Make-Einführung, Tipps und Tricks

Version 1.13 — 17.12.2014

OSTC Open Source Training and Consulting GmbH
Thomas Birnthaler
E-Mail: tb@ostc.de
Web: www.ostc.de

Inhaltsverzeichnis

1 Einführung

Mit dem Programm *make* (*nmake* = „new“ make unter MSDOS[1]) und einer zugehörigen **Makedatei** (Standardname ist zuerst `makefile`, dann `Makefile`) kann der zur Erstellung von Dateien (z.B. Programmen, aber auch großen Dokumenten wie z.B. Büchern) notwendige Kommandoablauf (z.B. Compilier- und Linkablauf) automatisiert werden. Weiterhin kann damit erreicht werden, dass nach Änderungen an einigen wenigen Stellen nur die geänderten und alle davon abhängigen Dateien (Programmteile) neu verarbeitet (übersetzt) werden. Der Verwaltungsaufwand sowie die Verarbeitungszeiten (Compilations- und Linkzeiten) bleiben dadurch minimal.

2 Bestandteile

Eine **Makedatei** besteht aus:

- Kommentaren
- Regeln
- Aktionen
- Makrodefinitionen
- Leerzeilen
- Fortsetzungszeilen

2.1 Kommentare

Kommentare können an beliebiger Stelle durch ‘`#`’ eingeleitet werden, der danach folgende Text bis zum Zeilenende wird von *make* ignoriert:

```
# Dies ist ein Kommentar
    gcc a.c      # Ein weiterer Kommentar
```

2.2 Regeln + Aktionen

Eine **Abhängigkeitsregel** (**spezielle Regel**) beschreibt in einer Zeile, welche **Zieldateien** `TARGETS` von welchen **Quelldateien** `SOURCES` abhängen, die Ziel- und die Quelldateien werden dabei durch einen **Doppelpunkt** **:** getrennt (beliebig viele oder auch kein Leerzeichen um den Doppelpunkt). Auf eine Abhängigkeitsregel folgen, durch einen **Tabulator** am Zeilenanfang eingerückt (*daran werden sie erkannt!*), beliebig viele **Aktionen** (bis zur nächsten Regel oder Leerzeile):

[1] Von Microsoft wurde zunächst zusammen mit dem MSC-Compiler ein *make* ausgeliefert, das nicht die üblichen Eigenschaften besaß. Später wurde ein dem Standard besser entsprechendes *make* mitgeliefert, das (leider) den Namen *nmake* bekam, um das alte *make* gegebenenfalls noch parallel verwenden zu können.

```
TARGETS: SOURCES
    ACTION     # Tabulator am Zeilenanfang nicht vergessen!
    ...
```

2.2.1 Pseudoziele

Ziele müssen keine echten Dateien, sondern können auch **Pseudoziele** sein. Diese Ziele werden zur Automatisierung von Aufgaben eingesetzt, die im Umfeld des „Programmbauens“ liegen (zum Beispiel zum Installieren, Deinstallieren, Paket bauen). Ein Pseudoziel erzeugt also **keine Dateien**, sondern führt einfach eine Reihe von Kommandos aus (da das Ziel grundsätzlich fehlt). Beispiel:

```
all: ...
    ACTION     # Tabulator am Zeilenanfang nicht vergessen!

clean: ...
    ACTION     # Tabulator am Zeilenanfang nicht vergessen!

new: clean all

install:
    ACTION     # Tabulator am Zeilenanfang nicht vergessen!
```

2.2.2 Suffixregeln

Eine **Suffixregel (allgemeine Regel)** besagt, dass Dateien eines bestimmten Typs von Dateien eines anderen Typs abhängen. Der **Typ einer Datei** wird dabei durch ihren Suffix (auch Extension genannt) festgelegt (`.c`, `.o`, ...). Auf eine Suffixregel folgen, durch einem Tabulator am Zeilenanfang eingerückt (*daran werden sie erkannt!*), ebenfalls beliebig viele **Aktionen** (bis zur nächsten Regel oder Leerzeile). Eine Suffixregel zum Erstellen von Objektdateien (Suffix `.o`) aus C-Quellcodedateien (Suffix `.c`) kann z.B. folgendermaßen lauten:

```
.c.o:
    gcc -Wall -O2 $<     # Tabulator am Zeilenanfang nicht vergessen!
```

Achtung: Die Reihenfolge von Ziel und Quelle in Suffixregeln `.QUELLE.ZIEL` ist **umgekehrt** zur Reihenfolge in Speziellen Regeln: (`ZIEL: QUELLE`).

2.2.3 Musterregeln

Einige Make-Versionen (z.B. GNU-Make) bieten **Musterregeln** als mächtige Alternative zu Suffixregeln. Das **Muster** `%` im Ziel steht dabei für eine beliebige Zeichenfolge, die dann in den Quellen eingesetzt wird.

Obige Suffixregel lässt sich per Musterregel so schreiben (Reihenfolge von Quelle und Ziel umgekehrt!):

```
%.o : %.c
    gcc -Wall -O2 $<     # Tabulator am Zeilenanfang nicht vergessen!
```

In Musterregeln können leicht **zusätzliche Abhängigkeiten** untergebracht werden (hier von `main.h`):

```
%.o : %.c main.h
    gcc -Wall -O2 $<     # Tabulator am Zeilenanfang nicht vergessen!
```

Musterregeln müssen nicht unbedingt Präfixe oder Suffixe verwenden, d.h. in folgender Musterregel steht `%` für eine beliebige Zeichenkette und passt so **auf alle Ziele**:

```
% :: RCS/%,v
    $(CO) $(COFLAGS) $<  # Tabulator am Zeilenanfang nicht vergessen!
```

Der doppelte Doppelpunkt `::` weist Make an, fehlende Quellen nicht neu zu bauen. So muss Make hier nicht bei jeder Datei prüfen, ob es einen Weg gibt, ihre RCS-Datei zu erzeugen.

2.2.4 Standard(Pseudo)ziele

Folgende **Standard(Pseudo)ziele** sollten in jedem Makefile vorhanden sein und die beschriebenen Aktionen auslösen:

Ziel	Bedeutung
`all`	Erstes Ziel zum Erzeugen aller Programme und Dateien
`clean`	Löschen aller Zwischen- und Enddateien
`new`	Neuerstellen aller Programm und Dateien (`clean` + `all`)
`depend`	Generieren und Eintragen der Abhängigkeiten der C-Quellcodedateien von den Headerdateien in das Makefile
`install`	Installation der erzeugten Programme und Dateien
`uninstall`	Deinstallation der erzeugten Programme und Dateien
`backup`	Sichern der seit der letzten Sicherung geänderten Quelldateien
`save`	Sichern aller Quelldateien
`dist`	Ein Quelltextpaket erzeugen (Distribution)
`check`	Prüfungen durchführen
`test`	Testen aller Programme
`usage`	Usage-Meldung ausgeben

2.3 Makros

Innerhalb eines Makefiles können **Makros** definiert und verwendet werden (analog zu C). Sie dienen vor allem zur Erhöhung der Übersichtlichkeit und Portierbarkeit. Die **Definition eines Makros** erfolgt durch:

```
MACRO=Beliebiger Text
```

Der **Text bis zum Zeilenende** wird dabei dem Makro `MACRO` zugewiesen (Leerzeichen *vor* dem Zeichen = werden ignoriert, solche danach und vor dem Zeilenende *nicht*). Die **Verwendung eines Makros** erfolgt durch Einklammern des Namens und Voranstellen eines **Dollarzeichens** (einbuchstabige Makros müssen nicht unbedingt eingeklammert werden):

```
$(MACRO)
```

Anstelle des Ausdrucks `$(MACRO)` wird dann der ihm zugewiesene Text eingesetzt. Im folgenden Beispiel werden die Makros `CC`, `CFLAGS`, `MAIN` und `OBJS` definiert und anschließend in einer Abhängigkeitsregel und den zugehörigen Aktionen verwendet:

```
CC     = gcc
CFLAGS = -Wall -O
MAIN   = main
OBJS   = main.o a.o b.o c.o

$(MAIN): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS)     # Tabulator am Zeilenanfang nicht vergessen!
```

2.3.1 Makroquellen

Die Werte von Makros können aus verschiedenen Quellen stammen, die sich in ihrem Vorrang unterscheiden (in absteigender Reihenfolge):

Vorrang	Quelle	Befehl
1	Als Aufrufargument	`make VAR="Wert" ...`
2	Im Makefile gesetzt	`VAR = Wert ...`
3	Temporäre Umgebungsvariable	`VAR="Wert" make ...`
4	(Normale) Umgebungsvariable	`export VAR="Wert"; make ...`

Bei Angabe des Schalters **--e** (envvar) überschreiben die Umgebungsvariablen (normal und temporär) die im Makefile gesetzten Variablen.

Wird ein Makro mehrfach über die gleiche Quelle definiert, so gilt die **letzte Definition** und es erfolgt keine Fehlermeldung.

2.3.2 Standardmakros

Folgende **Standardmakros** sollten in Makefiles anstelle der echten Kommandos verwendet werden, um sie portabel zu halten:

Makro	UNIX	MSDOS	Bedeutung
CO	*co*		Checkout-Befehl (RCS)
CC	*gcc*	*cl*	C-Compiler
LD	*ld*	*link*	Linker
AR	*ar*	*lib*	Bibliotheksverwalter
AS	*as*	*masm*	Assembler
LEX	*flex*	*lex*	Scannergenerator
YACC	*bison*	*yacc*	Parsergenerator
MAKE	*make*	*nmake*	*make* selbst (für rekursiven Aufruf)
COFLAGS			Flags für Checkout-Befehl
CFLAGS			Flags für C-Compiler
LDFLAGS			Flags für Linker
ARFLAGS			Flags für Bibliotheksverwalter
ASFLAGS			Flags für Assembler
LFLAGS			Flags für Scannergenerator
YFLAGS			Flags für Parsergenerator
MAKEFLAGS			Flags für *make* (für rekursiven Aufruf)
RCSFLAGS			Flags für *rcs* allgemein
RM	*rm*	*del*	Löschbefehl
ECHO	*echo*	*echo*	Ausgabebefehl
EXE	–	.exe	Suffix ausführbarer Programme
OBJ	.o	.obj	Suffix von Objektdateien
LIB	.a	.lib	Suffix von Bibliotheken
HDRS			Headerdateien
SRCS			C-Quellcodedateien
OBJS			Objektdateien
LIBS			Bibliotheken
INCDIRS			Includeverzeichnisse
INSTDIR			Installationsverzeichnis
LDFILE	–	*.lnk	Linkdatei

2.3.3 Spezielle Makros

Weiterhin gibt es **spezielle Makros**, die automatisch in Abhängigkeit von der aktuellen Regel definiert sind. Diese Makros sind vor allem in Suffixregeln nützlich:

Makro	Bedeutung	Wo verwendbar
$@	Name des Ziels	Beide Regelarten
$?	Alle Quellen, die jünger als das Ziel sind	Spezielle Regel
$<	Name der Quelle, die die Aktion auslöste	Suffixregel
$*	Analog, aber ohne Suffix (.c fällt weg)	Suffixregel

Folgendes Beispiel ist eine Suffixregel für die Übersetzung von C-Quellcodedateien in Objektdateien, ihre Aufnahme in eine Bibliothek und das Löschen der Objektdateien:

```
.c.a:
    $(CC) -c $<           # Tabulator am Zeilenanfang nicht vergessen!
    $(AR) $@ $*$(OBJ)     # Tabulator am Zeilenanfang nicht vergessen!
    $(RM) $*$(OBJ)        # Tabulator am Zeilenanfang nicht vergessen!
```

2.4 Leerzeilen + Fortsetzungszeilen

Leerzeilen sind überall in einem Makefile erlaubt (*außer zwischen Regeln und zugehörigen Aktionen*). Sie können z.B. zwischen Regeln sowie zur Einteilung der Makrodefinitionen in Gruppen eingefügt werden.

Durch einen **Backslash** '\' *direkt vor* dem Zeilenende wird eine **logische Zeile** auf der nächsten **physikalischen Zeile** fortgesetzt:

```
OBJS = a.o b.o c.o \
       d.o e.o f.o
```

3 Ablauf

3.1 Aufruf + Optionen

Der **Aufruf von** *make* lautet:

```
make [-f FILE] {OPTION} {MACRO=text...} [TARGET...]
```

Wird *make* ohne Argumente aufgerufen, sucht es ein **Makefile** namens `makefile` oder `Makefile` im aktuellen Verzeichnis. Die **erste spezielle Regel** darin (meist `all`) und alle davon abhängigen Regeln werden ausgeführt. Soll eine andere Makedatei verwendet werden, kann sie über die Option `-f FILE` angegeben werden. Sollen andere Ziele erzeugt werden, sind diese als Argument `TARGET...` anzugeben. Zusätzliche Makrodefinitionen können in der Form `MACRO=text...` angegeben werden. Sie überschreiben eine eventuell im Makefile vorkommende Definition des gleiche Makros. Als **Optionen** `OPTION` sind unter anderem möglich:

Option	Name	Bedeutung
`-f FILE`	file	Als Makefile `FILE` verwenden (Std: `makefile`, `Makefile`)
`-d`	debug	Ablaufinformationen und Datum ausgeben
`-p`	print	Vordefinierte Makros und Suffixregeln ausgeben
`-s`	silent	Aktionen bei Ausführung NICHT ausgeben
`-i`	ignore	Fehler in Aktion → NICHT abbrechen
`-k`	keep-go	Fehler in Aktion → NICHT durchführbaren Teil überspringen
`-n`	noexec	Aktionen ausgeben, aber NICHT ausführen
`-q`	question	Exit-Status „Ziel aktuell“ ermitteln (keine Aktionen)
`-t`	touch	Datum der Ziele aktualisieren (keine Aktionen)
`-e`	envvar	Umgebungsvariablen überschreiben Makrodefinitionen
`-j`	jobs	Bis zu dieser Anzahl Aktionen parallel starten
`-r`	norules	Eingebaute Regeln NICHT anwenden

3.2 Regeln + Aktionen

Eine **Abhängigkeitsregel** besagt, dass die Zieldateien jüngeren Datums (neuer) sein müssen als die Quelldateien (verglichen wird das **Datum der letzten Änderung**) Ist eine der

Quelldateien jünger, so ist die Regeln nicht erfüllt und die Zieldateien müssen neu erzeugt werden. Es werden dann alle auf eine Abhängigkeitsregel folgenden **Aktionen** (Kommandos) ausgeführt. Diese sollten die Neuerstellung der Zieldateien aus den Quelldateien auf geeignete Weise bewirken (müssen das aber nicht unbedingt):

```
foo: foo.c foo.h
    gcc foo.c           # Tabulator am Zeilenanfang nicht vergessen!
```

In diesem Beispiel hängt das Programm (die Datei) `foo` von den Dateien `foo.c` und `foo.h` ab. Ist eine der beiden Dateien `foo.c` oder `foo.h` neueren Datums als `foo`, so wird `foo.c` mit *gcc* übersetzt und das Programm `foo` dadurch neu erzeugt (*Achtung: vor gcc muss ein Tabulator stehen!*).

Da jede Quell(Datei) in einer Abhängigkeitsregel selbst wieder Ziel(Datei) einer anderen Abhängigkeitsregel sein kann, wird durch ein `Makefile` ein **Abhängigkeitsbaum** definiert. *make* baut diesen Baum auf und arbeitet ihn von den Blättern zur Wurzel hin ab.

Eine **Suffixregel** gilt für Zieldateien mit dem entsprechenden Suffix, für die keine spezielle Abhängigkeitsregel existiert. Mit den danach angegebenen Aktionen kann sie aus der (bis auf den Suffix) gleichnamigen Quelldatei erzeugt werden. Das Makro `$<` wird dabei durch den Namen der Quelldatei ersetzt (analog `$*` ohne Suffix), das Makro `$@` durch den Zielnamen.

Die nach einer Abhängigkeitsregel aufgelisteten Aktionen werden Zeile für Zeile an den **Standardkommandoprozessor** *sh* unter UNIX (*command.com* unter MSDOS) weitergegeben. Da für jede Zeile ein eigener Kommandoprozessor mit eigenem Environment aufgerufen wird, kann im Umgebungsbereich abzulegende Information nicht von einem Kommando an das nächste weitergegeben werden (z.B. werden Befehle zum Wechseln des Verzeichnisses oder zum Setzen von Umgebungsvariablen nach der Zeile wieder vergessen).

```
/usr/local/bin/foo: /usr/local/src/foo.c /usr/local/src/foo.h
    cd /usr/local/bin                               # Bringt nichts
    gcc /usr/local/src/foo.c                        # Bringt nichts
    cd /usr/local/bin; gcc /usr/local/src/foo.c     # Funktioniert

foo: foo.c foo.h
    VAR="Wert"                                      # Bringt nichts
    gcc foo.c                                       # Bringt nichts
    VAR="Wert" gcc foo.c                            # Funktioniert
    export VAR="Wert"; gcc foo.c                    # Funktioniert
```

3.3 Fehler

Kann ein Ziel aus irgendeinem Grund nicht erzeugt werden (weil z.B. eine Aktion zu einem Fehler führt), so wird seine veraltete Form (sofern vorhanden) gelöscht. Um dies für ausgewählte Ziele zu verhindern, können nach dem Befehl `.PRECIOUS` (*wertvoll*) alle Dateien angegeben werden, die nicht gelöscht werden dürfen:

```
.PRECIOUS mylib.a ...
```

Ein Makelauf wird normalerweise abgebrochen, wenn eine Aktion einer Regel zu einem Fehler führt. Durch Voranstellen eines **Minuszeichen** '-' direkt vor der Aktion kann dies für eine einzelne Aktion verhindert werden (bei Angabe des Flags `-i`=ignore führt ein Fehler grundsätzlich nicht zum Abbruch des Makelaufs):

```
.c.a:
    $(CC) -c $<
    $(LIB) $@ $*$(OBJ)
    -$(RM) $*$(OBJ)
```

3.4 Portabilität

In einem Makefile sollten aus Gründen der Portabilität und Übersichtlichkeit alle Kommandos und Dateinamen *klein*, sowie alle Makronamen *groß* geschrieben werden. Statt einen Kommando- oder Dateinamen mehrmals direkt zu verwenden, ist es aus dem gleichen Grund besser, ein Makro zu definieren und dieses zu verwenden.

Während unter UNIX Kommandozeilen (fast) beliebig lang sein können, dürfen sie unter MSDOS maximal 126 Zeichen lang sein. Daher muss dort bei längeren Kommandozeilen wie sie z.B. für das Linken notwendig sind, auf eine Ausweichkonstruktion mit einer externen Link-Datei zurückgegriffen werden (siehe folgendes Beispiel). Makros und Abhängigkeitsregeln werden dagegen von *make* direkt verarbeitet und dürfen auch unter MSDOS (fast) beliebig lang sein.

4 Musterbeispiel

Als Musterbeispiel für ein übersichtliches und portables Makefile sei folgendes, für das Programm `foo` erstelltes, besprochen. Die einzelnen logischen Abschnitte sind durch Großbuchstaben gekennzeichnet, zu jedem dieser Abschnitte folgt eine ausführliche Erklärung:

A. Zunächst werden die Namen der verwendeten Kommandos und Suffixe als Makros definiert. Soll das gleiche Makefile unter MSDOS ablaufen, sind die danebenstehenden Namen als Werte einzusetzen.

```
#-------------------------------------------------------------------------
# Makefile
#-------------------------------------------------------------------------
# A. Programmnamen und Suffixe unter UNIX  # MSDOS
#-------------------------------------------------------------------------
CC   = gcc                                 # cl
LD   = ld                                  # link
RM   = rm                                  # del
ECHO = echo                                # echo
EXE  = # leer!                             # .exe
OBJ  = .o                                  # .obj
```

B. Es folgen Makros für die C-Compilerflags und Verzeichnisse, in denen sich die zum Übersetzen notwendigen Includedateien befinden. Analog werden die Linkerflags und die Namen der zum Binden benötigten Bibliotheken als Makros definiert. Alternative Belegungen der Compiler- und Linkerflags für Test- und Debugzwecke sind als Kommentar angegeben, sie können so jederzeit aktiviert werden.

```
#-------------------------------------------------------------------------
# B1. Flags und Includeverzeichnisse fuer Compiler
#-------------------------------------------------------------------------
#CFLAGS  = -AL -W4              #large model, warn all
#CFLAGS  = -AL -G2 -Gs -Ox      #large model, 286-code, no stack check, optimize
#CFLAGS  = -AL -G2 -Od -Zi -W3  #large model, 286-code, no optimize, codeview-info
CFLAGS  = -Wall -O             #warnings, optimize
INCDIRS = -I. -Itoolbox -Iarrhandl

#-------------------------------------------------------------------------
# B2. Flags und Bibliotheken fuer Linker (MSDOS)
#-------------------------------------------------------------------------
#LDFLAGS = /NOI /SE:0x90 /CO   # Codeview-Info
#LDFLAGS = /NOI /SE:0x90
LDFLAGS =
#LIBS = toolbox\mar_ct_l.lib arrhandl\l_ary.lib c:\c600\lib\llibce.lib
LIBS =
```

C. Die nächsten Makros werden zum Festlegen der Abhängigkeiten verwendet: `MAIN` ist der Name des zu erzeugenden Programms `foo`, `HDRS` enthält die Namen aller darin verwendeten Headerdateien. Wie man an diesem Makro sieht, können Dateien auch über die Wildcards `*` und `?` definiert werden. `SRCS` enthält die Namen aller Quellcodedateien, aus denen `foo` besteht. Durch Backslash vor dem Zeilenende wird dabei die logische Zeile auf drei physikalische Zeilen aufgeteilt.

Die Definition des Makros `OBJS` für die Objektdateinamen wird zur Illustration über drei Untermakros `OBJ1-3` durchgeführt, um zu zeigen, wie man lange Listen übersichtlich aufbauen kann (jedes der Untermakros ist deutlich kürzer als 126 Zeichen, d.h. auch unter MSDOS können diese Makros in einer Aktion verwendet werden).

```
#-------------------------------------------------------------------------
# C. Hauptprogramm + seine Header-, Quellcode- und Objektdateien
#-------------------------------------------------------------------------
MAIN = foo$(EXE)
HDRS = *.h
SRCS = foo.c \
       foo1.c foo2.c foo3.c foo4.c \
       foo5.c foo6.c
OBJ1 = foo$(OBJ)
OBJ2 = foo1$(OBJ) foo2$(OBJ) foo3$(OBJ) foo3$(OBJ)
OBJ3 = foo5$(OBJ) foo6$(OBJ)
OBJS = $(OBJ1) $(OBJ2) $(OBJ3)
```

D. Die weiteren Makros legen den Namen der Linkdatei `LDFILE` und die Namen der für die Installation bzw. Sicherung notwendigen Programme und Pfade fest.

```
#-----------------------------------------------------------------------
# D1. Linkdatei und Installationsverzeichnis
#-----------------------------------------------------------------------
LDFILE  = foo.lnk # MSDOS
INSTDIR = /usr/local/bin

#-----------------------------------------------------------------------
# D2. Zum Sichern
#-----------------------------------------------------------------------
BACKUP = tar
BFLAGS = u
DEST   = /dev/rmt0
```

E. Danach folgt die Suffixregel zum Übersetzen von C-Quellcodedateien in Objektdateien. Ihre einzige Aktion ist allgemeingültig aus Makros zusammengesetzt. Änderungen sind daher nur an einer Stelle, nämlich den Makros, vorzunehmen. Diese Suffixregel wird auf jede Objektdatei angewendet, für die keine spezielle Regel existiert, die aber aufgrund einer anderen Regel benötigt wird.

```
#-----------------------------------------------------------------------
# E. Allgemeine Uebersetzungsregel fuer C-Programme
#-----------------------------------------------------------------------
.c$(OBJ):
    $(CC) $(CFLAGS) $(INCDIRS) -c $*.c
```

F. Die **erste spezielle Regel** mit dem Ziel `all` legt fest, dass das Programm `foo` zu erzeugen ist, da die Datei `all` (hoffentlich) nicht existiert. Aktionen werden in dieser Regel nicht durchgeführt, insbesondere wird auch keine Datei `all` erzeugt. D.h. beim nächsten Aufruf von *make* wird wieder `foo` erzeugt, sofern seit dem letzten Makelauf eine Änderung an einer seiner Quelldateien stattfand. Diese Regel wird standardmäßig ausgeführt, wenn *make* (im Verzeichnis des Makefiles) ohne Argumente aufgerufen wird.

```
#-----------------------------------------------------------------------
# F. Erste Regel, wird beim Aufruf von make ohne Argumente ausgefuehrt
#-----------------------------------------------------------------------
all: $(MAIN)
```

G. Die nächste Regel mit dem Ziel `clean` hat keine Abhängigkeiten. Da eine Datei `clean` (hoffentlich) nicht existiert, werden ihre Aktionen **immer** ausgeführt und damit alle Zwischendateien gelöscht. Soll z.B. das Programm `foo` mit den Compilerflags `-O1` neu übersetzt werden, so ist zunächst `make clean` und dann `make CFLAGS="-O1"` aufzurufen.

```
#-----------------------------------------------------------------------
# G. Loescht alle Zwischen- und Enddateien (fuer komplette Neuerstellung)
#-----------------------------------------------------------------------
clean:
    $(RM) $(MAIN)
    $(RM) *$(OBJ)
```

H. Die Regel `new` ruft zuerst `clean` und dann `all` auf, d.h. das Programm `foo` wird vollständig neu erzeugt.

```
#---------------------------------------------------------------------
# H. Programm vollstaendig neu erstellen
#---------------------------------------------------------------------
new: clean all
```

I. Die Regel `install` verschiebt die erzeugten Programme und Dateien in Verzeichnisse, die im Suchpfad liegen. Dies ist wichtig, um allen Anwendern den Aufruf zu ermöglichen.

```
#---------------------------------------------------------------------
# I. Programm installieren
#---------------------------------------------------------------------
install:
    $(CP) $(MAIN) $(INSTDIR)
```

J. Die Regel `backup` sichert seit der letzten Sicherung veränderte Quelldateien, die Regel `save` sichert sämtliche Quelldateien.

```
#---------------------------------------------------------------------
# J1. Geaenderte Quellcodedateien sichern
#---------------------------------------------------------------------
backup:
    $(BACKUP) $(BFLAGS) makefile $(DEST)
    $(BACKUP) $(BFLAGS) *.c      $(DEST)
    $(BACKUP) $(BFLAGS) *.h      $(DEST)

#---------------------------------------------------------------------
# J2. Alle Quellcodedateien sichern
#---------------------------------------------------------------------
save:
    $(BACKUP) makefile $(DEST)
    $(BACKUP) *.c      $(DEST)
    $(BACKUP) *.h      $(DEST)
```

K. Die letzte Regel ist für die Erzeugung des Programms `foo` verantwortlich. `foo` muss dann neu erzeugt werden, wenn sich eine der Header- oder Objektdateien geändert hat. Da es für keine der Objektdateien eine spezielle Regel gibt, aber eine Suffixregel für sie existiert, wird die Suffixregel auf sie angewendet. D.h. jede Objektdatei muss neuer als ihre zugehörige C-Quellcodedatei sein. Ist sie das nicht, wird sie durch die Aktion der Suffixregel erzeugt. Für die Headerdateien gibt es keine Abhängigkeitregeln, sie werden aus keiner anderen Datei generiert, d.h. es genügt, dass sie vorhanden sind.

Als Aktion der letzten Regel genügt im Prinzip ein einziger Linkbefehl zum Erzeugen von `foo`, der allerdings auskommentiert ist, da dieser Befehl nach dem Einsetzen der Makrotexte länger als 126 Zeichen würde. Daher wird das Kommando stückweise (jeweils weniger als 126 Zeichen) über `ECHO`-Befehle in der Linkdatei `foo.lnk` abgelegt und dem Linker dieses File übergeben (Kennzeichen ist ein `@` vor dem Linkdateinamen). Abschließend wird die Linkdatei wieder entfernt.

```
#-------------------------------------------------------------------------
# K. Programm erstellen (ECHO-Trick wg. beschraenkter Zeilenlaenge)
#-------------------------------------------------------------------------
$(MAIN): $(OBJS) $(HDRS)
#   $(LD) $(OBJS),$@,,$(LIBS),$(LDFLAGS); ### wg. MSDOS Kommando <= 126 Zeichen
#   $(ECHO) $(OBJ1) +   > $(LDFILE)
#   $(ECHO) $(OBJ2) +  >> $(LDFILE)
#   $(ECHO) $(OBJ3)    >> $(LDFILE)
#   $(ECHO) $@         >> $(LDFILE)
#   $(ECHO) nul        >> $(LDFILE)
#   $(ECHO) $(LIBS)    >> $(LDFILE)
#   $(ECHO) $(LDFLAGS) >> $(LDFILE)
#   $(ECHO) nul        >> $(LDFILE)
#   $(LD) @$(LDFILE)
#   $(RM) $(LDFILE)
    $(CC) -o $(MAIN) $(OBJS) $(LIBS) -lm
```

5 Zusammenfassung

5.1 Standard(Pseudo)ziele

Ziel	Bedeutung
`all`	Erstes Ziel zum Erzeugen aller Programme und Dateien
`clean`	Löschen aller Zwischen- und Enddateien
`new`	Neuerstellen aller Programm und Dateien (`clean` + `all`)
`depend`	Generieren und Eintragen der Abhängigkeiten der C-Quellcodedateien von den Headerdateien in das Makefile
`install`	Installation der erzeugten Programme und Dateien
`uninstall`	Deinstallation der erzeugten Programme und Dateien
`backup`	Sichern der seit der letzten Sicherung geänderten Quelldateien
`save`	Sichern aller Quelldateien
`dist`	Ein Quelltextpaket erzeugen (Distribution)
`check`	Prüfungen durchführen
`test`	Testen aller Programme
`usage`	Usage-Meldung ausgeben

5.2 Makroquellen

Vorrang	Quelle	Befehl
1	Als Aufrufargument	`make VAR="Wert" ...`
2	Im Makefile gesetzt	`VAR = Wert ...`
3	Temporäre Umgebungsvariable	`VAR="Wert" make ...`
4	(Normale) Umgebungsvariable	`export VAR="Wert"; make ...`

5.3 Standardmakros

Makro	UNIX	MSDOS	Bedeutung
CO	*co*		Checkout-Befehl (RCS)
CC	*gcc*	*cl*	C-Compiler
LD	*ld*	*link*	Linker
AR	*ar*	*lib*	Bibliotheksverwalter
AS	*as*	*masm*	Assembler
LEX	*flex*	*lex*	Scannergenerator
YACC	*bison*	*yacc*	Parsergenerator
MAKE	*make*	*nmake*	*make* selbst (für rekursiven Aufruf)
COFLAGS			Flags für Checkout-Befehl
CFLAGS			Flags für C-Compiler
LDFLAGS			Flags für Linker
ARFLAGS			Flags für Bibliotheksverwalter
ASFLAGS			Flags für Assembler
LFLAGS			Flags für Scannergenerator
YFLAGS			Flags für Parsergenerator
MAKEFLAGS			Flags für *make* (für rekursiven Aufruf)
RCSFLAGS			Flags für *rcs* allgemein
RM	*rm*	*del*	Löschbefehl
ECHO	*echo*	*echo*	Ausgabebefehl
EXE	–	.exe	Suffix ausführbarer Programme
OBJ	.o	.obj	Suffix von Objektdateien
LIB	.a	.lib	Suffix von Bibliotheken
HDRS			Headerdateien
SRCS			C-Quellcodedateien
OBJS			Objektdateien
LIBS			Bibliotheken
INCDIRS			Includeverzeichnisse
INSTDIR			Installationsverzeichnis
LDFILE	–	*.lnk	Linkdatei

5.4 Spezielle Makros

Makro	Bedeutung	Wo verwendbar
$@	Name des Ziels	Beide Regelarten
$?	Alle Quellen, die jünger als das Ziel sind	Spezielle Regel
$<	Name der Quelle, die die Aktion auslöste	Suffixregel
$*	Analog, aber ohne Suffix (.c fällt weg)	Suffixregel

5.5 Optionen

Option	Name	Bedeutung
`-f FILE`	file	Als Makefile `FILE` verwenden (Std: `makefile`, `Makefile`)
`-d`	debug	Ablaufinformationen und Datum ausgeben
`-p`	print	Vordefinierte Makros und Suffixregeln ausgeben
`-s`	silent	Aktionen bei Ausführung NICHT ausgeben
`-i`	ignore	Fehler in Aktion → NICHT abbrechen
`-k`	keep-go	Fehler in Aktion → NICHT durchführbaren Teil überspringen
`-n`	noexec	Aktionen ausgeben, aber NICHT ausführen
`-q`	question	Exit-Status „Ziel aktuell“ ermitteln (keine Aktionen)
`-t`	touch	Datum der Ziele aktualisieren (keine Aktionen)
`-e`	envvar	Umgebungsvariablen überschreiben Makrodefinitionen
`-j`	jobs	Bis zu dieser Anzahl Aktionen parallel starten
`-r`	norules	Eingebaute Regeln NICHT anwenden