

# MAKE İLE PROJE YÖNETİMİ VE MAKEFILE YAZIMI

Gürhan Yerlikaya

14.02.06

[V:1]

**Özet :** Bu makalede GNU make programının çalışması ve makefile yazım kuralları ele alınmıştır. make programının C programlarını derlemekte kullanış biçimlerinden bahsedilmiştir. Açıklamalarda Unix/Linux sistemlerindeki make programı esas alınmıştır.

## 1. make PROGRAMI

*make*, bir çok değişik çıktı üretebilen genel bir araçtır. Oluşturulan bir kural dosyası aracılığıyla, belirtilen sistem komutlarını ardarda çalıştırır. Bu vesile ile, çok karmaşık işleri, komut satırından verilen basit bir *make* komutu ile kolayca yapabilirsiniz.

Temel olarak *make* programı kural dosyası üzerinde bazı kontroller yaparak işlemektedir. *make*, kural dosyası içerisinde belirtilen hedef dosya ile kaynak dosyanın zaman damgası (*time stamp*) ilişkisine göre hareket eder ve bir takım komutların çalıştırılıp çalıştırılmayacağına karar verir.

Başka bir deyişle, her bir kaynak dosya bir hedef dosyaya ihtiyaç duyar ve kaynak dosya, hedef dosyadan daha yeniyse kural dosyasında belirtilen komutlar çalıştırılır. Bu komutlar hedef dosyanın yenilenmesini sağlar.

Aslen *make* daha genel bir araç olarak tasarlanmıştır. *make* programı ile birlikte kullanılan kural dosyası ile C/C++ programlarını derlemenin yanısıra, *HTML* kodlarındaki değişikliklerin takibi, *NIS* (*Network Information Service – Ağ Bilgi Hizmeti*) de kullanıcı bilgilerini işleme, *TEX / LATEX* dökümanların derlenmesi, program kurulumu gibi işler de yapılabilmektedir.

Günümüzde açık kaynak kodlu yazılımlar, kaynak dosyalarının yanında derleme işlemi için kullanılacak *makefile* dosyası ile birlikte gelmektedir. Herhangi bir *Linux* dağıtımını ya da açık kaynak kodlu bir yazılımı *make* programı aracılığıyla kendiniz derleyebilirsiniz. İnternet aracılığı ile kolayca elde edebileceğiniz bazı *betik* (*script*) dosyaları ve *make* programı ile birlikte kendi *Linux* dağıtımınızı bile oluşturabilmeniz mümkündür.

CSD İşletim Sistemi geliştirme projesinde de derleme işleri *gcc* (*GNU Compiler Collection*) derleyicisi ve *make* yardımcı programı aracılığı ile yapılmaktadır.

## 2. *make* PROGRAMININ KULLANIMI

Yorumlayıcı mantığıyla çalışan programlama dillerinin tersine, derleme mantığıyla çalışan C ya da C++ gibi programlama dillerinde kaynak dosyalar kabaca iki aşamadan geçtikten sonra çalıştırılabilir dosya haline getirilirler.

İşletim sistemi geliştirme projesi gibi taşınabilirliğin çok önemli olduğu projelerde *IDE* (*Integrated Development Environment – bütünleşik geliştirme ortamı*) derleyicileri kullanma imkanı olmadığından, derleme işlemi konsol ekrandan çalışan derleyiciler ile mümkün olabilmektedir. Böyle projelerde dosya sayısı fazla olduğundan derleme süreleri çok uzun ve konsol ekrandan yazımları da zor olmaktadır. Bu nedenle *make* gibi yardımcı programlara gerek duyulmuştur.

Eğer C programcısıysanız ve birden fazla kaynak dosyadan oluşan bir projeniz varsa bu dosyaları elle derlemek ve bağlamak çok zaman alacaktır. Yaptığınız her değişiklikte *gcc* derleyicisi aracılığıyla, elle bir çok komut ve dosya adı girmeniz gerekmektedir.

İşte *make* programı ve kural dosyası ile bu işlemleri otomatik olarak yalnızca konsol ekranına :

```
# make
```

komutunu yazarak yapabilirsiniz.

Kural dosyası içerisindeki etiket ya da etiketler altındaki tanımlamalar ile *make* programı bizim için gerekli derleme ve bağlama işlemlerini yapar. Eğer siz kaynak dosyalar içerisinde bir değişiklik yaptıysanız *make* programı ilk önce değişiklik yaptığınız dosyayı derler, sonra da diğer obje dosyalar ile bağlayarak çalıştırılabilir dosyayı oluşturur. Programcının sorumlu olduğu, kural dosyası içerisindeki bağımlılıklardır. *make*, ön tanımlı olarak adı *makefile* ya da *Makefile* olan bir dosyayı arar.

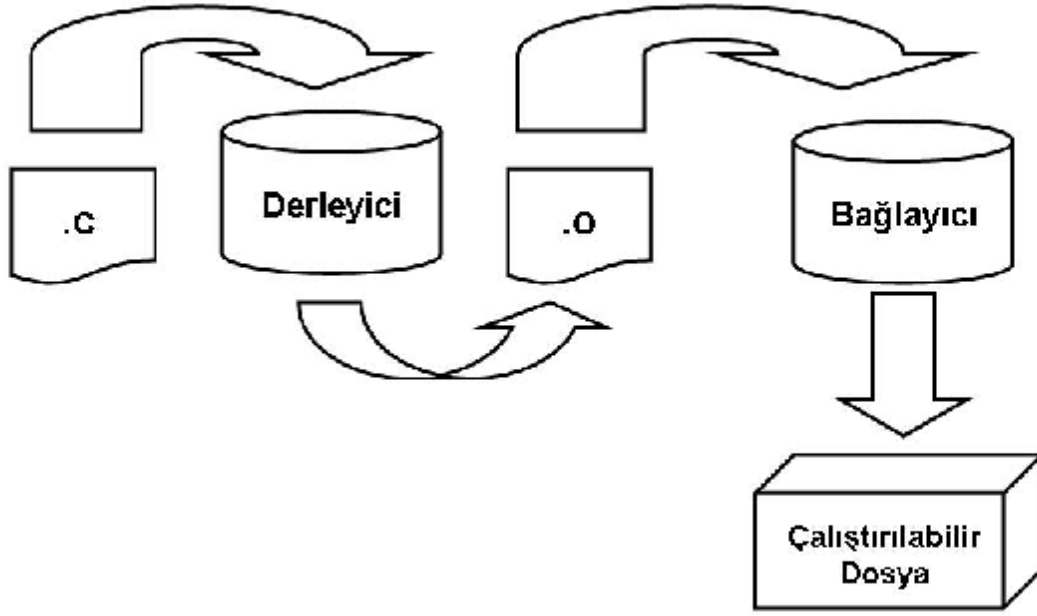
Kısaca *make* birden çok kaynak dosyadan oluşan programlarda sadece gerekli derleme işlerinin yapılmasını sağlayarak derleme sürecini hızlandırmaktadır.

## 3. DERLEYİCİLERİN ÇALIŞMA MANTIĞI VE *Makefile* DOSYASI

Bir program, birden fazla kaynak dosyadan oluşabilir. Bu kaynak dosyaların proje dosyasına eklenmesi ile, bir dosyadan include edilmesi, tamamen farklı anlamlara gelen kavramlardır.

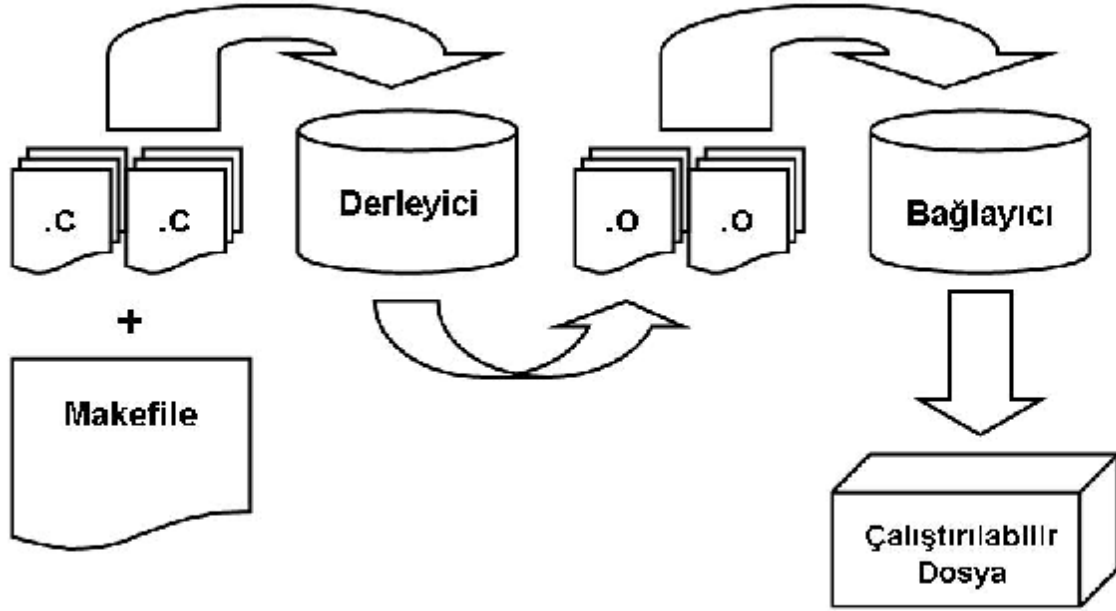
Projeye eklenmiş dosyalara modül denir. Ve modüller bağımsız olarak derlenmektedir. Sonra hep birlikte bağlanırlar. Büyük projeler tek modül halinde yazılamazlar. Bu nedenle birden fazla modül ile çalışılmaktadır.

Basitçe bir *.c* dosyasından çalıştırılabilir dosya oluşturulması işlemi aşağıdaki gibi olmaktadır :



Build ya da *make* işlemi, yalnızca değişen dosyaların yeniden derlenmesi ve hep birlikte bağlanması anlamına gelir. IDE' li derleyicilerdeki *Build All* ya da *Make All* seçenekleri kayıtsız şartsız, tüm dosyaların yeniden derlenmesi anlamına gelir.

Modern derleyicilerin bizim için yaptığı, her proje için bir *makefile* oluşturmaktır. Böylelikle biz her yeni dosya eklediğimizde *makefile*' in içeriğini açıp yeni tanımlamalar girmek zorunda kalmayız. Tabi IDE' li bir derleyici kullanamayacağımız durumlar da vardır. İşletim sistemi yazımı buna güzel bir örnektir elbette. Aşağıda basit bir C projesinin *makefile* ile birlikte nasıl çalıştırılabilir dosya haline getirildiği gösterilmiştir :



*Makefile* dosyası içerisine yazacağımız tanımlamalar sayesinde *make* programı hangi dosyaların derlenip bağlanacağına karar vermektedir.

*IDE* sisteminin olmadığı durumlarda, değişen dosyaların tespit edilerek, derlenmesi ve bağlanması, *make* programı tarafından yapılabilmektedir. Programcı bir *makefile* dosyası hazırlar ve sonra *make* programını çalıştırır.

*IDE*’li sistemlerde menülerdeki *compile* seçeneği, o anda editörde bulunan kaynak dosyayı derlemektedir. *Build* ya da *make* seçenekleri, aynı anlama gelir ve yalnızca değişen dosyalar yeniden derlenir. Sonra, bağlama işlemi yapılarak çalıştırılabilir dosya oluşturulur. *Build All* gibi bir seçenek projedeki tüm kaynak dosyaların koşulsuz derleneceği ve bağlanarak çalıştırılabilir dosya oluşturulacağı anlamına gelmektedir. Menülerdeki *run* gibi komutlar *Build* ya da *make* işlemi yaptıktan sonra üretilen çalıştırılabilir dosyayı çalıştırmaktadır.

*Makefile*, *make* programının kullanacağı kuralları içeren text dosyasıdır. *Makefile* dosyalarının oluşturulma biçimi ve genel sentaksı *Unix/Linux*, *DOS/Windows* sistemlerinde birbirine benzemektedir. Fakat bazı farklar bulunmaktadır. Programcı hangi sistemde ve hangi derleyici ailesiyle çalışıyorsa onun *makefile* oluşturma sentaksını dikkate almalıdır.

## 4. BASİT BİR *Makefile* DOSYASI OLUŞTURMAK

*Makefile* dosyaları kural (*rule*) denilen cümleciklerden oluşturulur. Bir kuralın genel biçimi aşağıdaki gibidir :

```
Hedef : Bağımlılıklar
<tab>Komut
...
...
```

Hedef (target) etiketi, make programı tarafından yapılacak işlerin başladığı satırı belirtir. Eğer bağımlılıklar (dependencies) hedeften daha yeni ise (burada dosyanın zaman damgası özelliği karşılaştırılır) komutların (command) bulunduğu satırdaki işler yapılır. Her komut bir tab içeriden başlamalıdır. Hedef ile bağımlılıklar “:” karakteri ile birbirinden ayrılır. Birden fazla hedefiniz olabilir.

```
Hedef1 Hedef2 : Bağımlılık1 Bağımlılık2
<tab>Komut
...
...
```

Burada Bağımlılık2 dosyasının zaman damgası Bağımlılık1 dosyasının zaman damgasında daha yeniye belirtilen komut çalıştırılır.

Programınızın daha farklı olarak derlenmesini istediğinizde, *makefile* dosyanız içerisinde oluşturduğunuz diğer hedef etiketlerini make programına parametre olarak verebilirsiniz.

```
# make clean
```

Burada *make* programı *makefile* dosyası içerisindeki `clean :` ile başlayan satırı arayacak, ve oradaki işleri yapacaktır. Küçük bir C projesinin *makefile* dosyası aşağıda verilmiştir :

```
#
# Makefile Test Project
# Copyright (c) 2006 by Gürhan Yerlikaya <han@csystem.org>
#

proje : main.o io.o data.o /usr/lib/libc.a
    cc -o proje main.o io.o data.o /usr/lib/libc.a

main.o : data.h io.h main.c
    cc -c main.c

io.o : io.h io.c
    cc -c io.c
```

```
data.o : data.h data.c
cc -c data.c
```

```
clean :
/usr/lib/rm *.o
```

*make* programı # ile başlayan satırları dikkate almaz. Buraya açıklama yazabiliriz. C99'daki // ile benzerdir. Aynı şekilde, sadece tek satır için kullanılır. Eğer komutlar çok uzunsa / ile alt satırdan devam edebiliriz. Başında @ karakteri bulunmadığı sürece komutlar da standart çıktı birimine gönderilir.

Eğer standart çıktı birimine istediğimiz yerlerde bazı uyarı mesajları göndermek istiyorsak *echo* komutunu kullanabiliriz. Yukarıdaki örneği *echo* komutunu kullanarak yazarsak :

```
proje : main.o io.o data.o /usr/lib/libc.a
cc -o proje main.o io.o data.o /usr/lib/libc.a
@echo == Derleme islemi basari ile tamamlandi!.. ==
```

```
main.o : data.h io.h main.c
cc -c main.c
@echo main.o dosyasi basari ile olusturulmustur.
```

```
io.o : io.h io.c
cc -c io.c
@echo main.o dosyasi basari ile olusturulmustur.
```

```
data.o : data.h data.c
cc -c data.c
@echo main.o dosyasi basari ile olusturulmustur.
```

```
clean :
/usr/lib/rm *.o
@echo Obje dosyalar basari ile temizlenmistir.
```

şeklinde olur. Böylelikle yaptığımız hataları daha rahat takip edebiliriz. Eğer @ karakterini kullanmasaydık, örneğin :

```
#make clean
```

komutu verildiğinde ekranda :

```
Obje dosyalar basari ile temizlenmistir.
```

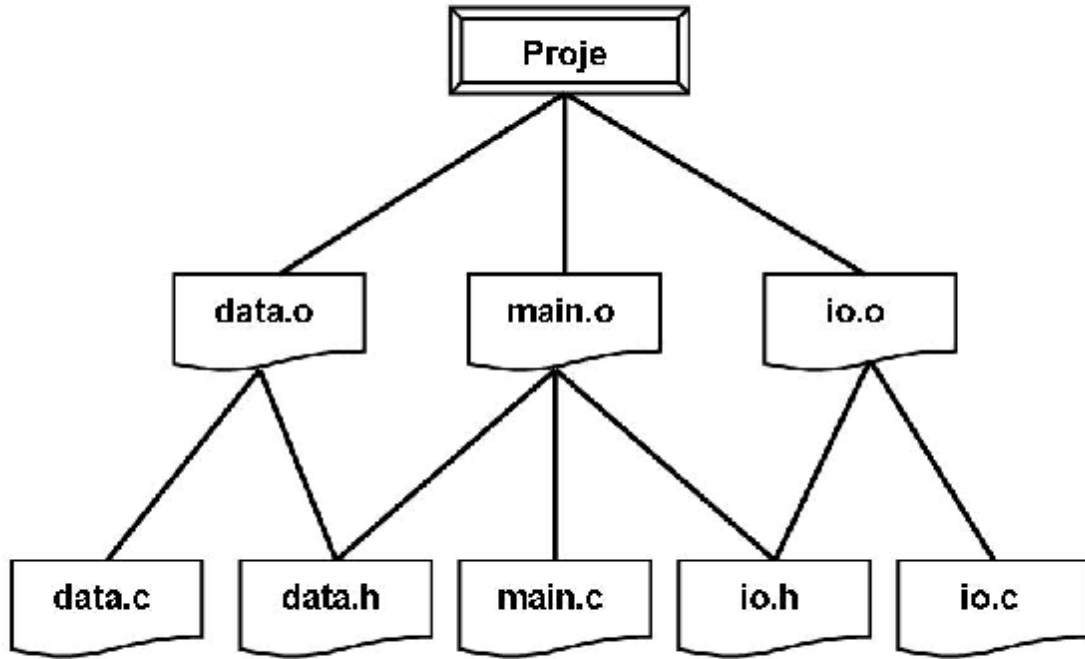
yazısı yerine :

```
echo Obje dosyalar basari ile temizlenmistir.
```

yazısını görecektik.

Yapacağımız tanımlamalarda, en son bakılmasını istediğimiz bağımlılıkları en başa yazmalıyız. Çünkü *make* programı işleri sondan başa doğru yapmaktadır. Bu nedenle bir hedef, kendisine bağlı hedeflerin tümünden daha sonra tanımlanmış olmalıdır.

Yukarıdaki *Makefile* içerisindeki bağımlılıkları grafikte ifade edersek :



*data.c* kaynak dosyasının içerisinde bir değişiklik yaptıysak, *data.o* obje dosyasının yeniden oluşturulması için ona bağlı dosyalar yeniden derlenir. Eğer *data.h* dosyasında bir değişiklik yaptıysak *data.o* ve *main.o* obje dosyalarına bağlı dosyalar yeniden derlenir.

## 5. MAKROLAR LA ÇALIŞMAK

*Makefile* dosyalarında bazı uzun yazımları engellemek için makrolar kullanılmaktadır. Makro bir çeşit *C*’deki sembolik sabit gibidir. Makroların temel söz dizimi şöyledir :

isim : değer\ler

Kullanımı ise :

$\$\{ \text{MAKRO\_ISMI} \}$

ya da

`$(MAKRO_ISMI)`

şeklindedir. Böylece yazımda gereksiz yinelenmeleri engellemiş oluruz. Bir makro birden fazla kez kullanılırsa en son kullanıldığı yerdeki hali geçerli olacaktır.

Bir önceki başlıkta verdiğimiz örnekte makro kullansaydık dosyamız aşağıdaki gibi olacaktı:

```
LIBCDIR = /usr/lib/
COMPILER = cc
OBJS = main.o io.o data.o
EXE = proje

${EXE} : ${OBJS} ${LIBCDIR}libc.a
    ${COMPILER} -o ${EXE} ${OBJS} ${LIBCDIR}libc.a
    @echo == Derleme islemi basari ile tamamlandi!.. ==

main.o : main.c data.h io.h
    ${COMPILER} -c main.c
    @echo main.o dosyasi basari ile olusturulmustur.

io.o : io.h io.c
    ${COMPILER} -c io.c
    @echo io.o dosyasi basari ile olusturulmustur.

data.o : data.h data.c
    ${COMPILER} -c data.c
    @echo data.o dosyasi basari ile olusturulmustur.

clean :
    ${LIBCDIR}rm *.o
    @echo Obje dosyalar basari ile temizlenmistir.
```

## 5.1 Yerleşik Makrolar

*make* programı bir kaç tane özel yerleşik makroyu kendinden sunmaktadır. Bu makrolar sistemden sisteme değişebilmektedir. Biz bu makalede *Unix/Linux* sistemlerinde olanları inceleyeceğiz. Bu yerleşik makroların tek karakterli olanlarını kullanırken, diğerlerinden farklı olarak, parantez kullanmamıza gerek yoktur. Örneğin yerleşik makrolardan biri olan `@` makrosunu kulanırken :

`$(@)`

yerine

`$@`

şeklinde kullanabiliriz. Fakat bizim oluşturduğumuz makro adları için tek karakterden çok betimleyici isimler kullanmamız salık verilmiştir. Böylelikle yazdığımız *makefile* dosyaları daha okunabilir olacaktır. Bu tip makroların içerisindeki değeri değiştirmek mümkün değildir.  $\$%$ ,  $\$?$ ,  $\$*$ ,  $\$<$  makroları diğer yerleşik makrolardır.

Ön tanımlı olan değeri değiştirilebilen makrolar da vardır. *VPATH*, *CC*, *LD* ve *SHELL* makroları bu tür makrolardandır. *CC* makrosu sistemdeki öntanımlı derleyiciyle, *LD* öntanımlı bağlayıcıyla ilişkilendirilmiştir. *SHELL*, bulunulan sistemdeki öntanımlı kabuk program ile ilişkilendirilmiştir. *VPATH* makrosu sistemdeki *PATH* değişkenine birden fazla değer girmemizi sağlamaktadır.

*VPATH* makrosu dizin ağacı çok karışık olan, işletim sistemi geliştirme projesi gibi projelerde çok kullanışlıdır. *Linux* ve *CSD* işletim sistemlerinin *makefile* dosyalarında *VPATH* makrosu çok büyük fayda sağlamaktadır. Yerleşik makroların ayrıntılarını *GNU Make Utility* kılavuzundan inceleyebilirsiniz.

## 6. YARARLI MAKE ARGÜMANLARI

- **d** : Hata ayıklama kipinde kullanmamızı sağlar.

- **f** : Daha önce de söylediğimiz gibi *make* programı parametresiz olarak çalıştırılırsa *makefile* dosyası olarak sırasıyla *Makefile* ve *makefile* isimli dosyaları arar. Kendi belirlediğimiz bir dosyanın işleme +sokulabilmesi için *-f* seçeneği kullanılabilir. Kullanımı :

```
# make -f dosya_ismi
```

şeklinde. Standart *makefile* dosyası adınının kullanılmadığı durumlarda genellikle dosya uzantısı olarak *.mk* kullanmak gelenek haline gelmiştir.

- **p** : Yapılan her işi adım adım standart çıktı birimine yönlendirmek için kullanılır.

- **k** : *make* programı yürütülürken bir hata oluştuğunda, oluşan hatanın bulunduğu yerdeki işlemi devre dışı bırakarak diğer işlemleri yapmaya devam eder.

- **q** : Komutları çalıştırmadan hedeflerin güncelliğini sorgulamak için kullanılır. *make* programı hedeflerin tümü güncelse sıfır değerine geri döner. Eğer biri güncel değilse sıfır dışı bir değer döndürür.

- **t** : Bu seçenek Unix/Linux sistemlerindeki *touch* komutuyla alakalıdır. *make* programını *touch* kipinde çalıştırır.

- **i** : *make* programı çalışırken oluşan bir hatadan dolayı durmamasını sağlar. *make* hataları görmezden gelerek sonuna kadar devam eder.

- **r** : *make* programının önceden tanımlanmış içsel kurallarını ve makrolarını devre dışı bırakarak yalnızca kullanıcı tarafından tanımlanmış kuralları dikkate alır.

## 7. YARDIMCI *make* PROGRAMLARI

**7.1 makedepend** : İşletim sistemi yazımı gibi büyük ölçekli projelerde dosyalar arasındaki bağımlılığın bulunması çok zaman alan ve zahmetli bir iştir. *makedepend* yardımcı programı bizim için kaynak dosyalarımız arasındaki bağımlılık listesini çıkartır.

**7.2 mkmf** : Proje içerisindeki dosyaların bağımlılıklarını araştırıp daha önceden belirlenmiş bir şablona göre bizim için bir *makefile* oluşturur. *mkmf* adı, *make makefile*’ ın kısaltmasıdır.

## 8. ALTERNATİF *make* PROGRAMLARI

Bir çok işi çok kısa zamanda, tek bir komut yardımıyla yapmamızı sağlasa da *make* bazı yönlerden hala çok yetersizdir. Örneğin *make* içerisinde *if* benzeri koşul değişimleri bulunmamaktadır. Bu nedenle standart *make* uygulamasına alternatif olarak bazı *make* programları geliştirilmiştir. Aşağıda bir kaç tanesi incelenmiştir.

**8.1 nmake** : *nmake*, AT&T’ nin geliştirdiği *make* için en gelişmiş alternatif programlardan birisidir. Büyük ölçekli C projeleri için çok gelişmiş özellikler sunmaktadır. *Microsoft*’ un da *nmake* isimli bir *make* sürümü bulunmaktadır. Fakat, teknik özellikler bakımından bu *make* programı AT&T’ nin *nmake*’ ine göre çok zayıf kalmaktadır.

**8.2 GNU make** : GNU’ nun geliştirdiği bu *make* sürümü sayesinde *if* gibi koşul değişimlerini *makefile* dosyasında kullanabiliyoruz. Ayrıca *GNU make* gelişmiş makro özellikleri, ek matn işleme özellikleri ve *:=* operatörünü de kullanabilmekteyiz.

**8.3 imake** : *imake*, *make* programı içerisinde daha rahat çalışabilememiz için bize bir ön işlemci mekanizması sunmaktadır. Oluşturduğu *makefile* dosyalarını daha önce belirlenmiş bir şablona göre oluşturmaktadır. *imake* büyük ölçekli projeler için kullanışlı bir araçtır.

## Kaynaklar

1. Managing projects with Make – Andrew Oran & Steve Talbott, O’Reilly, 1993
2. GNU Make Utility Manual -  
[http://www.gnu.org/software/make/manual/html\\_chapter/make.html](http://www.gnu.org/software/make/manual/html_chapter/make.html)