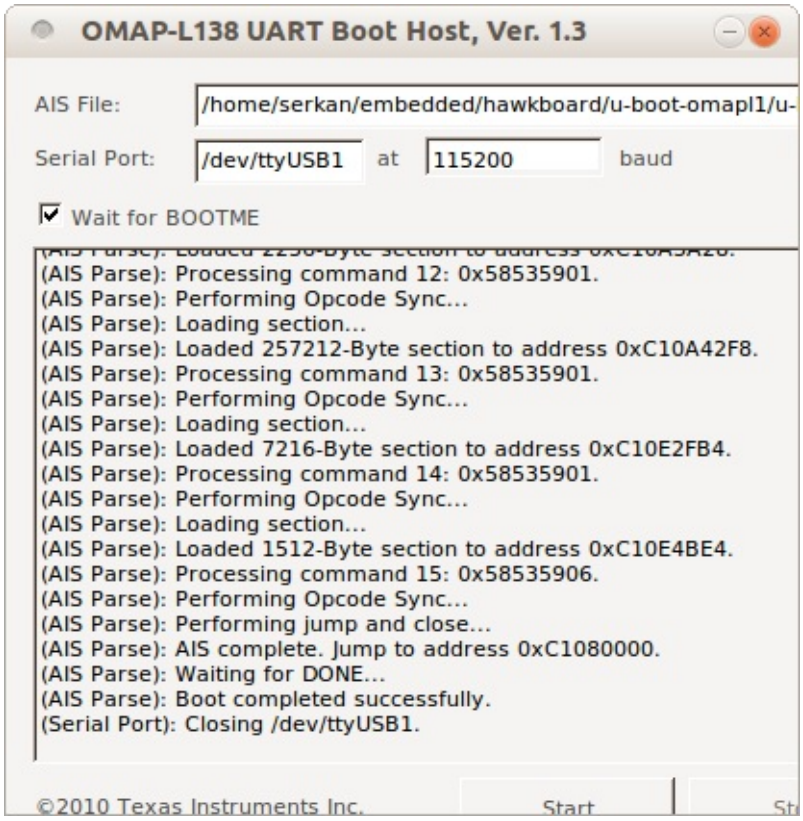
1	2	3	4
OFF	ON	OFF	ON

Bu durumda, cihaza enerji verildiğinde veya resetlendiğinde, seri terminale **BOOTME** mesajını gönderecektir. Cihaza önyükleyiciyi atmadan önce bu mesajın geldiğinden emin olunuz ve seri terminal bağlantınızı sonlandırınız.

Şimdi *UartHost.exe* uygulamasını kullanabiliriz. Uygulamaya daha önce imzaladığımız *u-boot* dosyasını ve seri portu göstermeliyiz, ayrıca *Wait for BOOTME* seçeneğinin seçili olduğundan emin olunuz. Bu durumda uygulamada *Start* butonuna bastığımızda Linux altında aldığımız örnek ekran görüntüsü aşağıdaki gibidir.



Bu aşamada uygulama cihaza enerji vermemizi veya resetlememizi beklemektedir. Cihaz yeniden başlatıldığında, imzalanmış *u-boot* cihaza atılacaktır. Örnek ekran görüntüsü aşağıdaki gibidir.



Sonrasında seri terminal üzerinden cihaza bağlanıp *enter* tuşuna bastığınızda *u-boot* komut satırını görmelisiniz. *printenv* ile önyükleyicinin kullandığı çevre değişkenlerini görebilirsiniz.

```
hawkboard.org > printenv
bootargs=mem=128M console=ttyS2,115200n8 root=/dev/ram0 rw initrd=0xc1180000,4M ip=dhc
p
bootcmd=
bootdelay=3
baudrate=115200
bootfile="uImage"
stdin=serial
stdout=serial
stderr=serial
ethaddr=0a:c1:a8:12:fa:c0
ver=U-Boot 2009.01 (Oca 21 2015 - 14:11:03)

Environment size: 254/131068 bytes
```

Bu aşamadan sonra sistemi açmak için bir çok seçeneğe sahibiz. Sırayla bunlardan bazılarını kısaca inceleyim.

Cihazın TFTP ve NFS Üzerinden Açılması

Bu yöntemi kullanarak, ağ üzerindeki başka bir sistemden veya geliştirme yaptığınız bilgisayardan, çekirdeği cihaza çekmek ve uzaktaki bir dosya sistemini *mount* etmek mümkündür. Öncesinde geliştirme yaptığınız bilgisayarınızda *TFTP* ve *NFS* sunucuları kurulu olmalıdır.

Uzaktaki bir makinadan çekirdeği cihaz üzerine aşağıdaki gibi çekebilirsiniz.

```
setenv autoload no
dhcp
setenv serverip <TFTP sunucu adresi>
tftp c0700000 uImage
```

Açılış sırasında *root* dosya sistemi olarak *NFS*'i kullanmak için ise *bootargs* değişkenine aşağıdaki formda değerler geçirilmelidir.

```
setenv nfsroot ${serverip}:<Uzak makinadaki NFS dizini>

setenv bootargs "root=/dev/nfs nfsroot=${nfsroot} ip=${ipaddr}:${serverip}:${gatewayip
}:${netmask}:${hostname}:eth0"
```

dhcp özelliğinin olmaması durumunda, cihaz IP, gateway ve netmask değerlerini aşağıdaki gibi verebilirsiniz.

```
setenv ipaddr <Cihaz IP adresi>
setenv gatewayip <Gateway adresi>
setenv netmask <Netmask>
```

Sonrasında daha önce çekirdeği çektiğiniz adresteki kodu çalıştırarak cihazı açabilirsiniz.

```
bootm <Çekirdeğin belleğe çekildiği adres>
```

Kendi sistemimiz için örnek bir kullanım aşağıdaki gibidir.

```
setenv autoload no
dhcp
setenv serverip 172.16.2.136
tftp c0700000 uImage
setenv nfsroot ${serverip}:/nfsroot
setenv bootargs "console=ttyS2,115200 init=/bin/sh root=/dev/nfs nfsroot=${nfsroot} ip
=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:eth0"
bootm c0700000
```

Şimdi, bir diğer alternatif olarak, cihazı NAND üzerinden nasıl açabileceğimize bakalım.

Cihazın NAND Üzerinden Açılması

Cihazı NAND üzerinden açabilmek için ilk olarak gerekli dosyaları bir şekilde NAND üzerine yazmalıyız. Bu işlemi, dosya sistemini NFS üzerinden mount edip, Linux üzerinden yapabildiğimiz gibi, U-Boot üzerinden de yapabiliriz. Fakat öncesinde, daha önce seri port için yaptığımıza benzer şekilde, NAND üzerine yazacağımız u-boot dosyasını imzalamalıyız.

Şimdi sırasıyla, imzalama işlemine ve gerekli dosyaları NAND üzerine nasıl yazabileceğimize bakalım.

NAND Hedefli İmzalama

İmzalama işlemi için yine *AISgen_d800k008.exe* uygulamasını kullanacağız, seri port için imzalama yaparken kullandığımız aynı değerleri kullanacağız yalnız *Boot Mode* tipini *NAND Flash* olarak seçmeliyiz.

General:

Özellik	Değer
Device Type	d800k002 - ARM
Boot Mode	NAND Flash

NAND Bölümlendirmesi

Çekirdek, NAND üzerinde sanki bir bölümlendirme (*Partition*) yapısı varmış gibi işlem yapabilir. Bu bölümlendirme, Linux kaynak kodu içerisinden sabit olarak ayarlanabileceği gibi çekirdeğe önyükleyici tarafından geçirilen argümanlar vasıtasıyla dinamik olarak da yapılabilir.

Sabit bölümlendirme bilgileri *arch/arm/mach-davinci/board-omapl138-hawk.c* dosyasında *mtd_partition* türündeki *omapl138_hawk_nandflash_partition* isimli bir dizi ile temsil edilmektedir.

Dinamik bölümlendirme için ise çekirdek kodu *command line partition table parsing* seçeneği seçilerek derlenmeli ve çekirdeğe *mtdparts* komut satırı seçeneği geçirilmelidir. Örnek olarak aşağıdaki gibi bir bölümlendirme oluşturalım.

Bölüm	Uzunluk
U-Boot_Env	128K
U-Boot	512K
Kernel	4M
Recovery	40M
FileSystem	Kalan Boş Alan

Bu bölümlendirmeyi oluşturabilmek için çekirdeğe, diğer komut satırı parametreleriyle beraber, aşağıdaki *mtdparts* değerini geçirmeliyiz.

```
mtdparts=davinci_nand.1:128K(U-Boot_Env),512K(U-Boot),4M(Kernel),40M(Recovery),-(FileS
ystem)
```

> **Not:** Yukarıdaki ifadede *davinci_nand*, mtd id değerini, noktadan sonra eklenen 1 değeri ise aygıt numarasını göstermektedir. mtd id değerine, Linux üzerinden, aşağıdaki gibi ulaşabilirsiniz.
>
> *ls /sys/bus/platform/drivers | grep nand*

Linux üzerinden NAND bölümlendirmesini *cat /proc/mtd* şeklinde görebilirsiniz. Örneğimiz için çıktı aşağıdaki gibidir.

```
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00020000 00020000 "U-Boot_Env"
mtd1: 00080000 00020000 "U-Boot"
mtd2: 00400000 00020000 "Kernel"
mtd3: 02800000 00020000 "Recovery"
mtd4: 05360000 00020000 "FileSystem"
```

Her bir bölüm üzerinde, bir blok aygıt (*block device*) olarak işlem yapılabilir. İlgili dosya düğümlerini aşağıdaki gibi listeleyebilirsiniz.

```
# ls /dev/mtdblock*
/dev/mtdblock0  /dev/mtdblock1  /dev/mtdblock2  /dev/mtdblock3  /dev/mtdblock4
```

Şimdi sırasıyla önyükleyici, çekirdek imajı ve dosya sistemini nasıl yazabileceğimize bakalım. Ayrıca, *FileSystem* olarak isimlendirdiğimiz son bölümden önce *Recover* isimli başka bir bölüm daha oluşturduğumuza dikkat ediniz. Bu bölüm üzerinde, güvenlik amaçlı, yalnız okunabilen bir dosya sistemi oluşturacağız.

Gerekli Dosyaların Linux Üzerinden Yazılması

İlk olarak, NAND üzerine yazacağımız önyükleyici, çekirdek ve dosya sistemi imajlarını NFS üzerinden açtığımız cihazımızın dosya sistemine kopyalamalıyız. Gerekli dosyaları */opt* altına attığımızı kabul edelim.

```
/opt # ls
rootfs.jffs2     uImage
rootfs.sqsh      u-boot_nand.bin
```

Dosya	Türü
u-boot_nand.bin	NAND için imzalanmış önyükleyici
uImage	Çekirdek imajı
rootfs.sqsh	squashfs dosya sistemi imajı
rootfs.jffs2	jffs2 dosya sistemi imajı

Gerekli dosyaları *dd* aracı ile aşağıdaki gibi sırasıyla NAND üzerine yazabiliriz.

> **Uyarı:** Bu işlemler öncesinde, U-Boot komut satırından *nand erase* ile tüm NAND'i silmeyi unutmayınız. Diğer bir alternatif *mtd-utils* araçlarını cihaz hedefli derleyerek aynı işlemi Linux üzerinden yapmak olabilir.

```
dd if=u-boot_nand.bin of=/dev/mtdblock0 bs=128k
dd if=uImage of=/dev/mtdblock2 bs=128k
dd if=rootfs.sqsh of=/dev/mtdblock3 bs=128k
dd if=rootfs.jffs2  of=/dev/mtdblock4 bs=128k
```

Şimdi aynı işlemi u-boot üzerinden nasıl yapabileceğimize bakalım.

Gerekli Dosyaların U-Boot Üzerinden Yazılması

```
nand erase 0x20000 0x80000
tftpboot 0xc0700000 u-boot_nand.bin
nand write.e 0xc0700000 0x20000 0x80000

nand erase 0xA0000 0x400000
tftpboot 0xc0700000 uImage
nand write.e 0xc0700000 0xA0000 0x400000

nand erase 0x2CA0000 0x1000000
tftpboot 0xc0700000 rootfs.sqsh
nand write.e 0xc0700000 0x2CA0000 0x1000000

nand erase 0x4A0000 0x1000000
tftpboot 0xc0700000 rootfs.jffs2
nand write.e 0xc0700000 0x4A0000 0x1000000
```

nand write.e komutunun genel şekli aşağıdaki gibidir.

```
nand write.e <memory address> <offset> <size>
```

Çekirdeğe geçireceğininiz NAND bölümlendirmesini değiştirmeniz durumunda, kullandığımız ofset değerleri de değişecektir. Ayrıca yukarıdaki örnek kullanımda dosya sistemlerinin 16M sınırında olduğu kabul edilmiştir, boyut olarak 0x1000000 değerinin kullanıldığına dikkat ediniz.

Root dosya sisteminin uygun parametlerle *jffs2* imajının oluşturuluması ve sonrasında, U-Boot veya Linux üzerinden, NAND üzerine yazıldıktan sonra tekrar okunmasında bazı problemler ortaya çıkabilmektedir. Bu yüzden önyükleyici ve çekirdeği u-boot üzeriden yazmanızı, dosya sistemini ise Linux üzerinden ilgili bölümü mount edip dosya sistemini kopyalamanızı öneririz.

Önerilen Yöntem

U-Boot:

```
nand erase

tftpboot 0xc0700000 u-boot_nand.bin
nand write.e 0xc0700000 0x20000 0x80000

tftpboot 0xc0700000 uImage
nand write.e 0xc0700000 0xA0000 0x400000

setenv nfsroot ${serverip}:/nfsroot

setenv bootargs "console=ttyS2,115200 init=/bin/sh root=/dev/nfs nfsroot=${nfsroot} ip
=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:eth0"

bootm c0700000
```

Linux:

```
mount -t jffs2 /dev/mtdblock4 nand/
tar xf rootfs.tgz -C nand/
umount nand
```

Bu aşamada ROM yükleyicinin, bir sonraki açılışta, u-boot yükleyicisini NAND üzerinden araması için boot anahtarları aşağıdaki gibi ayarlanmalıyız.

1	2	3	4
ON	OFF	OFF	OFF

Gerekli dosyaları NAND üzerine yazdıktan sonra cihazın kendiliğinden NAND üzerinden açılabilmesi için önyükleyici üzerinde bazı değişiklikler yapmalıyız. Cihazı resetleyerek tekrar u-boot komut satırına düşelim.

Çekirdeğe, *root* dosya sisteminin yerini ve tipini geçirmeliyiz. Aşağıdaki kullanımda *root=/dev/mtdblock4* dosya sisteminin aranacağı NAND bölümünü, *rootfstype=jffs2* ise dosya sisteminin tipini göstermektedir. Sonrasında NAND üzerindeki çekirdek kodu, güvenli bir adrese çekilip başlatılmalıdır. *nand read* ve *bootm* komutları bu işleri yapmaktadır. *bootcmd* ile gösterilen komutlar, açılışta önyükleyici tarafından otomatik olarak çalıştırılarak, bu komutların icra edilmesi sağlanır.

```
setenv bootargs "console=ttyS2,115200 init=/bin/sh root=/dev/mtdblock4 rootwait rootfs
type=jffs2 mtdparts=davinci_nand.1:128K(U-Boot_Env),512K(U-Boot),4M(Kernel),40M(Recove
ry),-(FileSystem)"

setenv bootcmd 'nand read c0700000 A0000 200000; bootm c0700000'

saveenv
```

Bu aşamadan sonra cihaz resetlendiğinde, dışarıdan herhangi bir müdahale olmaksızın, dahili NAND hafızası üzerinden açılacaktır.

Olimex A20

Olimex firması tarafından geliştirilen board, Dual-Core Allwinner A20 Cortex-A7 işlemcisiyle gelmektedir. Sadece yazılım değil, donanım da açık bir platform olarak geliştirilmekte ve tüm tasarım dosyaları paylaşılmaktadır.

Kartın tam adı **A20-OLinuXino-MICRO** olarak geçmektedir fakat dokümanda kısaca **Olimex A20** adını kullanacağız.

Temel olarak aşağıdaki bileşenlere sahiptir.

İşlemci	**Allwinner A20 Cortex-A7**
Çekirdek Sayısı	2
DSP	Yok
GPU	Mali 400
RAM	1 GB DDR3
NAND	Yok
MMC/SD	Var
MicroSD	Var
Ethernet	Var
VGA	Var
HDMI	Var
Konsol	Var

Bu bölümde, cihaz için önyükleyici ve işletim sistemi çekirdeğinin nasıl derlendiğinden, sonrasında cihazın mikro SD kart üzerinden nasıl açıldığından bahsedeceğiz.

İlk olarak, önyükleyici ve çekirdeği nasıl edinip derleyebileceğimize bakalım.

> **Not:** Derleme sürecinde **armhf** hedefli GNU C çapraz derleme araçlarını kullanacağız. Paket yöneticinizi kullanarak gerekli araçları edinebilirsiniz, Ubuntu için *gcc-arm-linux-gnueabihf* paketini kurabilirsiniz.
>
> Başka bir derleyici kullanıyorsanız, önyükleyici ve çekirdek derleme aşamalarında, derleyicinize ait öneki (*prefix*) kullanmalısınız.

Önyükleyici (*Bootloader*)

Olimex A20, *U-Boot* önyükleyicisini kullanmaktadır.

```
git clone https://github.com/linux-sunxi/u-boot-sunxi.git
cd u-boot-sunxi/
make CROSS_COMPILE=arm-linux-gnueabihf- A20-OLinuXino-Micro_config
make CROSS_COMPILE=arm-linux-gnueabihf-
```

Bu aşamada ana dizinde *u-boot.img* ve spl altında *sunxi-spl.bin* dosyaları oluşmuş olmalıdır. *sunxi-spl.bin* dosyası, u-boot önyükleyicisini yüklemekten sorumlu, göreli olarak daha küçük olan önyükleyici olup *SPL (Secondary Program Loader)* olarak isimlendirilmektedir.

Ayar Dosyası (*Script file*)

GPIO atamaları, DDR bellek parametreleri ve video çözünürlüğü gibi önemli bazı ayar parametreler, çekirdek kodu yerine, bir veri dosyasında tutulabilmektedir. Bu sayede çekirdek kodu her defasında yeniden derlenmeksizin, bu parametreler üzerinde sonradan değişiklik yapılabilmektedir.

Gerekli ayar dosyası ve araçları aşağıdaki gibi sisteminize indirebilirsiniz.

```
git clone git://github.com/linux-sunxi/sunxi-tools.git
git clone git://github.com/linux-sunxi/sunxi-boards.git
```

İlk olarak *sunxi-tools* dizininde *fex2bin* uygulamasını derleyecek, sonrasında bu uygulamayı kullanarak, *sunxi-boards* dizinindeki *a20-olinuxino_micro.fex* isimli ayar dosyasını, çekirdeğin okuyabileceği, uygun formata dönüştüreceğiz.

```
cd sunxi-tools
make fex2bin
cd ../sunxi-boards/
../sunxi-tools/fex2bin sys_config/a20/a20-olinuxino_micro.fex script.bin
```

Bu aşamada hedeflediğimiz *script.bin* dosyasını elde etmiş olduk.

Çekirdek (*Kernel*)

Çekirdek kodunu aşağıdaki gibi indirip derleyebilirsiniz.

```
git clone -b sunxi-3.4 https://github.com/linux-sunxi/linux-sunxi.git
cd linux-sunxi/
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- sun7i_defconfig
make -j2 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- uImage modules
```

Bu aşamada u-boot tarafından yüklenebilecek çekirdek imajı *uImage* ve gerekli modüller oluşmuş olmalıdır. Daha sonra dosya sistemine taşıyacağımız modülleri, aşağıda gösterildiği gibi bir dizin altına toplayabiliriz. Modülleri taşıyacağımız dizini, aşağıdaki örnek için, *modules* olarak isimlendirdik.

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- INSTALL_MOD_PATH=modules modules_inst
all
```

Bu işlem sonrasında, ana dizinden itibaren, çekirdek imajı ve gerekli modüller aşağıdaki gibi bir dizin yolunda bulunacaktır. Çekirdek versiyonuna göre *xyz* değerleri değişmektedir.

```
arch/arm/boot/uImage
modules/lib/modules/3.4.x-y-z
```

Şimdi cihazı mikro SD üzerinden nasıl açabileceğimize bakalım.

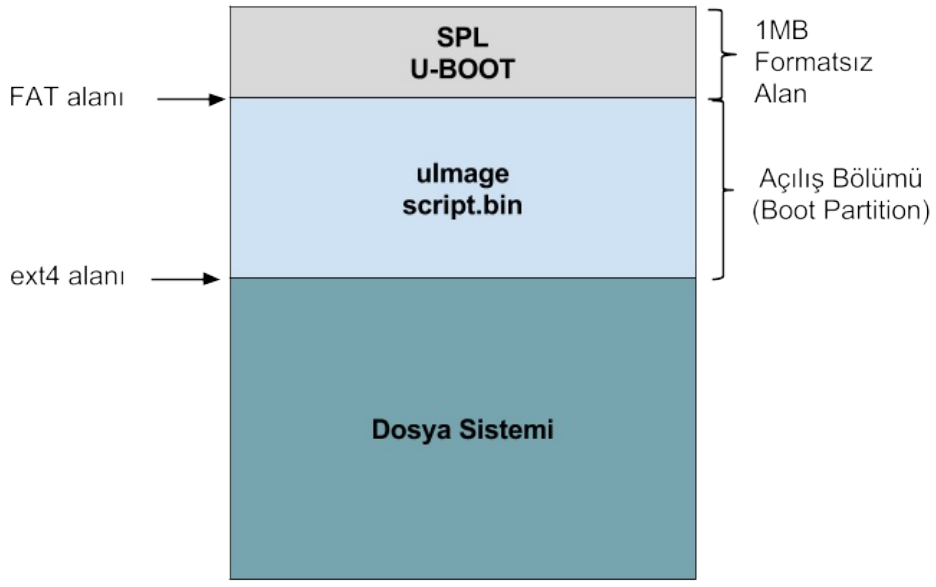
Cihazın Mikro SD Üzerinden Açılması

Mikro SD kartın yerleşimi genel olarak aşağıdaki gibi olacaktır.

Başlangıç	Boyut	Kullanım
0	8KB	Bölümlendirme tablosu
8	24KB	SPL önyükleyicisi
32	512KB	u-boot
544	128KB	Çevre değişkenleri
672	352KB	Saklı alan
1024	-	Dosya sistemi için boş alan

> **Not:** Kart üzerinde hem MMC/SD hem de mikro SD bulunmaktadır. Cihaz yalnız mikro SD üzerinden açılabilmektedir, açılış sürecini anlattığımız aşağıdaki bölümlerde çoğnlukla mikro SD yerine yalnız SD ifadesini kullandık.

SD kart üzerinde, ilk 1MB alandan sonra sırasıyla FAT32 ve ext4 olarak formatlayacağımız iki adet bölüm oluşturacağız. FAT32 bölümüne çekirdek imajı ve ihtiyaç duyduğu ayar dosyasını, ext4 bölümüne ise dosya sistemini kopyalayacağız. İlk 1MB'lık alana ise önyükleyicileri yazacağız. SD kartın hedeflediğimiz son durumunu görsel olarak aşağıdaki gibi gösterebiliriz.



Şimdi SD kart üzerine bu işlemleri nasıl yapabileceğimize bakalım.

Sonraki işlemleri kolaylaştırmak için SD karta ilişkin dosya düğümünü ve bölümlendirmeye ilişkin isim bilgisini sırasıyla *CARD* ve *PARTITION* isimli çevre değişkenlerinde tutalım. SD karta ilişkin dosya düğümü dahili bir SD okuyucu ya da USB bağlantılı bir kart okuyucu kullanılmasına göre değişecektir. Dahili bir SD okuyucu için kart ve bölümlendirme bilgisi aşağıdaki gibi tanımlanabilir.

```
export CARD=/dev/mmcblk0
export PARTITION=p
```

SD kartın ilk 1MB'lık alanını temizledikten sonra *SPL* ve *u-boot* önyükleyicilerini aşağıdaki gibi yazabiliriz.

```
dd if=/dev/zero of=${CARD} bs=1M count=1
dd if=sunxi-spl.bin of=${CARD} bs=1024 seek=8
dd if=u-boot.img of=${CARD} bs=1024 seek=40
```

Şimdi SD kart üzerinde, sonrasında FAT32 ve ext4 olarak biçimlendireceğimiz, iki adet bölüm oluşturalım. Bunun için *sfdisk* aracını kullanan aşağıdaki betiği kullanabilirsiniz.

```
#!/bin/sh
CARD=/dev/mmcblk0
sfdisk -R ${CARD}
cat << EOT | sfdisk --in-order -L -uM ${card}
1,16,c
,,L
EOT
```

Yeni oluşan bölümleri sırasıyla FAT32 ve ext4 olarak biçimlendirelim.

```
mkfs.vfat ${CARD}${PARTITION}1
mkfs.ext4 ${CARD}${PARTITION}2
```

Bu aşamadan sonra açılış bölümü olarak kullanacağımız FAT32 bölümüne, daha önce derlediğimiz, *uImage* ve *script.bin* dosyalarını, ext4 bölümüne ise çekirdek modüllerini içeren dosya sistemini yazmalıyız.

Bu işlemlerden sonra cihazı hazırladığımız SD kart ile açarak u-boot komut satırına düşebiliriz. Açılış sürecinde sırasıyla ROM Bootloader, SPL ve u-boot çalışacaktır. u-boot komut satırına düştükten sonra, çekirdeğe geçirilecek komut satırı argümanlarını oluşturup sonrasında çekirdek kodunu güvenli bir adrese çekip çalıştırabiliriz.

```
setenv bootargs console=ttyS0,115200 init=/bin/sh root=/dev/mmcblk0p2 rootwait panic=1
0
fatload mmc 0 0x48000000 uImage
bootm 0x48000000
```

TI DM6446 EVM

1 Geliştirme araçlarının hazırlanması:

İlk olarak geliştirme araçları indirilip açıldıktan sonra .../bin dizini altındaki derleme araçlarının arm-linux önekine (prefix) sahip sembolik linkleri oluşturulup (görece daha uzun olan arm-none-linux-gnueabi öneki ile uğraşmamak için), dizin yolu PATH değişkenine eklenmiş ve bir çevresel değişkende saklanmıştır.

```
wget -c http://www.codesourcery.com/sgpp/lite/arm/portal/package6488/public/arm-none-l
inux-gnueabi/arm-2010q1-202-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2
arm-2010q1-202-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2
tar jxvf arm-2010q1-202-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2
cd arm-2010q1/bin/
for i in `ls`; do s=`echo $i | cut -b 24-`; ln -s $i arm-linux-$s; done
export PATH=$(pwd):$PATH
export DEVELOPMENT_TOOLS=$(pwd)
```

2 u-boot önyükleyicinin (bootloader) derlenmesi:

DM644x EVM, ubl(User Boot Loader) ve u-boot olmak üzere iki adet önyükleyiciye ihtiyaç duymaktadır. Göreli olarak daha küçük olan ubl, çekirdeği yükleyecek olan u-boot önyükleyicisini başlatmaktan sorumludur. u-boot'un derlenme aşamaları temel olarak aşağıdaki gibidir. u-boot, arm-linux-gcc'yi ARM için öngörülen derleyici olarak kullandığından, çapraz derleme için çekirdeğin aksine CROSS COMPILE veya ARCH çevresel değişkenlerine ihtiyaç duymamaktadır.

```
git clone git://www.denx.de/git/u-boot.git u-boot
make distclean
make davinci_dvevm_config
make
```

u-boot kaynak kodu içerisinde include/configs/davinci_dvevm.h yolu izlenerek önyükleyicinin konfigür ayarları yapılabilir. u-boot öngörülen olarak NOR flash için derlendiğinden bu haliyle NAND üzerinde çalışmamaktadır, ayrıca önyükleyicinin autoboot özelliğinin çalışması için bir bootdelay değeri girilmelidir. Bunun için aşağıdaki değişiklikler yapılmalıdır.

```
-#define CONFIG_SYS_USE_NOR
+#define CONFIG_SYS_USE_NAND

-#undef CONFIG_BOOTDELAY
#define CONFIG_BOOTDELAY    5
```

Ayrıca bootcmd, bootargs gibi çevresel değişkenlerin değerleri hard coded olarak buradan ayarlanabilir.

u-boot temel olarak yukarıdaki gibi derlenmesine karşın birkaç problemle karşılaşılabilir. u-boot derleme işlemi konak makinanın doğal C derleyici ile başlayan bir süreçtir ve konak makinaya bağımlılığı vardır. Konak bilgisayardaki /usr/local/include altındaki bazı başlık dosyalarının varlığı derleme işleminde sorunlara yol açabilmektedir(u-boot, sistemde var olan başlık dosyaları ile ilgili konfigürasyon sonucunda yanlış bir varsayımda bulunuyor olabilir.). Konfigür işleminden önce konak bilgisayarda geçici olarak /usr/local/include dizinin adını değiştirmek sorunu çözebilir.

```
mv /usr/local/include /usr/local/_include
```

3 Çekirdeğin derlenmesi:

En son DaVinci linux çekirdeği asağıdaki yol izlenerek derlenmiştir.

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/khilman/linux-davinci.git linu
x-davinci-2.6
cd linux-davinci-2.6
make ARCH=arm davinci_all_defconfig
make ARCH=arm menuconfig [1]
make ARCH=arm CROSS_COMPILE=arm-linux- uImage
```

4 Dosya sisteminin oluşturulması:

Dosya sisteminin oluşturulması için temel olarak busybox, dropbear araçları derlenmiş, dosya sisteminde bulunması gereken asgari dizin, açılış betikleri, aygıt düğümleri oluşturulmuş ve geliştirme araçlarının kullandığı standart C kütüphanesi dosya sistemine eklenmiştir.

busybox aşağıdaki gibi derlenebilir, ROOTFS çevresel değişkenine başka bir değer verilebilir.

```
export ROOTFS=/tmp/dosyasistemi
mkdir -p ${ROOTFS}
wget -c http://www.busybox.net/downloads/busybox-1.17.1.tar.bz2
tar jxvf busybox-1.17.1.tar.bz2
cd busybox-1.17.1/
make  defconfig
#make  menuconfig [ilave ayarlar yapılması durumda...]
make  CROSS_COMPILE=arm-linux-
make CROSS_COMPILE=arm-linux- CONFIG_PREFIX=${ROOTFS} install
```

Geliştirme araçlarının içindeki standart C kütüpnanesi dosya sisteminde / altına kopyalanmalıdır.

```
cp -a ${DEVELOPMENT_TOOLS}/../arm-none-linux-gnueabi/libc/lib ${ROOTFS}
```

Açılış ve sonrasında ihtiyaç duyulacak olan temel dizinler aşağıdaki gibi oluşturulabilir.

```
mkdir -p ${ROOTFS}/etc/init.d
mkdir -p ${ROOTFS}/dev
mkdir -p ${ROOTFS}/tmp
mkdir -p ${ROOTFS}/proc
mkdir -p ${ROOTFS}/home/root
```

busybox içinden çıkan örnek inittab üzerinde değişiklik yapılarak dosya sisteminde kullanılabilir.

```
cp examples/inittab ${ROOTFS}/etc
```

inittab dosyasına aşağıdaki satır eklenmeli veya açıklama şeklinde olan benzer satır aynı forma dönüştürülmelidir.

```
::respawn:/sbin/getty -L ttyS0 115200 vt100
```

Cihaza şu an yalnız uzaktan bağlanılacağı için(seri port veya ağ üzerinden) "/sbin/getty invocations for selected ttys" altındaki tty4 ve tty5 ile başlayan satırlar açıklama haline getirilebilir aksi halde /dev altında gerekli tty aygıt düğümleri oluşturulmalıdır.

/etc/init.d altında içeriği aşağıdaki gibi olan rcS betiği oluşturulmalı ve gerekli izinleri verilmelidir.

```
#!/bin/sh
mount -t proc proc /proc
mount -t devpts devpts /dev/pts
```

rcS için aşağıdaki gibi gerekli izinleri verilebilir.

```
chmod 755 etc/init.d/rcS
```

/dev altındaki gerekli aygıt düğümleri ve pts dizini aşağıdaki gibi oluşturulabilir veya düğümler konak bilgisayardan cp komutuna -a anahtarı verilerek kopyalanabilir.

```
mknod -m 666 ${ROOTFS}/dev/tty c 5 0
mknod -m 666 ${ROOTFS}/dev/console c 5 1
mknod -m 666 ${ROOTFS}/dev/tty1 c 4 1
mknod -m 666 ${ROOTFS}/dev/tty2 c 4 2
mknod -m 666 ${ROOTFS}/dev/tty3 c 4 3
mknod -m 666 ${ROOTFS}/dev/tty4 c 4 4
mknod -m 666 ${ROOTFS}/dev/random c 1 8
mknod -m 666 ${ROOTFS}/dev/urandom c 1 9
mknod -m 666 ${ROOTFS}/dev/ptmx c 5 2
mknod -m 666 ${ROOTFS}/dev/zero c 1 5
mknod -m 666 ${ROOTFS}/dev/null c 1 3
mkdir -p ${ROOTFS}/dev/pts
```

Cihaz bu noktadan sonra bu dosya sistemi kullanılarak nfs üzerinden açılabilir, cihaza seri port üzerinden login olunabilir. Ağ üzerinden login olmak için küçük bir SSH2 sunucu ve istemci uygulaması olan Dropbear aşağıdaki adımlar izlenerek derlenmiş ve cihaz üzerinde çalıştırılmıştır.

```
wget -c http://matt.ucc.asn.au/dropbear/releases/dropbear-0.52.tar.gz
tar xf dropbear-0.52.tar.gz
cd dropbear-0.52/
./configure --host=arm-linux --disable-zlib
make PROGRAMS="dropbear dbclient dropbearkey scp" MULTI=1 STATIC=1
cp dropbearmulti ${ROOTFS}/usr/bin
```

Dropbear uygulamasının cihaz üzerinde çalıştırılması için cihaz üzerinde aşağıdaki adımlar izlenebilir.

```
cd /usr/bin
ln -s dropbearmulti dropbear
ln -s dropbearmulti dbclient
ln -s dropbearmulti dropbearkey
ln -s dropbearmulti scp
mkdir -p /etc/dropbear
dropbearkey -t rsa -f /etc/dropbear/dropbear_rsa_host_key
```

Cihazda /etc altında kullanıcının id numarası, home dizini ve sisteme login olduğunda çalıştıralacak olan programın belirtildiği passwd dosyası oluşturulmalıdır. root kullanıcısı için passwd dosyasının içeriği aşağıdaki gibi olabilir.

```
root::0:0:root:/home/root:/bin/sh
```

/etc/passwd dosyasında kriptolu olarak tutulacak olan, root şifresi passwd komutu ile oluşturulabilir. devpts mount edildikten ve dropbear uygulaması başlatıldıktan sonra cihaza ssh üzerinden uzaktan bağlanılabilir.

```
mount -t devpts devpts /dev/pts
dropbear
```

dropbear uygulamasının rcS betiği içerisinden sistemin açılışında başlatılması bu işlemi otomatikleştirecektir.

mtd aygıtı ile ilgili düğümler aşağıdaki gibi oluşturulabilir, ardından cihaz nfs üzerinden açılıp dosya sistemi 7. bölümde gösterildiği gibi NAND üzerine yazılabilir.

```
mknod -m 666 ${ROOTFS}/dev/mtd0 c 90 0
mknod -m 666 ${ROOTFS}/dev/mtd1 c 90 2
mknod -m 666 ${ROOTFS}/dev/mtd2 c 90 4
mknod -m 666 ${ROOTFS}/dev/mtd3 c 90 6
mknod -m 666 ${ROOTFS}/dev/mtdblock0 b 31 0
mknod -m 666 ${ROOTFS}/dev/mtdblock1 b 31 1
mknod -m 666 ${ROOTFS}/dev/mtdblock2 b 31 2
mknod -m 666 ${ROOTFS}/dev/mtdblock3 b 31 3
```

5 Cihazın açılma yöntemleri:

5.1 FLASH (NOR) üzerindeki önyükleyici ile açılma durumu:

Sistem FLASH üzerinden açılacaksa CS2 SELECT pinlerinden jumper ile FLASH seçilmeli ve işlemci ayarları ile açılış modunun değiştirildiği, kart üzerinde S3 etiketli, kullanıcı anahtarları aşağıdaki konumlara getirilmelidir.

```
1            ON
2            OFF
3            ON
4            ON
....            (ilgisiz)
```

Anahtarların hangi durumda ON oldukları anahtar kılıfına bakılarak anlaşılabilir.

Sistem bu şekilde açıldığında FLASH'a önceden yazılmış olan ubl ve u-boot sayesinde, minicom ile cihaza bağlanıldığında u-boot komut satırına düşülebilir. Bundan sonra kernel tftp üzerinden çekilip, u-boot çevresel değişkenleri ile harddisk dosya sistemi veya NFS dosya sistemi seçilerek cihaz açılabilir.

5.2 NAND üzerindeki önyükleyici ile açılma durumu:

Sistem NAND üzerinden açılacaksa CS2 SELECT pinlerinden jumper ile NAND seçilmeli ve kullanıcı anahtarları aşağıdaki konumlara getirilmelidir.

```
1            OFF
2            OFF
3            OFF
4            OFF
....            (ilgisiz)
```

Cihazın NAND flashının boş olması durumunda cihaz açılmayacaktır, aynı durum NOR FLASH içinde geçerlidir. Bu örnek için cihazın NAND'ine ubl ve u-boot önyükleyici çiftinin atılması gereklidir. Bunun icin kullanıcı anahtarlarından ilk ikisi ON konumuna alınarak cihaz UART üzerinden boot moduna alınmalıdır. Bu durumda konak bilgisayardan cihaza minicom ile bağlanıldığında ekranda DaVinci yongası içindeki ROM hafızada saklanan, kullanıcının değiştiremeyeci, ROM Boot Loader(RBL) tarafından gönderilen BOOTME mesajı görüntülenecektir. Önyükleyici çifti seri port üzerinden cihaza aşağıdaki gibi yazılabilir.

```
wget -c http://sourceforge.net/projects/dvflashutils/files/DM644x/v2.00/DM644x_FlashAn
dBootUtils_2_00.tar.gz/download
tar zxvf DM644x_FlashAndBootUtils_2_00.tar.gz
cd DM644x_FlashAndBootUtils_2_00/DM644x/GNU
./sfh_DM644x.exe -erase -flashType NAND -targetType DM6446 -p /dev/ttyUSB0
./sfh_DM644x.exe -flash  -flashType NAND -targetType DM6446 -p /dev/ttyUSB0 ubl/ubl_DM
6446_NAND.bin u-boot.bin [3]
```

u-boot olarak, deneme kartının üreticisi Spectrum Digital tarafından sağlanan nand flash paketinden çıkan u-boot kullanıldığında(u-boot-567-nand.bin) temel iki sorunla karşılaşılmıştır[2], kaynak koddan derlenen u-boot ile bu sorunlar yaşanmamıştır. ubl olarak seri programlama paketiyle gelen dosya kullanılmalıdır.

Sonrasında ilk iki kullanıcı anahtarı OFF yapılarak cihaz tftp ve nfs üzerinden açılabilir veya çekirdek ve dosya sistemi imajları kalıcı olarak NAND flash üzerine yazılabilir. Örnek açılış senaryoları kısmında gerekli u-boot çevresel değişkenlerinin değerleri verilmiştir.

6 NAND Flash partisyon yapısı:

Kernel tarafından flashın yorumlanma şekli, yani hangi partisyon yapısında görüleceği Linux kaynak kodu içerisinden sabit olarak ayarlanabileceği gibi çekirdeğe önyükleyici tarafından geçirilen argümanlar vasıtasıyla dinamik olarak da yapılabilmektedir. Sabit partisyon için arch/arm/mach-davinci/board-dm644x-evm.c dosyasinda mtd_partition türündeki diziye yeni elemanlar eklenebilir, var olanlar üzerinde değişiklik yapılabilir. Varsayılan partisyon yapısını belirleyen davinci_evm_nandflash_partition isimli dizi aşağıdaki gibidir.

```
static struct mtd_partition davinci_evm_nandflash_partition[] = {
    /* Bootloader layout depends on whose u-boot is installed, but we
     * can hide all the details.
     *  - block 0 for u-boot environment ... in mainline u-boot
     *  - block 1 for UBL (plus up to four backup copies in blocks 2..5)
     *  - blocks 6...? for u-boot
     *  - blocks 16..23 for u-boot environment ... in TI's u-boot
     */
    {
        .name        = "bootloader",
        .offset        = 0,
        .size        = SZ_256K + SZ_128K,
        .mask_flags    = MTD_WRITEABLE,    /* force read-only */
    },
    /* Kernel */
    {
        .name        = "kernel",
        .offset        = MTDPART_OFS_APPEND,
        .size        = SZ_4M,
        .mask_flags    = 0,
    },
    /* File system (older GIT kernels started this on the 5MB mark) */
    {
        .name        = "filesystem",
        .offset        = MTDPART_OFS_APPEND,
        .size        = MTDPART_SIZ_FULL,
        .mask_flags    = 0,
    }
    /* A few blocks at end hold a flash BBT ... created by TI's CCS
     * using flashwriter_nand.out, but ignored by TI's versions of
     * Linux and u-boot.  We boot faster by using them.
     */
};
```

Çekirdek command line partition table parsing seçeneği seçilerek derlenmişse, mtdparts komut satırı seçeneği kullanılarak partisyon yapısı dinamik olarak oluşturulabilir. DM644x EVM için kullanımına örnek aşağıdaki gibidir.

```
setenv bootargs 'console=ttyS0,115200n8  noinitrd  root=/dev/mtdblock2  rw rootfstype=
jffs2 mtdparts=davinci_nand.0:512K(Onyukleyici),4M(Cekirdek),40M(Dosya_Sistemi),-(Ek)'
```

mtd id bilgisine(davinci_nand) çalışan bir sistemde aşağıdaki gibi ulaşılabilir, id'ye eklenen 0 aygıt numarasıdır.

```
root@dm6446-evm:~# ls /sys/bus/platform/drivers
davinci_emac  dm9000        gen_nand      i2c_davinci
davinci_nand  edma          gpio-keys     serial8250
```

7 Çekirdek ve Dosya Sisteminin NAND Flash üzerine yazılması:

Çekirdek u-boot önyükleyicisi üzerinden yazılmasına karşın jffs2 dosya sistemi bu şekilde yazıldığında sorunlar yaşandığından dosya sistemi Linux üzerinden yazılmıştır. Çekirdeğin yazılma aşamaları aşağıdadır. Rakamlar başlarına 0x getirilmesede 16'lık tabanda yorumlanmaktadır. u-boot önyükleyicisinde "help nand" şeklinde yardım alınabilir.

```
nand erase 80000 400000
tftpboot 0x82000000 uImage
nand write 0x82000000 80000 400000
```

Cihaz nfs üzerinden açıldıktan sonra, ilgili mtdblock aygıt jffs2 türünde mount edilip dosya sistemi arşivi içinde açılabilir. rootfs.tar isimli bir dosya sistemi arşivi nand flash üzerine aşağıdaki gibi yazılabilir.

```
mkdir -p /mnt/nand
mount -t jffs2 /dev/mtdblock2 /mnt/nand
tar xvf rootfs.tar -C /mnt/nand
sync
umount /mnt/nand
```

Bir diğer seçenek Linux üzerinde mtd-utils kullanmak olabilir.

8 Örnek açılış senaryoları:

Geliştirme ve son çalışma durumu için uygun olan iki örnek açılma senaryosuna ilişkin önemli u-boot çevresel değişkenleri ve ilgili komutlar aşağıdaki gibidir.

8.1 Cihazın tftp ve nfs dosya sistemi üzerinden açılması:

```
setenv serverip 172.16.2.68
setenv ipaddr 172.16.2.12
setenv  bootargs 'console=ttyS0,115200n8 noinitrd rw ip=dhcp root=/dev/nfs nfsroot=172
.16.2.68:/home/serkan/hawkNFS/davincifs,nolock mem=120M  mtdparts=davinci_nand.0:512K(
Onyukleyici),4M(Cekirdek),40M(Dosya_Sistemi),-(Ek)'
tftpboot 0x82000000 uImage
bootm 0x82000000
```

8.2 Cihazın NAND üzerinden açılması:

```
setenv bootcmd 'nboot 82000000 0 80000;bootm 0x82000000'
setenv bootargs 'console=ttyS0,115200n8  noinitrd  root=/dev/mtdblock2  rw rootfstype=
jffs2 mtdparts=davinci_nand.0:512K(Onyukleyici),4M(Cekirdek),40M(Dosya_Sistemi),-(Ek)'
```

---

[1]

mtd ve IDE ilgili bağzı anahtarlar asağıdaki gibi atanmıştır.

```
CONFIG_MTD=y
CONFIG_MTD_PARTITIONS=y

CONFIG_MTD_CHAR=y
CONFIG_MTD_BLKDEVS=y
CONFIG_MTD_BLOCK=y

CONFIG_MTD_NAND=y
CONFIG_MTD_NAND_VERIFY_WRITE=y
CONFIG_MTD_NAND_IDS=y
CONFIG_MTD_NAND_DAVINCI=y
```

[2]

```
http://c6000.spectrumdigital.com/davincievm/revd/files/nand_flash_package.zip
```

Linux çekirdeğinin başlarken yaptığı ilk işlemlerden bir tanesi önyükleyici tarafından kendisine geçirilen donanım platformuyla ilgili makina numarasını, kendisinin desteklediği makina numaraları ile karşılaştırmaktır. Önyükleyici tarafından yanlış bir makina numarası gönderildiğinde cihazın açılış mesajları aşağıdaki gibi sonlanacaktır.

```
Starting kernel ...
Uncompressing Linux................... done, booting the kernel
```

Starting kernel mesaji önyükleyicinin son, Uncompressing Linux ile başlayan mesaj ise çekirdeğin ilk mesajıdır. Çekirdek kodu içerisinde konsol(burada ttyS0) başlatılmadan önce çalışan kodlarla ile mesajları görmek için çekirdek CONFIG_DEBUG_LL anahtarı set edilerek derlenmelidir. Cihaz çekirdeğe, konsol argümanına ek olarak, "earlyprintk=serial,ttyS0" argümanları geçirilerek başlatıldığında ekran çıktısı aşağıdaki gibi olmaktadır. Desteklenen makina tipleri çekirdeğin konfigürasyonuna göre birden çok olabilir.

```
Starting kernel ...
Uncompressing Linux... done, booting the kernel.
Error: unrecognized/unsupported machine ID (r1 = 0x00000356).
```

```
Available machine support:

ID (hex)        NAME
00000385        DaVinci DM644x EVM
```

NAND flash paketinden çıkan u-boot çekirdeğe DAVINCI_EVM'in numarası 901 sayısını göndermek yerine DAVINCI_DVDP'nin numarası olan 854'ü göndermektedir(açılış ekranında bu sayıların 16'lık tabandaki karşılıkları görülmektedir). Çekirdek kaynak kodu içeresindeki mach-types.h dosyasinda aşağıdaki değişiklik yapılarak bu sorun aşılabilir.

```
//#define MACH_TYPE_DAVINCI_DVDP         854
#define MACH_TYPE_DAVINCI_EVM            854
//#define MACH_TYPE_DAVINCI_EVM          901
```

Bir diğer problem ise çekirdeğin FLASH'dan hafızaya(RAM) çekildikten sonra başlatılması sırasında doğrulama hatasının alınmasıdır. Hata mesajı aşağıdaki gibidir.

```
DaVinci EVM # bootm 0x82000000
## Booting image at 82000000 ...
   Image Name:   Linux-2.6.35
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    1557300 Bytes =  1.5 MB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... Bad Data CRC
OK
```

[3]

```
serkan@serkanDesktop:~/garbage/temporary/davinci/DM644x_FlashAndBootUtils_2_00/DM644x/
GNU$ ./sfh_DM644x.exe -flash  -flashType NAND -targetType DM6446 -p /dev/ttyUSB1 ubl/u
bl_DM6446_NAND.bin u-boot.bin
-----------------------------------------------------
   TI Serial Flasher Host Program for DM644x
   (C) 2010, Texas Instruments, Inc.
   Ver. 1.67
-----------------------------------------------------


Flashing UBL and application image: ubl/ubl_DM6446_NAND.bin and u-boot.bin. the NAND f
lash device on the DM6446
```

```
Attempting to connect to device /dev/ttyUSB1...
Press any key to end this program at any time.


Waiting for the DM644x...
BOOTME commmand received. Returning ACK and header...
ACK command sent. Waiting for BEGIN command...
Target:  BEGIN
BEGIN commmand received. Sending CRC table...
 100% [ ################################################## ]
                           CRC table sent.

Waiting for DONE...
DONE received.  Sending the UBL...
 100% [ ################################################## ]
                              UBL sent.

DONE received.  UBL was accepted.

Waiting for SFT on the DM644x...
Target:BOOTUBL
BOOTUBL commmand received. Returning CMD and command...
CMD value sent.  Waiting for DONE...
Target:   DONE
DONE received. Command was accepted.
Sending the UBL image
Waiting for SENDIMG sequence...
ACK command sent. Waiting for BEGIN command...
BEGIN commmand received.
 100%################################################## ]
                  Image data transmitted over UART.

Waiting for DONE...
DONE received.  All bytes of image data received...
Target:Number of blocks needed for header and data: 0x0x00000001
Target:NAND block 0x00000001 is bad!!!
Target:NAND block 0x00000002 is bad!!!
Target:Attempting to start in block number 0x0x00000003.
Target:Magicnum: 0x0xA1ACED00
Target:Entrypoint: 0x0x00000100
Target:Numpage: 0x0x00000019
Target:Writing header and image data to Block 0x00000003, Page 0x00000000
Target:SENDING
Target:   DONE
Sending the Application image
Waiting for SENDIMG sequence...
ACK command sent. Waiting for BEGIN command...
BEGIN commmand received.
 100%##################################################]
                  Image data transmitted over UART.

Waiting for DONE...
```

```
DONE received.  All bytes of image data received...
Target:Number of blocks needed for header and data: 0x0x0000000D
Target:NAND block 0x00000006 is bad!!!
Target:NAND block 0x00000007 is bad!!!
Target:Attempting to start in block number 0x0x00000008.
Target:Magicnum: 0x0x55424CBB
Target:Entrypoint: 0x0x81080000
Target:Numpage: 0x0x00000197
Target:Writing header and image data to Block 0x00000008, Page 0x00000000
Target:   DONE

Operation completed successfully.
```

KAYNAKLAR

- http://processors.wiki.ti.com/index.php/Serial_Boot_and_Flash_Loading_Utility
- http://c6000.spectrumdigital.com/davincievm/revd/files/DaVinciEVM_FAQ.html
- http://processors.wiki.ti.com/index.php/Kernel_-_Common_Problems_Booting_Linux

> **Uyarı:** Nand'e uboot'tan yazarken blocksize'ın katı olmalıdır.

BeagleBoard

Bu bölümde, cihaz üzerinde herhangi bir alt seviye yazılım yok iken, cihazın nasıl açıldığına göz atacağız.

BeagleBoard seri port (*UART Recovery*) veya SD kart üzerinden (*MMC Recovery*) açılabilir. Biz burada ikinci yöntemi kullanacağız.

Cihazı açabilmek için bir önyükleyici (*bootloader*) çiftine, Linux çekirdeğine ve dosya sistemine ihtiyacımız olacak. Cihaza enerji verildiğinde, ilk olarak, işlemci yongasına üretim aşamasında kodlanan önyükleyici (*ROM Bootloader*) çalışmaktadır. Bu ilk önyükleyiciyi, işlev olarak, bilgisayarlarımızdaki *BIOS* yazılımına benzetebiliriz. Sonrasında, bu önyükleyici tarafından *MLO* (*MMC Loader*) isimli küçük bir önyükleyici yüklenmekte, *MLO* ile de, çekirdeği yüklemekten sorumlu ikinci önyükleyici olan *U-Boot* yüklenmektedir.

Şimdi sırasıyla önyükleyici çiftini, çekirdeği ve dosya sistemini nasıl oluşturabileceğimize kısaca bakalım.

Elde edeceğimiz nihai dosyaları bir dizinde toplamak, sonrasında SD karta yazma işleminde, işimizi kolaylaştıracaktır. Bu dizinin yol ifadesini *DEV_DIR* isimli bir çevre değişkeninde tutalım.

```
export DEV_DIR=<Dizin Yol İfadesi>
```

> **Not:** Biz derleme sürecinde *CodeSourcery* geliştirme araçlarını kullanacağız. Bu aşamada çapraz derleyicinizin yol ifadesinin *PATH* çevre değişkeninde tanımlı olduğundan emin olunuz.
>
> Ayrıca başka bir derleyici kullanıyorsanız, önyükleyici ve çekirdek derleme aşamalarında, derleyicinize ait öneki (*prefix*) kullanmalısınız. *CodeSourcery* için derleyici öneki *arm-none-linux-gnueabi-* şeklindedir.

Önyükleyici Çiftinin Derlenmesi

U-Boot

U-Boot çekirdeği yüklemekten sorumlu daha gelişmiş bir önyükleyicidir. U-Boot ayrıca kendisini yüklemek için kullanacağımız MLO konunu da barındırmaktadır.

Aşağıdaki gibi indirip, cihaz için derleyebilirsiniz.

```
git clone git://git.denx.de/u-boot.git
cd u-boot
make CROSS_COMPILE=arm-none-linux-gnueabi- omap3_beagle_defconfig
make CROSS_COMPILE=arm-none-linux-gnueabi-
cp MLO u-boot.img ${DEV_DIR}
```

İşletim Sistemi Çekirdeği

BeagleBoard için çekirdek kodunu aşağıdaki gibi indirip sonrasında derleyebilirsiniz. Burada *2.6.32* versiyonunu derleyeceğiz.

> **Not:** TI firması, *C6Run* projesini *3.X.X* versiyonlu çekirdekler için desteklememektedir. *3.X.X* versiyonlu çekirdekler için de projeyi derlemek mümkün olmasın karşın çok fazla sorunla karşılaşılmaktadır, ayrıca *2.6.36* versiyonunundan önceki bir versiyonu seçmenizi öneririz. *2.6.36* ve sonrası için *C6Run* kaynak kodunda bazı değişiklikler yapmak gerekmektedir.

```
git clone git://github.com/RobertCNelson/stable-kernel.git
./patch.sh
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
cd linux-stable/
git checkout v2.6.32
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- omap3_beagle_defconfig
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- uImage
cp arch/arm/boot/uImage ${DEV_DIR}
```

arch/arm/boot/ altında *U-Boot* tarafından yüklenebilecek *uImage* çekirdek imajı oluşacaktır.

> **Uyarı:** uImage dosyasını oluşturabilmek için sisteminizden *mkimage* uygulaması bulunmalıdır, aksi halde derleme sürecinin sonunda aşağıdaki gibi bir hata ile karşılaşılacaktır.
>
> **"mkimage" command not found - U-Boot images will not be built**
>
> mkimage uygulaması Debian tabanlı dağıtımlarda *u-boot-tools* paketinden çıkmaktadır.

Dosya Sistemi

Basit bir dosya sistemini *BusyBox* kullanarak oluşturabileceğiniz gibi hazır bir dosya sistemi de kullanabilirsiniz. Biz burada dosya sistemini kendimiz oluşturacağız.

> **Not:** Hazır bir dosya sistemi kullanmak istiyorsanız, *Angstrom* dağıtımın dosya sistemini kullanabilirsiniz. Bu dosya sistemini, aşağıdaki bağlantıyı kullanarak, internet üzerinden de kolaylıkla oluşturabilirsiniz.
>
> http://narcissus.angstrom-distribution.org/
>
> Bu durumda çapraz derleyicinizin kullandığı standart C kütüphanesi ile dosya sistemi bünyesindekinin uyumluluğunu kontrol etmelisiniz.

Dosya sistemini oluşturmak için temel olarak *BusyBox* kodunu derleyecek, asgari açılış betiklerini, aygıt düğümlerini oluşturacak ve çapraz derleyicinin kullandığı standart C kütüphenesini dosya sistemine kopyalayacağız.

BusyBox kodunu aşağıdaki gibi indirip derleyebilirsiniz.

```
export ROOTFS=<Dosya Sisteminin Saklanacağı Dizin Yolu>
mkdir -p $ROOTFS
wget -c http://www.busybox.net/downloads/busybox-1.23.0.tar.bz2
tar xf busybox-1.23.0.tar.bz2
cd busybox-1.23.0
make defconfig
make CROSS_COMPILE=arm-none-linux-gnueabi-
make CROSS_COMPILE=arm-none-linux-gnueabi- CONFIG_PREFIX=${ROOTFS} install
```

Bu aşamadan sonra *ROOTFS* çevre değişkeniyle gösterilen dizinde asgari dizin yapısı oluşacaktır.

Geliştirme araçlarının içindeki *lib* dizinini hedef dosya sisteminde / altına kopyalayalım.

```
cp -a <Geliştirme Araçları Ana Dizini>/arm-none-linux-gnueabi/libc/lib ${ROOTFS}
```

Açılış ve sonrasında ihtiyaç duyulacak olan temel dizinleri aşağıdaki gibi oluşturabiliriz.

```
mkdir -p ${ROOTFS}/etc/init.d
mkdir -p ${ROOTFS}/dev
mkdir -p ${ROOTFS}/tmp
mkdir -p ${ROOTFS}/proc
mkdir -p ${ROOTFS}/home/root
```

Bu aşamada gerekli açılış betiklerini oluşturalım. *inittab* ve *rcS* olmak üzere iki adet betiğe ihtiyacımız olacak. *BusyBox* içinden çıkan örnek *inittab* dosyasını hedef dosya sistemimize koplayalayıp üzerinde değişiklik yapabiliriz.

```
cp examples/inittab ${ROOTFS}/etc
```

Cihaza seri terminal üzerinden ulaşabilmek için, *inittab* dosyasına aşağıdaki satır eklenmeli veya açıklama şeklinde olan benzer satır aynı forma dönüştürülmelidir.

```
::respawn:/sbin/getty -L ttyS0 115200 vt100
```

/etc/init.d altında ise içeriği aşağıdaki gibi olan *rcS* betiği oluşturulmalı ve gerekli izinleri verilmelidir.

```
#!/bin/sh
mount -t proc proc /proc
mount -t devpts devpts /dev/pts
```

rcS için aşağıdaki gibi gerekli izinleri verebiliriz.

```
chmod 755 etc/init.d/rcS
```

Son olarak, */dev* altındaki gerekli aygıt düğümleri ve *pts* dizinini aşağıdaki gibi oluşturabiliriz.

```
mknod -m 666 ${ROOTFS}/dev/tty c 5 0
mknod -m 666 ${ROOTFS}/dev/console c 5 1
mknod -m 666 ${ROOTFS}/dev/tty1 c 4 1
mknod -m 666 ${ROOTFS}/dev/tty2 c 4 2
mknod -m 666 ${ROOTFS}/dev/tty3 c 4 3
mknod -m 666 ${ROOTFS}/dev/tty4 c 4 4
mknod -m 666 ${ROOTFS}/dev/random c 1 8
mknod -m 666 ${ROOTFS}/dev/urandom c 1 9
mknod -m 666 ${ROOTFS}/dev/ptmx c 5 2
mknod -m 666 ${ROOTFS}/dev/zero c 1 5
mknod -m 666 ${ROOTFS}/dev/null c 1 3
mkdir -p ${ROOTFS}/dev/pts
```

Bu aşamada dosya sistemimizi hazırlamış olduk, daha sonra SD kart üzerine yazmak için dosya sistemimizi arşiv halinde saklayabiliriz.

```
tar czvf rootfs.tgz -C ${ROOTFS} .
cp rootfs.tgz ${DEV_DIR}
```

SD Kartın Hazırlanması

SD kart üzerinde, biri *FAT32* diğeri ise *ext3* şeklinde biçimlendirilecek iki adet bölüme (*partition*) ihtiyacımız olacak. *FAT32* bölümüne önyükleyicileri ve çekirdeği, *ext3* bölümüne ise dosya sistemini taşıyacağız. SD kart üzerindeki bölümlendirme ve biçimlendirme işlerini, komut satırından elle yapmak yerine, *Angstrom* dosya sistemi bünyesindeki *mkcard.txt* betiğini kullanabiliriz. Betiği aşağıdaki gibi indirebilirsiniz.

```
wget -c http://downloads.angstrom-distribution.org/demo/beagleboard/mkcard.txt
```

Betiğe SD kartımıza ilişkin aygıt düğümünü (*device node*) geçirmeliyiz, sistemimiz için örnek aşağıdaki gibidir:

```
./mkcard.txt /dev/mmcblk0
```

> **Not:** SD kartınıza ilişkin aygıt düğümünü *fdisk -l* ile sisteminizdeki bölümleri inceleyerek bulabilirsiniz. *fdisk* komutunu *root* haklarıyla çalıştırmalısınız.
>
> **Not:** Betiğe daha yakından baktığımızda, sırasıyla SD kartın boyutunun hesaplandığını, *sfdisk* ile iki adet bölümün oluşturulduğunu, sonrasında bu bölümlerin *mkfs.vfat* ve *mke2fs* ile biçimlendirildiğini görmekteyiz. FAT bölümünün *bootable* olduğuna dikkat ediniz.

Bu aşamadan sonra, önyükleyici çiftiyle çekirdeği *FAT* bölümüne kopyalayabilir ve önceden hazırladığımız dosya sistemi arşivini *ext3* bölümüne açabiliriz.

```
cp ${DEV_DIR}/MLO /media/boot
cp ${DEV_DIR}/u-boot.img /media/boot
cp ${DEV_DIR}/uImage /media/boot
tar xf ${DEV_DIR}/rootfs.tgz -C /media/Angstrom
```

BeagleBoneBlack

Beagle Bone Black son zamanlarda popüler olan, 50$ mertebesindeki fiyatı ve genişleme imkanlarıyla öne çıkan bir geliştirme platformudur.

İşlemci	**TI AM335x 1GHz ARM® Cortex-A8**
RAM	512MB DDR3
Storage	2GB beya 4GB 8-bit eMMC on-board flash
GPU	3D hızlandırıcı
NEON Desteği	Var
USB Host	Var
USB Power	Var
Micro SD Kart	Var
Ethernet	Var
HDMI	Var
Konsol Kablosu	Yok, ayrıca 3.3V FTDI - TTL dönüştürücü kablosu alınmalı
NAND Flash	Yok

Konsol Kablosu

- BeagleBone-Black versiyonunun beraberinde konsol kablosu gelmemektedir
- Bootloader seviyesinde konsol erişimi için FTDI-TTL 3.3V USB dönüştürücü kablosuna ihtiyaç bulunmaktadır.

Çalışma Dizinin Hazırlanması

- Bu dokümanda kullanıcının ev dizini altında, beagle alt dizini oluşturulduğu varsayılmıştır:

```
$ mkdir $HOME/beagle
```

- Linux kernel kaynak kodları **beagle/kernel** dizininde bulundurulacaktır
- Buildroot kaynak kodları, **beagle/buildroot** dizininde bulundurulacaktır

Buildroot BeagleBoneBlack Kernel

- Buildroot içerisinde BeagleBoard için hazırlanmış öntanımlı kernel derleme konfigürasyonu bulunmaktadır
- Ancak Ağustos 2013 itibariyle, bu konfigürasyon ile çalışan BeagleBoardBlack kernel imajı üretmek mümkün değildir
- Kernel derleme işleminin Buildroot dışında ayrıca yapılması gerekmektedir

Buildroot Kaynak Kodunun İndirilmesi

Buildroot kaynak kodları git üzerinden klonlanmalıdır:

```
$ git clone git://git.buildroot.net/buildroot
```

Out-Of-Tree build yöntemi ile `$HOME/beagle` dizini altında build dizini içerisinde buildroot ana çalışma dizinimizi oluşturalım:

```
$ cd buildroot
$ make O=$HOME/beagle/build beaglebone_defconfig
```

İşlem bitiminde ana beagle çalışma dizininiz altında, **build** dizini oluşacaktır.

Bu noktadan sonra *buildroot* ile ilgili tüm işlemlerinizi, *build* dizini altında iken gerçekleştirmelisiniz:

```
$ cd $HOME/beagle/build
```

Buildroot Konfigürasyonu

Buildroot içerisinden çıkan **beaglebone_defconfig** dosyasında board ile ilgili temel ayarlar mevcuttur.

Bununla birlikte aşağıdaki temel başlıklar için konfigürasyon üzerinde düzenlemeler yapılmalıdır:

- Kullanılacak toolchain seçimi
- Dosyaların download edileceği dizinle ilgili düzenlemeler
- İşlem sonucunda oluşturulacak dosya sistemi tipi
- **/dev** yönetimi için tercih edilen yöntem
- Sistemde yer alması istenilen paketler
- Bunlar temel düzenlemeler olup, öncelikle bu başlıklara değinilecektir.

Toolchain Seçimi

Buildroot içerisinde toolchain seçiminde *External Toolchain* kullanımı önerilir. External Toolchain kullanılacağı belirtildikten sonra, toolchain tipi seçilmelidir

Genelde 3 toolchain tipi bulunur:

- Linaro Toolchain versiyonları
- Sourcery CodeBench Toolchain versiyonları
- Custom toolchain

BeagleBone Black için Linaro toolchain güncel versiyonu seçilmelidir. Sourcery CodeBench toolchain'leri henüz **EABIhf** ABI desteğine sahip olmadığından bu menüden seçilememektedir

Download Dizin Seçimi

Buildroot ile çalışırken, özellikle birden fazla projede buildroot kullanıldığında veya aynı projede farklı konfigürasyonlarda build işlemleri gerektiğinde, dosyaların her seferinde yeniden download edilmesi işlemleriyle vakit kaybetmemek için, genel bir download dizini belirtmekte fayda vardır.

Örneğimizde download dizini `/opt/buildroot/downloads` şeklinde seçilmiş olup aşağıdaki komutlarla öncelikle bu dizini oluşturabilirsiniz:

```
$ sudo mkdir -p /opt/buildroot/downloads
$ sudo chown -R $USER /opt/buildroot/downloads
```

Dosya Sistemi Seçimi

Buildroot ile işlem sonunda `cloop` , `cpio` , `cramsfs` , `ext2` , `jffs2` , `romfs` , `squashfs` , `tar` ve `ubifs` dosya sistemi imajları çıkartılabilir.

BeagleBone-Black üzerinden NAND flash değil, eMMC bulunduğundan *ext4* gibi blok tabanlı aygıtlar üzerinde çalışan dosya sistemleri kullanılmalıdır.

Kernel Kaynak Kodunun İndirilmesi

Kaynak kodlar git üzerinden klonlanıp, 3.8 branch'i checkout edilmelidir:

```
$ git clone git://github.com/beagleboard/kernel.git
$ cd kernel
$ git checkout 3.8
$ ./patch.sh
$ cp configs/beaglebone kernel/arch/arm/configs/beaglebone_defconfig
$ wget "http://arago-project.org/git/projects/?p=am33x-cm3.git;a=blob_plain;f=bin/am33
5x-pm-firmware.bin;hb=HEAD" -O kernel/firmware/am335x-pm-firmware.bin
```

Kernel Derleme

Derleme işlemi öncesinde oluşan kernel dosyasını sıkıştırma ve u-boot kernel başlık bilgilerini eklemek için gerekli yardımcı araçların kurulu olduğundan emin olunuz:

```
$ sudo apt-get install lzop u-boot-tools
```

Kernel derleme işlemini aşağıdaki adımlarla yapabilirsiniz (tftp sunucu ana dizini `/srv/tftp` olduğu varsayılmıştır):

```
$ make ARCH=arm beaglebone_defconfig
$ make ARCH=arm \
CROSS_COMPILE=/home/demirten/beagle/build/host/usr/bin/arm-linux-gnueabihf- \
uImage dtbs -j4
$ make ARCH=arm \
CROSS_COMPILE=/home/demirten/beagle/build/host/usr/bin/arm-linux-gnueabihf- \
uImage-dtb.am335x-boneblack -j4
$ cp arch/arm/boot/uImage-dtb.am335x-boneblack /srv/tftp/uImage
```

Nfs-Root Çalışma

Aşağıdaki şekilde u-boot üzerinden temel değişkenleri ayarlayarak NFS üzerinden boot işlemini gerçekleştirebilirsiniz:

```
setenv serverip 192.168.7.1
setenv ipaddr 192.168.7.2
setenv console ttyO0,115200n8
setenv rootpath /home/demirten/beagle/build/target
setenv bootargs console=${console} root=/dev/nfs nfsroot=${serverip}:${rootpath},vers=
3 rw ip=${ipaddr}
tftp 0x80200000 uImage-dtb.am335x-boneblack
bootm
```

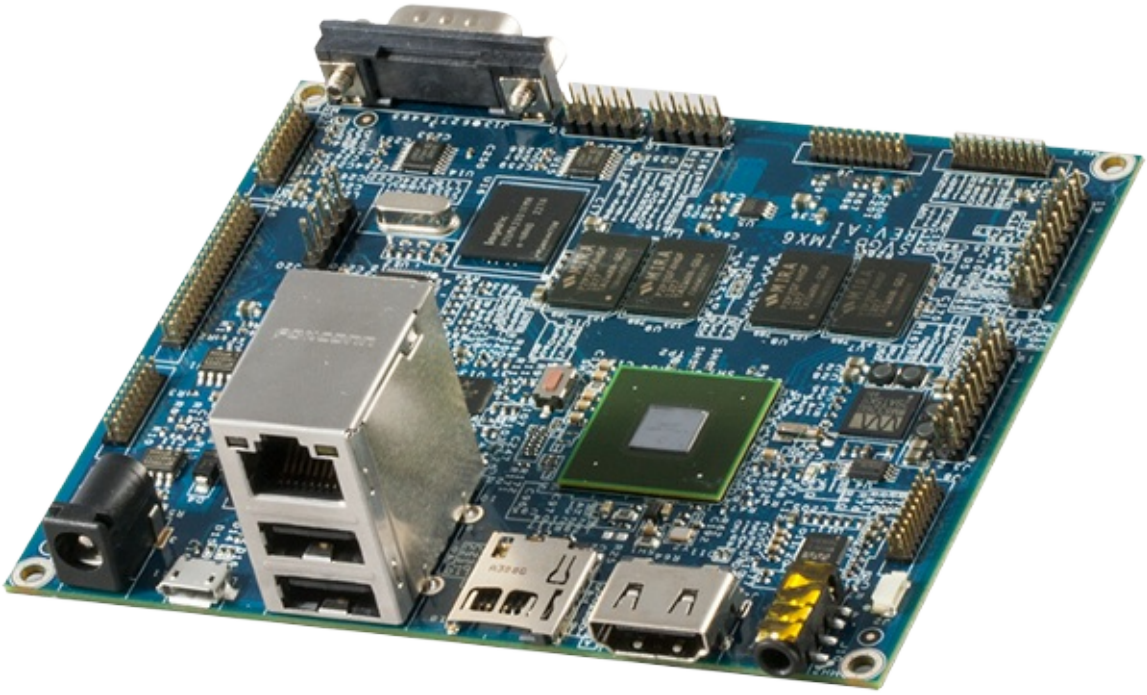
uEnv.txt Kullanımı

BeagleBoneBlack için hazırlanan u-boot imajı, çalışmaya başladığında `uEnv.txt` dosyası bulması halinde içerisindeki değişkenleri import etmektedir. Bu metin dosyasını doğru şekilde düzenlemek suretiyle açılış işlemini u-boot imajını yeniden üretmek zorunda kalmaksızın kalıcı olarak özelleştirmeniz mümkündür.

Örnek bir `uEnv.txt` dosyasının içeriği aşağıda listelenmiştir:

```
kernel_file=zImage
initrd_file=uInitrd
serverip=192.168.7.1
ipaddr=192.168.7.2
rootpath=/home/demirten/beagle/rootfs
console=ttyO0,115200n8

loadkernel=load mmc ${mmcdev}:${mmcpart} 0×80200000 ${kernel_file}
loadinitrd=load mmc ${mmcdev}:${mmcpart} 0×81000000 ${initrd_file}; setenv initrd_size
 ${filesize}
loadfdt=load mmc ${mmcdev}:${mmcpart} 0x815f0000 /dtbs/${fdtfile}

netargs=setenv bootargs console=${console} ${optargs} root=/dev/nfs nfsroot=${serverip
}:${rootpath},vers=3 rw ip=${ipaddr}

#just zImage
boot_ftd=run loadkernel; run loadfdt
uenvcmd=run boot_ftd; run netargs; bootz 0×80200000 – 0x815f0000

#zImage + uInitrd: where uInitrd has to be generated on the running system.
#boot_ftd=run loadkernel; run loadinitrd; run loadfdt
#uenvcmd=run boot_ftd; run mmcargs; bootz 0×80200000 0×81000000:${initrd_size} 0x815f0
000
```

Savage Board



İşlemci	Freescale i.MX6 Cortex A9 1Ghz / Quad Core
RAM	DDR3 1GB
Storage	eMMC 8GB
GPU	3D GPU Vivante GC2000
Hard FP (eabihf)	Var
USB	2x Ext / 1x Int 1with mPCIe
SATA	Var
Micro SD Kart	Var
Ethernet	Var (Gigabit)
HDMI	Var
Konsol	Var, RS232 DB9
NAND Flash	Yok

Toolchain Seçimi

Savage Board hard-floating desteğine sahip olduğundan **eabihf** toolchain'ler kullanılabileceği, **soft-float** toolchain'ler de kullanılabilmektedir.

Güncel Linaro, CodeSourcery ve Arago toolchain'leri sorun yaşamadan kullanabilirsiniz.

U-boot

http://www.savageboard.org adresinden *Download* bölümünden `uboot-savage-1.5.tar.gz` dosyası indirilmelidir. Aşağıdaki şekilde güncel versiyonu indirilip hardfp veya softfp toolchain'ler ile derlenebilir:

```
$ wget http://www.savageboard.org/Downloads/uboot-savage-1.5.tgz
$ tar xf uboot-savage-1.5.tgz
$ cd uboot-imx
$ make CROSS_COMPILE=arm-none-linux-gnueabi- mx6q_savage_config
$ make CROSS_COMPILE=arm-none-linux-gnueabi- -j4
```

İşlem bitiminde `uboot.bin` dosyası oluşacaktır. Bu dosyayı kullanarak sistemi recovery yöntemiyle açabilirsiniz. Ancak derlediğiniz u-boot binary imajını eMMC içerisinde yer alan imaj ile değiştirmek için yapılması gereken ek işlemler bulunmaktadır. Konunun detayına geçmeden, iMX6 serisinde kullanılan **IVT Tablosu Kullanımı** konusuna değinmemiz gereklidir.

IVT Tablosu

U-boot Konsolunda Çalışmak

```
mmc dev 2
setenv bootargs console=ttymxc0,115200 root=/dev/mmcblk0p2 rootwait rw
fatload mmc 2:1 0x12000000 uImage
bootm 0x12000000
```

Kernel

http://www.savageboard.org adresinden *Download* bölümünden `kernel_savage-1.5.tgz` dosyası indirilmelidir.

```
$ wget http://www.savageboard.org/Downloads/savage-arch-kernel-3.10.17-1.1.tar.gz
$ tar xf savage-arch-kernel-3.10.17-1.1.tar.gz
$ cd savage-kernel-3.10.17-20141127
```

```
$ cp arch/arm/config/savage_defconfig .config
$ make ARCH=arm imx6q-savage.dtb uImage -j8 \
    LOADADDR=0x10008000 DTB=imx6q-savage.dtb \
    CROSS_COMPILE=arm-none-linux-gnueabi-
```

Modülleri derlemek için:

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- modules -j8
```

Modüller derlendikten sonra `modules_install` target'ı kullanılarak, modüllerin kurulacağı kök dosya sistemi `INSTALL_MOD_PATH` parametresi ile aşağıdaki gibi verilmelidir (kök dosya sisteminin `/opt/savage` dizininde oluşturulduğunu varsayarsak):

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- \
    INSTALL_MOD_PATH=/opt/savage modules_install
```

Recovery

Board'un açılmaması, boot yükleyici alanının bozulması vb. durumlarında kullanabileceğiniz bir recovery mekanizması mevcuttur.

Cihazı bu moda geçirebilmek için, micro-usb kablosu cihaz ile bilgisayarınız arasında takılı durumdayken, kartın ortasındaki küçük butona basılı tutup karta enerji vermeniz gereklidir. Bu şekilde açıldığında bilgisayarınızdan `lsusb` ile bakıldığında aşağıdaki gibi bir çıktı görünecektir:

```
$ lsusb
Bus 001 Device 009: ID 15a2:0054 Freescale Semiconductor, Inc. i.MX6Q SystemOnChip in
RecoveryMode
```

Kart recovery modunda iken, micro-usb kablosu üzerinden derlemiş olduğunuz u-boot.bin dosyasını karta gönderebilir ve çalıştırılmasını sağlayabilirsiniz. Ancak bu süreci işletebilmek için, öncelikle **imx_usb_loader** projesini indirip derlemeliyiz. Projeyi derleyebilmeniz sistemde **libusb** development paketleri de kurulu olmalıdır, değilse öncelikle aşağıdaki şekilde paketleri yüklemelisiniz:

```
$ sudo apt-get install libusb-1.0-0-dev libusb-dev
```

Ardından projeyi clone'layıp aşağıdaki şekilde derleyiniz:

```
$ git clone https://github.com/boundarydevices/imx_usb_loader.git
$ cd imx_usb_loader
$ make
```

İşlem bitiminde **imx_usb** uygulaması oluşacaktır. Uygulamayı **sudo** aracılığıyla root erişim haklarıyla çalıştırıp, parametre olarak derlediğiniz u-boot.bin dosyasını vermelisiniz:

```
$ sudo ./imx_usb ../uboot-imx/u-boot.bin
```

EKLER

Bu bölümde belirli bir akış içerisine girmeyen konulara değinilmektedir.

Seri Konsol Kullanımı

Gömülü Linux projeleri üzerinde çalışırken vazgeçilmez unsurlardan biri seri konsol arayüzü üzerinden cihazınıza erişmek olacaktır.

Bunun için geliştirme yaptığınız bilgisayarda seri port bulunması gereklidir. Ancak günümüzde hemen hiç bir bilgisayarda seri port çıkışı bulunmadığından, USB-Seri çevirici aparatlarına ihtiyaç duyulacaktır.

USB - Seri Çeviriciler

Piyasada bulabileceğiniz hemen her usb seri dönüştürücü Linux tarafından otomatik olarak tanınır.

Herhangi bir usb çeviriciyi bilgisayarınıza taktıktan sonra sisteminizde hangi dosya adı ile tanındığını anlamak için `dmesg` komutu ile çekirdek mesajlarının son bölümüne bakabilirsiniz:

```
$ dmesg
[185805.857823] usb 1-1: New USB device found, idVendor=067b, idProduct=2303
[185805.857827] usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[185805.857830] usb 1-1: Product: USB-Serial Controller
[185805.857832] usb 1-1: Manufacturer: Prolific Technology Inc.
[185805.858150] pl2303 1-1:1.0: pl2303 converter detected
[185805.858905] usb 1-1: pl2303 converter now attached to ttyUSB0
```

Yukarıdaki örnekte Prolific markalı seri çeviricinin `ttyUSB0` ismiyle sisteme iliştirildiği belirtilmektedir. Sistemimiz tarafından bu şekilde tanınmış olan seri çeviriciyi, `/dev/ttyUSB0` aygıt dosyası üzerinden kullanabiliriz.

Bilgisayarınıza takılan usb seri çeviriciler Linux tarafından tanındıktan sonra, kullandığınız dağıtıma göre **devtmpfs** veya **udev** çözümlerinden biriyle `/dev` dizini altında otomatik olarak aygıt dosyası doğru **major**, **minor** ve aygıt türünü gösterecek şekilde oluşturulur. Usb seri çeviricilerde isimlendirme aynı anda takılı her bir çeviri için `ttyUSB0` , `ttyUSB1` , `ttyUSBX` şeklinde yapılır.

> Not: Usb seri çevirici takılı durumda ve bir uygulama tarafından kullanımdayken çıkartılıp tekrar takılacak olursa, sistem tarafından yeni bir numara ile isimlendirilip kullanıma sunulacaktır. `dmesg` komut çıktısında her zaman doğru aygıt ismini görebilirsiniz.

Seri Aygıtlar Üzerinde Erişim Yetkisi

Takılan usb çevirici aygıt dosyalarının öntanımlı erişim yetkileri genellikle aşağıdaki gibi olur:

```
$ ls -l /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 0 Jun 10 15:34 /dev/ttyUSB0
```

Yukarıdaki çıktıya bakarak şunları söyleyebiliriz:

- Bu bir karakter tabanlı aygıttır (baştaki **c** harfi)
- Dosya sahibi **root** kullanıcısıdır ve kullanıcının okuma ve yazma yetkileri bulunmaktadır
- Dosyanın grup sahibi **dialout** grubudur ve bu gruba dahil olan kullanıcıların da okuma ve yazma yetkileri bulunmaktadır
- Geri kalan kullanıcıların dosya üzerinde herhangi bir okuma ve yazma hakkı yoktur

Geliştirme yaptığımız bilgisayarımızı **root** kullanıcısıyla değil de normal bir kullanıcı hesabıyla kullandığımızı düşündüğümüzde, eğer **dialout** grubuna üye değil isek sistemimize taktığımız usb seri çevirici cihazı üzerinde okuma-yazma yapamayacağımız görülmektedir.

Sorunun çözümü için ya kendi kullanıcımızı **dialout** grubuna üye yapmalı ya da sistemimizdeki **udev** kural dosyalarını düzenleyerek, usb seri çeviricileri takıldığında dosyanın erişim yetkilerinin kendi kullanıcımız ile erişebileceğimiz bir mod ile ayarlanmasını sağlamalıyız.

Kolaylık açısından birinci yöntem izlenmelidir. Öncelikle `id` komutu ile kendi kullanıcımızın hangi gruplara üye olduğunu öğrenelim:

```
$ id
uid=1000(demirten) gid=1000(demirten) groups=1000(demirten),27(sudo),29(audio)
```

Görüldüğü üzere **dialout** grubuna üyeliğimiz bulunmuyor. Gruba kendimizi üye yapmak için Debian tabanlı sistemlerde **aduser** uygulamasını kullanabiliriz:

```
$ sudo adduser demirten dialout
Adding user `demirten' to group `dialout' ...
Adding user demirten to group dialout
Done.
```

Grup üyeliğimiz gerçekleşti. Ancak grafik oturumlarında grup üyelik bilgilerini ilk login işleminde kontrol edilip, sonrasında çalışan tüm süreçlere aynı bilgiler aktarılmaktadır. Bu sebeple değişikliklerin etkin olması için grafik oturumundan çıkış yaptıktan sonra tekrar giriş yapmalı veya sisteminizi yeniden başlatmalısınız. Bu işlemin ardından seri çevirici cihaz üzerinde okuma - yazma yetkileriniz olacak ve rahatlıkla çalışabileceksiniz.

Seri Konsol Uygulamaları

minicom

Oldukça eski fakat halen iş gören bu uygulamayı aşağıdaki gibi sisteminize kurabilirsiniz:

```
$ sudo apt-get install minicom
```

Uygulama ilk açıldığında öntanımlı olarak `/dev/modem` gibi bir aygıt dosyasını açmaya çalışır, dosyayı bulamadığında sonlanır.

Bu sorunun üstesinden gelmek için uygulamayı `-s` parametresi ile doğrudan ayarlar ekranıyla açabilirsiniz:

```
$ minicom -s
+-----------------------------------------------------+
| A -    Serial Device      : /dev/ttyUSB0            |
| B - Lockfile Location     : /var/lock               |
| C -   Callin Program      :                         |
| D -  Callout Program      :                         |
| E -    Bps/Par/Bits       : 115200 8N1              |
| F - Hardware Flow Control : No                      |
| G - Software Flow Control : No                      |
|                                                     |
|    Change which setting?                            |
+-----------------------------------------------------+
        | Screen and keyboard      |
        | Save setup as dfl        |
        | Save setup as..          |
        | Exit                     |
        | Exit from Minicom        |
        +--------------------------+
```

A tuşuna basarak aygıt ismini, **E** tuşu ile seri port hızını ayarlayabilirsiniz. Ardından **Exit** ile çıktığınızda ilgili seri portu kullanabiliyor olacaksınız.

Uygulama ana ekranında iken `CTRL-A + O` ile konfigürasyon ekranına dönebilir, `CTRL-A + X` ile uygulamadan çıkabilir, `CTRL-A + W` ile *line wrap* modunu aktifleştirebilir, `CTRL-A + Z` ile diğer kısayollar hakkında yardım alabilirsiniz.

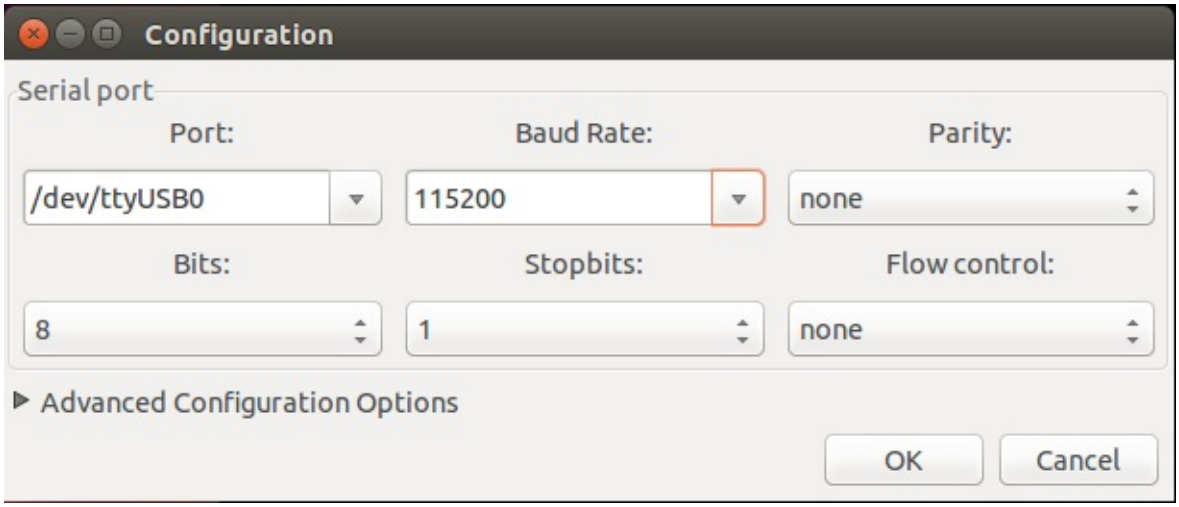
minicom uygulaması genellikle alışkanlık edinmiş kişilerce kullanılır. Yeni başlayanlar için kullanılması pek önerilmez.

gtkterm

Uygulamayı aşağıdaki gibi sisteminize kurabilirsiniz:

```
$ sudo apt-get install gtkterm
```

Uygulama açıldıktan sonra `Configuration->Port` menüsü üzerinden aygıt ismi ve hız parametreleri aşağıdaki gibi ayarlanabilir:



Yapılan ayarları öntanımlı olarak kaydedip daha sonraki kullanımlarınızı da kolaylaştırabilirsiniz.

screen

Çoklu terminal yönetimi için kullanılan **screen** uygulamasına alışkın iseniz seri port erişimi için de aşağıdaki şekilde kullanabilirsiniz:

```
$ screen /dev/ttyUSB0 115200
```

TFTP Sunucu Kurulumu

Protokol

Trivial **F**ile **T**ransfer **P**rotocol, **1980** yılı **Ocak** ayında ilk **RFC** dokümanı *Karen R. Sollins* tarafından yayınlanmış olan, oldukça eski bir protokoldür.

Geliştirildiği zamanlarda temel hedefi ağ üzerinden basit bir şekilde dosya almak ve dosya göndermek idi. Özellikle de sistemlerin ağ üzerinden boot edebilmelerine sağlamak amacıyla, açılış sırasında ihtiyaç duyulan dosyaların transfer edilebilmesi için kullanılıyordu.

Bugün halen daha bu amaçlar için kullanılıyor ve desteklediği özelliklerde de temel anlamda bir değişiklik yok. TFTP basitçe **UDP/IP** üzerinden dosya indirmek ve dosya göndermek için kullanılır. Gelişmiş dosya transfer protokollerinde yer alan kimlik ve yetki denetimi, dosya listeleme, silme, ismini değiştirme vb. gibi hiç bir ek fonksiyona sahip değildir. Ayrıca iletim katmanında **TCP** kullanan gelişmiş dosya transferi protokollerinden farklı olarak **UDP** protokolü üzerinde çalıştığından, dosyaya ait paketlerin karşı tarafa doğru gidip gitmediğinin kontrolü veya paketler için yeniden gönderimin denenmesi gibi özellikleri de bulunmamaktadır. Bu kısıt nedeniyle internet veya geniş alan ağlarından ziyade, yerel ağlarda kullanılması daha uygundur.

Yukarıda sayılan tüm bu olumsuz gibi görünen özelliklere karşın, TFTP protokolünün çok güçlü olduğu bir yanı vardır: **basitlik**

Protokolün gerçeklenmesi, üzerinde işletim sistemi olmayan ortamlar için dahi, alternatiflerine göre oldukça kolaydır ve bu özelliği nedeniyle gömülü sistemlerde de geniş bir kullanım alanı bulmuştur.

Kurulum

Gömülü sistemler üzerinde çalışırken bilgisayarınızda **TFTP** sunucu servisinin bulunması neredeyse zorunludur. Linux platformlarında çalışan birden fazla TFTP sunucu uygulaması mevcuttur. Debian tabanlı bir dağıtım kullanıyorsanız, `tftpd-hpa`, `tftpd` veya `atftpd` paketlerinden birini sisteminize kurabilirsiniz. Biz burada `ftpd-hpa` paketinin kullanılmasını öneriyoruz:

```
$ sudo apt-get install tftpd-hpa
```

Kurulum sonrası TFTP servisi **69** nolu **UDP** portunu dinlemeye başlayacaktır. Dosyaları TFTP sunucu üzerinden diğer sistemlere sunmaya başlamak için, ilgili dosyanın kullanılan TFTP sunucu uygulamasının dinlediği ana dizin veya altındaki bir dizine kopyalanması ve dosya izinlerinin herkes tarafından okunmasına imkan verecek şekilde düzenlenmesi gereklidir.

Paket kurulumu sonrasında TFTP sunucu ana dizininin ne şekilde ayarlandığını öğrenmek için, `/etc/default/tftpd-hpa` dosyasında yer alan `TFTP_DIRECTORY` değişkeninin ne şekilde ayarlanmış olduğuna bakabilirsiniz. Genellikle `/var/lib/tftpboot` veya `/srv/tftp` gibi dizinler kullanılmakta olup, isterseniz bu dizini değiştirip servisi yeniden başlatabilirsiniz.

Kullanım kolaylığı sağlaması açısından, ilgili tftp ana dizininin sahibini kendi kullanıcınız olacak şekilde değiştirirseniz, sonrasında bu dizine yapacağınız kopyalamalarda **root** olmanıza gerek kalmayacaktır:

```
$ sudo chown -R $USER /var/lib/tftpboot
```

> **NOT:** Tftp sunucu paket adları ve öntanımlı olarak baktığı ana dizinler kullanılan Linux dağıtımına göre farklılık gösterebilir.

Dosya Gönderme

Bazen gömülü Linux sisteminizden bir dosyayı dış ortama taşımak istediğinizde TFTP'nin tek alternatif olduğu durumlarla karşılaşabilirsiniz (sisteme dosyayı yazabileceğiniz herhangi bir yazılabilir ortam takılamıyor olabilir vb.) Bu durumda muhtemelen **busybox** içerisinde **tftp** istemcisi de derlenmiş olacağından, sistemde yer alan bir dosyayı ağ üzerindeki bir tftp sunucusuna gönderebilirsiniz. Tftp istemci uygulamasının genel kullanımı:

```
# tftp
BusyBox v1.23.0 (2015-01-22 18:30:08 EET) multi-call binary.

Usage: tftp [OPTIONS] HOST [PORT]

Transfer a file from/to tftp server

-l FILE    Local FILE
-r FILE    Remote FILE
-g    Get file
-p    Put file
-b SIZE    Transfer blocks of SIZE octets
```

şeklinde olup, bölümü sisteminizdeki örnek.bin dosyasını 192.168.1.100 ip adresli TFTP sunucusuna göndermek için şu şekilde bir komut kullanmanız gerekecektir:

```
# tftp -l örnek.bin -p 192.168.1.100
```

Yukarıdaki komut doğru olmasına rağmen dosyayı TFTP sunucunuza aktarma işleminde hata alacaksınız. Dönen hata mesajı açıklayıcı olmadığından asıl problemin ne olduğunun anlaşılması güçtür.

Buradaki problem, TFTP sunucunun güvenlik nedeniyle kurulduğu sisteme bir dosyanın gönderilebilmesi (upload) için, ön şart olarak dosyanın yazılacağı dizinde, aynı isimle bir dosya olmasını ve bu dosyanın erişim yetkilerinin herkes tarafından yazılabilir olacak şekilde düzenlenmiş olmasını gerektirmesidir.

Yani TFTP sunucuda mevcut olmayan bir dosyanın, tftp istemcileri üzerinden yüklenmesi mümkün değildir. Öncelikle dosyayı oluşturup erişim yetkilerini düzenler isek, yukarıdaki upload işlemi başarıyla sonuçlanacaktır. Bunun için TFTP sunucu sisteminde ilgili TFTP sunucu ana dizininde aşağıdaki komutları çalıştırmalıyız:

```
$ touch örnek.bin
$ chmod 666 örnek.bin
```

Artık upload işlemimizi gerçekleştirebiliriz.

Yukarıda anlattığımız güvenlik senaryosunu devre dışı bırakıp, TFTP sunucunun kendi baktığı dizinde olmayan bir dosyayı da oluşturmasını sağlamak mümkündür. Bunun için `tftpd-hpa` uygulamasını başlatırken `-c` veya `--create` parametresinin verilmiş olması gerekir. Bunun için `/etc/default/tftpd-hpa` dosyasındaki `TFTPD_OPTIONS` değişkeninin mevcut haline bu parametrenin eklenmesi yeterlidir:

```
# /etc/default/tftpd-hpa

TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/srv/tftp"
TFTP_ADDRESS="0.0.0.0:69"
TFTP_OPTIONS="--secure --create"
```

NFS Sunucu Kurulumu

Network **F**ile **S**ystem, Sun Microsystems tarafından 1984 yılında geliştirilmiş, ağa bağlı sistemlerin ağ üzerindeki NFS hizmeti veren sunculardaki paylaşımları yerel diskleri gibi kullanmasına olanak veren, RPC temelli dağıtık dosya sistemi yapısıdır.

Gömülü Linux sistemimiz üzerinde geliştirme yaparken, cihazımızı doğrudan üzerindeki depolama biriminden değil de (NAND Flash, eMMC, MMC vb.) ağ üzerindeki (genellikle geliştirme yaptığımız kendi bilgisayaramızdaki) bir NFS dosya paylaşımı üzerinden açmak çok kullanışlı ve zaman kazandırıcı bir uygulamadır.

Aynı zamanda daha nadir olmakla birlikte, sisteminizi doğrudan NFS paylaşımı üzerinden açmasanız bile, sistem açıldıktan sonra bir NFS paylaşımını mount edip, onun üzerinden dosya paylaşımları gerçekleştirmek de istenebilir.

Her iki senaryonun çalışması için de geliştirme yaptığınız bilgisayarda çalışan bir NFS sunucu kurulumu gerçekleştirilmelidir.

Bunun için Debian tabanlı bir sistem kullanıyorsanız, `nfs-kernel-server` paketini aşağıdaki gibi yüklemelisiniz:

```
$ sudo apt-get install nfs-kernel-server
```

İşlem bitiminde NFS sunucunuz otomatik olarak çalışacaktır. Ancak bu noktada NFS sunucumuz henüz bilgisayarımızda yer alan hangi dizinleri ağ üzerinde paylaşıma açmak istediğimizi bilmediğinden, herhangi bir paylaşım sağlamayacaktır.

Aynı NFS sunucu üzerinde, birden fazla dizini birbirinden farklı yetkilendirme ve kısıtlamalarla ağ paylaşımına açabiliriz.

Herhangi bir dizini NFS sunucu üzerinden paylaştırmak için, `/etc/exports` dosyasına dizinle ilgili bir ayar satırı girilmesi gereklidir. Bu dosyanın sahibi **root** kullanıcısı olduğundan, `sudo` mekanizmasıyla herhangi bir editörle açıp aşağıdaki gibi bir satır ekleyebiliriz:

```
# /etc/exports
/home/demirten/beagle  192.168.100.0/24(rw,no_root_squash,no_subtree_check)
```

NFS sunucu üzerindeki `/etc/exports` dosyasında izin verdiğiniz ip aralıklarının dışındaki bir sistem ilgili kaynağa ulaşmaya çalıştığında, NFS sunucunuz tarafından reddedilecektir. Gömülü sisteminizde nfs mount işlemlerinde `permission denied` mesajları alıyorsanız, NFS

sunucunun çalıştığı bilgisayardaki `/var/log/syslog` dosyasının sonlarında aşağıdakine benzer hata mesajları görünecektir:

```
rpc.mountd[1041]: refused mount request from 192.168.2.2 for
/home/training/beagle/target (/home/training/beagle/target):
unmatched host
```

Yukarıdaki gibi bir `unmatched host` log mesajını gördüğünüzde, ilgili sistem için de paylaşım dizinini erişime açmak istiyorsanız, `/etc/exports` dosyasındaki ilgili kurala ait IP/Ağ Maskesi bölümünü genişletmeniz veya tüm IP adresleri için erişim vermek istiyorsanız, `*` özel karakterini kullanmalısınız.

`/etc/exports` dosyasında değişiklik yaptıktan sonra NFS servisini yeniden başlatabilir:

```
$ sudo service nfs-kernel-server restart
```

veya `exportfs` komutuna `-r` parametresini vermek suretiyle halihazırda paylaşıma açık olmasına rağmen bu örnekteki gibi paylaşımla ilgili herhangi bir ayarı değişmiş dizinleri yeniden paylaştırmasını ve yeni ayarların geçerli olmasını sağlayabiliriz:

```
$ sudo exportfs -r
```

NFS Sunucu - Mount Gecikme Problemi

Sunucu üzerinde NFS protokolünün versiyon 4 ve yukarısı kullanıldığında, dağıtımlarda kullanılan NFS sunucu yazılımı uygulamalarının öntanımlı konfigürasyonları ile geleneksel çalışma senaryolarında, istemci tarafında mount işlemi sırasında 15 saniyeye varan gecikmeler yaşanabilmektedir.

Debian Jessie, Ubuntu 12.04, Fedora 19 ve yeni versiyonlarında bu sorun görünmektedir.

Benzer bir mount işlemlerinde gecikme durumu yaşıyorsanız, sunucu tarafındaki log dosyalarını inceleyerek (/var/log/syslog, /var/log/messages) aşağıdakine benzer bir log mesajı olup olmadığını kontrol edebilirsiniz:

```
... RPC: AUTH_GSS upcall timed out
```

Bu mesaj **Kerberos Authentication** işleminin başarısız olduğu ve zaman aşımına uğradığını belirtiyor. Ağ üzerinde güvenli kimlik denetimi için kullanılan Kerberos protokolü muhtemelen çalışma ortamınızda gerekli olmayacaktır. Bu şekilde yapılandırılmış bir ağda bulunuyor olsanız dahi en azından gömülü Linux sistemlerinizle Kerberos Authentication

mekanizmasını devreye alma ihtiyacınız olmayacaktır. Her defasında mount işlemlerinde bu gecikmeyi yaşamak yerine, sorunu kökünden çözebilirsiniz. Her ne kadar problemin çözümü için NFS sunucu tarafında NFS ile birlikte GSSD servisini çalıştırma yönünde alternatifler önerilmiş olsa da, bu yöntemler tüm dağıtımlarda ve paket versiyonlarında aynı etkiyi göstermediğinden biz sorunu kökünden çözmeyi yeğliyoruz. Bunun için NFS sunucuyu çalıştırdığınız Linux sisteminde, `rpcsec_gss_krb5` kernel modülünün yüklenmesini engellemeniz (karalisteye almanız) gerekiyor. Bilgisayarınızı her açtığınızda bu ayarın devreye girmesi için `/etc/modprobe.d/blacklist-nfs-gss.conf` gibi yeni bir dosya oluşturup, dosya içerisinde aşağıdaki satırları eklemeniz yeterlidir:

```
blacklist rpcsec_gss_krb5
```

Dosyayı kaydedip sisteminizi yeniden başlattıktan sonra problemli senaryonun düzelmiş olduğu görünecektir.

TI işlemcilerinde DSP kullanımı

Bu bölümde, TI (*Texas Instruments*) firması tarafından üretilen heterojen işlemciler üzerinde DSP modülünün kullanımına bakacağız.

Heterojen işlemciler, tek bir yonga üzerinde, genel amaçlı bir ARM işlemciyle birlikte bir DSP çekirdeğini de barındırmaktadırlar. Özellikle, video ve ses işleme uygulamalarında, yoğun tekrarlanan matematiksel işlemler DSP üzerinde çalıştırılarak performans artışı hedeflenmektedir.

DSP üzerinde kod çalıştırmak için birden fazla yöntem bulunmaktadır. Biz bu bölümde genel bir fikir vermeyi amaçladığımızdan TI firmasının, ARM Linux geliştiricileri için hazırladığı, *C6Run* projesine ait geliştirme araçlarını kullanacağız.

İncelemelerimizde, *Hawkboard* gömülü sistemini kullanacağız. Bölüm içerisinde sırasıyla, *C6Run* projesinin kurulumuna ve *Hawkboard* hedefli uygulamaların nasıl derlenip hedef sistemde çalıştırıldığına bakacağız.

C6Run

C6Run, TI tarafından, çift çekirdekli (ARM+DSP) heterojen işlemcileri için geliştirilmiş bir projedir. *ARM Linux* ve *C6000 DSP* çekirdeği içeren sistemler hedeflenmektedir.

Projenin 2011 yılı itibariyle geliştirilmesi sonlanmasına karşın, proje indirilebilmekte ve kullanılabilmektedir.

C6Run, *DSP* ile ilgili birçok detayı gizlediğinden ve burada *DSP* kullanımı hakkında genel bir fikir vermeyi amaçladığımızdan bu projeyi kullanacağız.

Bu bölümde ilk olarak, projeye ilişkin araçları sistemimize nasıl kuracağımıza, sonrasında *DSP* hedefli kodu nasıl üretebileceğimize bakacağız.

C6Run Kurulumu

Kurulum işlemine ilk olarak küçük bir başlangıç paketi indirerek başlayacağız. Projenin diğer bağımlılıklarını, bir tanesi hariç, bu paket üzerinden sağlayacağız. Ayrıca derlenmiş bir *kernel* dizinine de ihtiyacımız olacak. Başlangıç paketini aşağıdaki gibi indirebilirsiniz.

```
wget -c https://gforge.ti.com/gf/download/frsrelease/535/4556/%43%36%52%75%6e%5f%30%5f
%39%37%5f%30%33%5f%30%33%2e%74%61%72%2e%67%7a
```

> **Uyarı:** İndirdiğimiz paketin, **0.98** numaralı bir üst versiyonu bulunmasına veya **SVN** üzerinden son halinin indirilebilmesine karşın, sonrasında birçok probleme neden olmakta o yüzden **0.97** numaralı bu versiyonu kullanmanızı öneriyoruz.

Paketi açıp, *DSP* derleyicisi hariç diğer bağımlılıkları aşağıdaki gibi sisteminize yükleyip kurabilirsiniz. Bu amaçla ana dizin içindeki *Makefile* dosyasında *get_component* isimli bir hedef bulunmaktadır.

```
tar xf C6Run_0_98_03_03.tar.gz
cd C6Run_0_98_03_03
make get_components
```

DSP'ye ilişkin araçları ise sistemimize elle kurmalıyız. Bu araçlara ilişkin kurulum dosyasını aşağıdaki bağlantıdan indirebilirsiniz. **7.2.2** versiyonunu seçiniz.

```
http://software-dl.ti.com/codegen/non-esd/downloads/download.htm
```

> **Not:** Bu işlem öncesinde TI üyeliği gerektirmektedir, TI'ın sitesinden ücretsiz olarak üye olabilirsiniz.

İndirme işlemi sonucunda elde ettiğiniz *ti_cgt_c6000_7.2.2_setup_linux_x86.bin* kurulum dosyası ile *DSP*'ye ilişkin araçları *home* dizinin altında oluşturacağınız *toolchains/TI_CGT_C6000_7.2.2* isimli bir dizin altına kurabilirsiniz. Neden böyle bir dizin yolu seçtiğimize birazdan bakacağız.

Gerekli bağımlılıkları edindikten sonra, ana dizindeki *Rules.mak* ve *platforms/hawkboard/platform.mak* dosyalarındaki bazı değişkenere uygun değerleri geçirmeliyiz.

Rules.mak dosyasında, *DSP* ve *ARM* araçlarının öngörülen yol ve önek ifadeleri aşağıdaki gibidir.

```
CODEGEN_INSTALL_DIR     ?= $(HOME)/toolchains/TI_CGT_C6000_7.2.2
ARM_TOOLCHAIN_PATH      ?= $(HOME)/toolchains/arm-2009q1
ARM_TOOLCHAIN_PREFIX    ?= arm-none-linux-gnueabi-
```

DSP veya *ARM* derleyicinizi başka bir yere kurduysanız veya başka bir çapraz derleyici kullanıyorsanız, bu değişkenlere uygun değerleri vermelisiniz.

platforms/hawkboard/platform.mak dosyasında ise derlenmiş çekirdek dizinini göstermeliyiz, öngörülen değeri aşağıdaki gibidir.

```
LINUXKERNEL_INSTALL_DIR     ?= $(HOME)/workdir/hawkboard/kernel/
```

Kendi sisteminize uygun olarak bu değişkenleri değiştirebilirsiniz.

> **Not:** Board Recovery bölümünde, Hawkboard için çekirdeği nasıl derleyebileceğiniz bulunmaktadır.

Bu aşamadan sonra, projeyi *Hawkboard* hedefli konfigür edebilir ve gerekli araçları aşağıdaki gibi derleyebiliriz.

```
make hawkboard_config
make everything
```

Bu süreç sonunda sisteminizde gerekli kütüphaneler kurulmuş olmalıdır. Bu aşamadan sonra proje dizinindeki örnekleri derleyip, örneklerle beraber gerekli modülleri bir dizinde sakalayabiliriz. Öncesinde *C6Run* ana dizindeki *bin* dizinini *PATH* çevre değişkenine eklemeliyiz.

```
export PATH=$PATH:$(pwd)/bin
make examples
make INSTALL_DIR=/tmp/hawkdps install_programs
```

Hedef platform için derlenmiş örneklerle beraber, gerekli çekirdek modülleri ve betiklerler aşağıdaki gibi olacaktır.

```
# ls /tmp/hawkdps/
cmemk.ko  dsplinkk.ko  examples  loadmodules.sh  test  unloadmodules.sh
```

Son olarak, DSP modülünü kullanan kendi uygulamalarımızı nasıl derleyebileceğimize bakalım.

DSP Hedefli Uygulamaların Derlenmesi

C6Run araçları ile, DSP modülünü farklı şekilde kullanan, 2 tür uygulama oluşturmak mümkündür. Birinci durumda uygulamanın tamamı DSP modülünde çalışırken, ikinci durumda yalnız istenilen kritik bir kodu DSP tarafında çalıştırmak mümkündür. Her iki durumda da uygulama doğal bir ARM uygulaması olarak gözükmekte, DSP modülünün başlatılması ve program kodunun yüklenmesi içsel olarak yapılmaktadır. Tüm giriş/çıkış işlemleri ise DSP modülünden ARM tarafına yönlendirilmektedir.

Uygulamalarımızı derlemek için kullanacağımız önyüz araçları projenin ana dizini altındaki *bin* dizininde yer almaktadır. Kullanacağımız araçlar aşağıdaki gibidir:

- c6runapp-cc
- c6runlib-cc
- c6runlib-ar

Bash betiği olarak yazılmış bu önyüz araçları, arka planda TI C6000 derleyicisi *cl6x*'i, çapraz ARM derleyicisini, diğer gerekli araçları kullamakta ve bize *gcc* derleyicisine oldukça benzer bir arayüz sunmaktadır.

Şimdi sırasıyla, uygulamanın tümünün ya da yalnız bir kısmının DSP tarafında çalıştırılacağı uygulamaları nasıl derleyebileceğimize bakalım.

Tüm Uygulamanın DSP Hedefli Derlenmesi

Aşağıdaki örnek uygulamayı *measure.c* adıyla saklayıp derleyebilirsiniz.

```
#include <stdio.h>
#include <time.h>
#include <math.h>

#if defined(_TMS320C6X)
#elif defined(__GNUC__)
    #include <sys/time.h>
#endif

#define N 1000000

typedef unsigned long long timestamp_t;

int i;
timestamp_t t0, t1;
float secs;
double d;

static timestamp_t get_timestamp () {
#if defined(_TMS320C6X)
      return (timestamp_t) clock();
#elif defined(__GNUC__)
    struct timeval now;
    gettimeofday (&now, NULL);
    return  now.tv_usec + (timestamp_t)now.tv_sec * 1000000;
#endif
}

void bench() {
    t0 = get_timestamp();
    for (i = 0; i < N; ++i) {
        d = sin(i) * cos(i);
    }
    t1 = get_timestamp();
    secs = (t1 - t0) / 1000000.0L;
    printf("%d times sin(i) * cos(i)   sec: %f\n", N, secs);
}

int main() {
    bench();
    return 0;
}
```

```
c6runapp-cc -omeasure_dsp measure.c
```

Aynı uygulamayı ayrıca ARM hedefli derleyerek, *bench* fonskiyonunda geçen süreleri karşılaştırabilirsiniz. Kullandığımız çapraz derleyici için derleme işlemi aşağıdaki gibi olacaktır.

```
arm-none-linux-gnueabi-gcc -omeasure_arm measure.c -lm
```

> **Not:** ARM ve DSP modüllerinde sistem zamanının farklı şekilde alındığına ve kodda bu bölümlerin `_TMS320C6X` ve `__GNUC__` makrolarıyla birbirinden ayrıldığına dikkat ediniz.

Kritik Kodun DSP Hedefli Derlenmesi

DSP tarafında çalışacak kod, ilk önce C6000 obje koduna, ardından linklenebilecek ARM hedefli bir kütüphaneye dönüştürülmelidir. Sırasıyla bu işlemler için *c6runlib-cc* ve *c6runlib-ar* araçlarını kullanacağız.

Bir önceki örneğimizdeki *bench* fonksiyonunu bir kütüphane fonksiyonu olarak derleyelim. Aşağıdaki örnek kodları sırasıyla *libmeasure.c* ve *main.c* olarak kaydedebilirsiniz.

```
#include <stdio.h>
#include <time.h>
#include <math.h>

#if defined(_TMS320C6X)
#elif defined(__GNUC__)
    #include <sys/time.h>
#endif

#define N 1000000

typedef unsigned long long timestamp_t;

int i;
timestamp_t t0, t1;
float secs;
double d;

static timestamp_t get_timestamp () {
#if defined(_TMS320C6X)
      return (timestamp_t) clock();
#elif defined(__GNUC__)
    struct timeval now;
    gettimeofday (&now, NULL);
    return  now.tv_usec + (timestamp_t)now.tv_sec * 1000000;
#endif
}

void bench() {
    t0 = get_timestamp();
    for (i = 0; i < N; ++i) {
        d = sin(i) * cos(i);
    }
    t1 = get_timestamp();
    secs = (t1 - t0) / 1000000.0L;
    printf("%d times sin(i) * cos(i)   sec: %f\n", N, secs);
}
```

```
void bench();

int main() {
    bench();
    return 0;
}
```

bench kodunun bulunduğu kaynak kodu, aşağıdaki gibi bir statik kütüphaneye dönüştürdükten sonra uygulamamıza linkleyebiliriz.

```
c6runlib-cc -c -olibmeasure.o libmeasure.c
c6runlib-ar rcs libmeasure.a libmeasure.o
arm-none-linux-gnueabi-gcc -c main.c
arm-none-linux-gnueabi-gcc -omeasure_dsp main.o libmeasure.a
```

DSP Testi

Bu bölümde daha önce derlediğimiz örnek uygulamaları nasıl çalıştıracağımıza bakacağız.

Daha önce *C6Run* içindeki örnekleri derleyip bir dizinde saklamıştık. Bu dizinin içereğini tekrar hatırlayalım.

```
# ls /tmp/examples/
cmemk.ko  dsplinkk.ko  examples  loadmodules.sh  test  unloadmodules.sh
```

2 adet çekirdek modülünün, örnekleri içeren *examples* dizinin, ayrıca modülleri yüklemek ve kaldırmak için, *loadmodules.sh*, *unloadmodules.sh* isimli betiklerin bulunduğunu görmekteyiz.

C6Run modüllerinin belleğin neresini kullanacakları önem taşımaktadır. Modüllerin bellekte kullanacakları alanın başlangıç ve bitiş adresleri derleme zamanında değiştirilebileği gibi yükleme zamanında da belirtilebilir. Biz burada öngörülen değerleri kullanacağız. *loadmodules.h* dosyasında bu adresler aşağıdaki gibi gösterilmektedir.

```
DSP_REGION_START_ADDR="0xC2000000"
DSP_REGION_END_ADDR="0xC4000000"
```

Hawkboard için çekirdek kodu ise 0xC000000 adresinden başlamaktadır, çekirdeğin DSP modülleri için ayrılmış alana taşmasını önlemek için çekirdeği kullandığı alanı, modüllerin başlangıç adresi olan, 0xC2000000 adresinde sonlandırmalıyız.

Bu durumda çekirdek 0xC000000 ile 0xC2000000 arasında bulunmalıdır, çekirdek için ayrılan alan 0x2000000 yani 32M olmalıdır. Çekirdeğin bu bölgeyi kullanmasını sağlamak için çekirdeğe *mem=32M* boot argümanı geçirilmelidir. Bu işlem U-Boot üzerinden aşağıdaki gibi yapılabilir.

```
setenv bootargs "console=ttyS2,115200 mem=32M init=/bin/sh
root=/dev/mtdblock4 rootwait rootfstype=jffs2"
```

loadmodules.sh ile gerekli modülleri yükledikten sonra, proje içinden çıkan ya da kendi derlediğiniz test uygulamalarınızı çalıştırabilirsiniz.

ARM ve DSP hedefli derlenen FFT uygulamaları için elde ettiğimiz sonuçlar aşağıdaki gibidir.

```
# ./cfft_arm
N=16,nTimes=100: 0.007027 s
N=32,nTimes=100: 0.017506 s
N=64,nTimes=100: 0.04224 s
N=128,nTimes=100: 0.098568 s
N=256,nTimes=100: 0.226512 s
N=512,nTimes=100: 0.510898 s
N=1024,nTimes=100: 1.15786 s
N=2048,nTimes=100: 2.61276 s
N=4096,nTimes=100: 5.8848 s
N=8192,nTimes=100: 12.8365 s
N=16384,nTimes=100: 27.6982 s

# ./cfft_dsp
N=16,nTimes=100: 0.056423 s
N=32,nTimes=100: 0.055537 s
N=64,nTimes=100: 0.058968 s
N=128,nTimes=100: 0.059059 s
N=256,nTimes=100: 0.06587 s
N=512,nTimes=100: 0.080403 s
N=1024,nTimes=100: 0.107609 s
N=2048,nTimes=100: 0.168634 s
N=4096,nTimes=100: 0.302576 s
N=8192,nTimes=100: 0.619787 s
N=16384,nTimes=100: 1.33111 s
```

Kullandığınız DSP modülünün sabit (*fixed point*) ya da kayan noktalı (*floating point*) oluşuna göre bu sonuçlar değişecektir. Ayrıca ARM tarafından DSP modülüne yapılan çağrılar maliyetli olduğundan mümkün olduğunca az çağrı yaparak, çağrı başına daha fazla işlem yapmak daha uygun olacaktır.

Ubuntu Sanal Makine Performansı

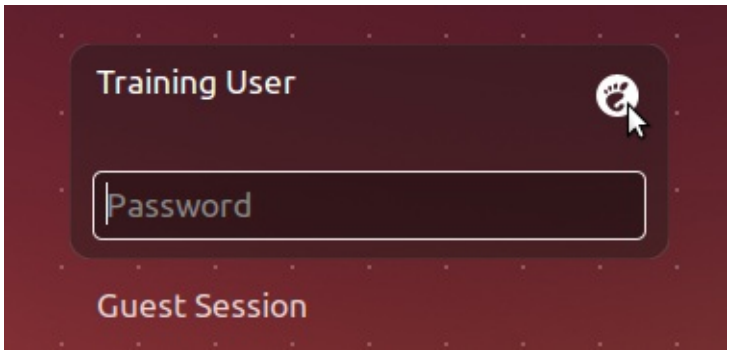
Özellikle düşük performanslı **host** sistemlerde Gömülü Linux eğitimlerimizde de kullandığımız Ubuntu sanal makinesinin Virtualbox veya Vmware altındaki performansını artırmak için aşağıdaki işlemleri yapabilirsiniz.

Öncelikle sanal makinemizde `gnome-session-fallback` paketi kurulu değil ise bunu kurmalısınız:

```
$ sudo apt-get install gnome-session-fallback
```

Ardından sistemden çıkış yapınız.

Yukarıdaki paket kurulduktan sonra, giriş ekranında parola yazılan bloğun sağ üst köşesinde Ubuntu logosunu göreceksiniz:



Bu bölüme tıklarak açılan seçenekler arasından **GNOME Flashback (Metacity)** değerini seçiniz.

Parolanızı yazıp giriş yaptığınızda GUI performansının daha iyi olacağı eski nesil bir arayüz ile sistem açılacaktır.