```
                    UNIX-Werkzeuge
                    ==============


                       2. make
                       =======




       Hilfsprogramm zur Verwaltung von veränderlichem Gut
       für vergessliche und faule Unix-Anwender.
```



```
Historisches
------------

   Verfasser: Stuart I. Feldman für UNIX 1978

     Dialekte:
              imake - Projektmanagment für portable Software
                       (X-Window-Syste)
              gmake - GNU-Make, Erweiterung von make
              nmake - AT&T Erweiterung von make

   Abgeschrieben für Windows-Welt:
              nmake - Microsoft
              make  - Borland


Einsatzgebiet von "make"
------------------------
```

Hauptaufgabe von "make" ist die Verwaltung von großen und kleinen Projekten. Derartige Projekte bestehen in der Regel aus vielen kleinen Quelltexten, die nach bestimmten Regeln zu Objekten zusammengefügt werden. Diese Abhängigkeitsregeln können in "make" formalisiert aufgeschrieben werden. Dadurch kann "make" dann jederzeit alle Operationen durchführen, die für die Herstellung eines vollständigen, korrekt erzeugten Projekts notwendig sind. "make" berücksichtigt dabei die Zeitstempel der jeweiligen Quellen und Ziel-Objekte, so daß nur die minimal notwendigen Aktionen durch "make" veranlaßt werden. m1

Arbeitsweise von "make"

"make" verlangt für seine Arbeit eine Beschreibungsdatei genannt "Makefile". Dieses Makefile wird, wenn es nicht explizit spezifiziert wurde, in der aktuellen Directory unter den Namen "makefile" und "Makefile" gesucht.

```
Achtung: Reihenfolge: 1. makefile
                      2. Makefile
```

m2

Mit der Option -f kann der Anwender das Makefile explizit bestimmen.

```
   z.B.:  make -f mein-makefile all
```

Bei der Arbeit protokolliert "make" alle Aktionen, die ausgeführt werden, auf der Standardausgabe. Durch das Voranstellen von "@" vor ein Kommando im Makefile wird die Ausgabe des entsprechenden Kommandos unterdrückt. Derartig maskierte Kommandos werden durch die Option -n sichtbar, die aber die Ausführung aller Kommandos unterdrückt.
Durch die Option -s wird "make" veranlaßt, keinerlei Ausgaben zu machen.

m3

Bei Fehlern (Exit-Kode eines ausgeführten Kommandos ungleich 0) bricht "make" die Arbeit sofort ab. Dies kann durch das Voranstellen eines Minuszeichen unmittelbar vor dem auszuführenden Kommando verhindert werden. Das Minuszeichen kann auch beliebig mit "@" kombiniert werden.
Die Option -i bewirkt generell das Ignorieren von Fehlern.

m4

Was erzeugt "make"?

Beim Aufruf von "make" kann ein Objekt(Ziel) angegeben werden. Dann werden alle Aktionen ausgeführt, die zur Bildung des angegeben Objekts notwendig sind. Die Aktionen für die Bildung von Objekten (Zielen) werden im Makefile beschrieben. Wird kein Ziel angegeben, so wird das erste zu bildende Objekt, das im Makefile beschrieben ist, gebildet.

Das Makefile

Das Makfile besteht im wesentlichen aus Regeln. Diese Regeln haben folgende einfache Syntax:

```
 <Make-Regel>::=
   <Zielobjekt> ":"  { <Quellobjekt> } [";" <Kommando> ] { <NL>
                          <TAB> <Kommando> }  |
   <Zielobjekt> "::" { <Quellobjekt> } [";" <Kommando> ] { <NL>
                          <TAB> <Kommando> }

   <Zielobjekt>  - ist das Objekt, das erzeugt werden soll
   <Quellobjekt> - ist ein Objekt, von dem das Zielobjekt in irgendeiner
                   Form abhängig ist. Mehrere Quellobjekte sind zulässig.
                   Ein <NL>-Zeichen in der Liste der Quellobjekte muß
                   durch "\" maskiert werden. Die Abhängigkeitsinformationen
                   müssen in einer Zeile stehen.
   <Kommando>    - Kommandos deren Abarbeitung, für die Bildung des
                   Zielobjektes aus den Quellobjekten notwendig sind.
                   In der Regel Compilerläufe, Formatierungskommandos
                   und Kopieroperationen.
```

Ein Zielobjekt kann mehrfach auf der linke Seite auftreten (von verschieden Quellobjekten abhängen), allerdings darf nur einmal ein Kommandoteil folgen. Als Trennzeichen zwischen Zielobjekt und Quellobjekt ist der ":" zu verwenden. Soll ein Zielobjekt mehrfach durch verschiedene Kommandoteile gebildet werden, so sind Zielobjekt und Quellobjekte durch "::" zu trennen.
Kommentare: Kommentare beginnen mit "#" und enden am Zeilenende. Sie können anstelle von Make-Regeln oder am Ende jeder Zeile stehen.

Beispiel:

```
c_sock:               # Client erzeugen
c_sock: c_sock.c inet.h
        gcc -c   c_sock.c
        gcc -o c_sock c_sock.o ${LIB} ${LIB1}
Tabulator!!!!!!!!!!

libcom::               # 1.LIB-Modul erzeugen
libcom:: c_sock.c inet.h
        gcc -c   c_sock.c
        ar -r $@ c_sock.o
libcom::               # 2.LIB-Modul erzeugen
libcom:: s_sock.c inet.h
        gcc -c   s_sock.c
        ar -r $@ s_sock.o

Tabulator!!!!!!!!!!
```

m5

Es gibt zwei Philosophien für das Aussehen des ersten Ziels innerhalb des Makefiles:

1. Die Regel:

```
all: <Liste aller zu bildenden Objekte des Porjekts>
```

2. Die Regel:

```
help: ;@egrep '^[^:;=.]*::?[     ]*#' [mM]akefile
```

und alle Startregeln ordentlich Kommentieren.

```
xyx:      # erzeugen von xyz
```

m6

Makro-Mechanismus

"make" unterstützt innerhalb des Makefiles einen Makro-Mechanismus zur einfacheren und übersichtlicheren Schreibweise des Files.

```
<Makrodefinition>::= <Makroname> "=" <String>
```

Makrodefinitionen können vor und zwischen den Make-Regeln innerhalb des Makefiles auftreten. Der Makroname besteht aus Buchstaben, Ziffern und Unterstrichen.

Der Zugriff auf ein Makro kann durch:

```
                    "$("<Makroname>")"
                         oder
                    "${"<Makroname>"}"
```

erfolgen.

Bei der Definition von Makros ist der Zugriff auf vorher definierte Makros erlaubt.

Im Makefile kann auf folgende 4 Makroarten zugegriffen werden:

- im Makefile vom Nutzer definierte Makros
- Umgebungsvariablen der Shell können ebenfalls innerhalb des Makefiles wie Makros benutzt werden.
- beim Aufruf von "make" können in der Kommandozeile mittels Schlüsselwortparameter Makros definiert werden, z.B.

```
make CC=cc all
```

- intern vordefinierte Makros:

```
env - make -p -f /dev/null | egrep  -e "^[a-zA-Z_-]* ="
```

m7/int-vor-make

```
MAKE = $(MAKE_COMMAND)
COFLAGS =
CO = co
FC = f77
CC = cc
CXX = g++
```



```
AR = ar
CWEAVE = cweave
YACC = yacc
MAKEFLAGS = p
OUTPUT_OPTION = -o $@
TANGLE = tangle
LD = ld
MFLAGS = -p
GET = get
PC = pc
AS = as
TEX = tex
LINT = lint
RM = rm -f
WEAVE = weave
CPP = $(CC) -E
LEX = lex
ARFLAGS = rv
CTANGLE = ctangle
MAKEINFO = makeinfo
```

Reihenfolge der Benutzung der Makros im Makefile durch "make":

ohne -e Option	mit -e Option
1. Schlüsselwortparameter	1. Schlüsselwortparameter
2. Definitionen im Makefile	2. Umgebungsvariable
3. Umgebungsvariable	3. Definitionen im Makefile
4. intern vordefinierte Makros	4. intern vordefinierte Makros

m7

```
Interne Makros:

$@, $(@D), $(@F)
     Name des ungültigen Zielobjektes (das Objekt, das erzeugt
     werden soll). $(@D) steht für den Path-Name und $(@F) für den Filenamen.

$?
     Liste der Namen aller neuen Quellen, neuer als das Zielobjekt

$<, $(<D), $(<F)
     Name des transformierbaren Objektes (das Objekt, das übersetzt
     werden soll). $(<D) steht für den Path-Name und $(<F) für den Filenamen.

$*, $(*D), $(*F)
     Name des ungültigen Zielobjektes ohne Suffix, das die Transformation
     ursprünglich ausgelöst hat.
     $(*D) steht für den Path-Name und $(*F) für den Filenamen.
                                                                          m8

$%, $(%D), $(%F)
     Wenn das Zielobjekt eine Bibliothek ist (lib.a(file.o)), so beschreibt
     $% die Datei file.o und $@ die Bibliothek lib.a.
     $(%D) steht für den Path-Name und $(%F) für den Filenamen der
     Datei.
                                                                          m9
```



```
Suffixregeln
------------

"make"  hat eingebaute Abhängigkeitsregeln - Suffixregeln, die  es für die
Bildung von Objekten benutzt, wenn keine expliziten Bildungsvorschriften für
ein Ojekt angegeben sind. Man kann diese vollständig mittels:

        env - make -pf /dev/null  | sed -e "/^#/d" -e "/^$/d" | more

besichtigen.
Mit

   env - make -pf /dev/null  | sed -e "/^#/d" -e "/^$/d" | grep .SUFFIXES:
                                                                m9/get-sufix
erhält man alle intern unterstützten Suffixe

    .SUFFIXES : .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l .s .S \
       .mod .sym .def .h .info .dvi .tex .texinfo .texi .txinfo .w .ch \
       .web .sh .elc .el

Für die Suffixe sind dann auch noch zusätzliche Bildungsregeln definiert.
Mittels
      #!/bin/sh
      env - make -pf /dev/null | awk '  /^\.[a-zA-Z]\.[a-zA-Z]:/ { \
                      print $0 ;  getline ;   getline;  \
                      getline;  getline;  \
                      getline; print $0 ; \
                      getline; print $0 ; \
                 } '                                            m9/getreg2
erhält man eine kurze Übersicht über die internen Bildungsregeln.
```

```
.c.o:
        $(COMPILE.c) $(OUTPUT_OPTION) $<
.y.c:
        $(YACC.y) $<
        mv -f y.tab.c $@
.s.o:
        $(COMPILE.s) -o $@ $<
.F.f:
        $(PREPROCESS.F) $(OUTPUT_OPTION) $<
.S.s:
        $(PREPROCESS.S) $< > $@
.F.o:
        $(COMPILE.F) $(OUTPUT_OPTION) $<
.p.o:
        $(COMPILE.p) $(OUTPUT_OPTION) $<
.r.f:
        $(PREPROCESS.r) $(OUTPUT_OPTION) $<
.l.r:
        $(LEX.l) $< > $@
        mv -f lex.yy.r $@
.r.o:
        $(COMPILE.r) $(OUTPUT_OPTION) $<
.C.o:
        $(COMPILE.C) $(OUTPUT_OPTION) $<
.l.c:
        @$(RM) $@
        $(LEX.l) $< > $@
.f.o:
        $(COMPILE.f) $(OUTPUT_OPTION) $<
```


Selbstverständlich enthält "make" auch interne Regeln zum Bilden von ausführbaren Objekten direkt aus dem Quelltext ohne zwischendurch Objektmodule zu bilden:

```
  env - make -pf /dev/null | awk '  /^\.[a-zA-Z]:/ { print $0 ; \
                                       getline ;   \
                                       getline ;   \
                                       getline ;  \
                                       getline ;  print $0 ; \
                                       getline ; print $0 ; \
                                    } '
```

m9/getreg

z.B.:

```
      .c:
      #  commands to execute (built-in):
              $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
      .s:
      #  commands to execute (built-in):
              $(LINK.s) $^ $(LOADLIBES) $(LDLIBS) -o $@
      .F:
      #  commands to execute (built-in):
              $(LINK.F) $^ $(LOADLIBES) $(LDLIBS) -o $@
      .f:
      #  commands to execute (built-in):
              $(LINK.f) $^ $(LOADLIBES) $(LDLIBS) -o $@
      .o:
      #  commands to execute (built-in):
              $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

```
      .p:
      #  commands to execute (built-in):
              $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
      .r:
      #  commands to execute (built-in):
              $(LINK.r) $^ $(LOADLIBES) $(LDLIBS) -o $@
      .C:
      #  commands to execute (built-in):
              $(LINK.C) $^ $(LOADLIBES) $(LDLIBS) -o $@
      .S:
      #  commands to execute (built-in):
              $(LINK.S) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

```
.o  - Objektmodule
.c  - C-Quelltext
.f  - Fortran-Quelle
.s  - Assembler-Text
.l  - lex-Quelltext
.y  - yacc-Quelltext
.h  - Heder-Dateien
.sh - Shellscript
```


Der Anwender kann aber auch zusätzliche eigene Suffix-Regeln definieren:

```
     .SUFFIXES:.o.c.s
```

Hierdurch wird festgelegt, daß im aktuellen Makefile nur Abhängigkeiten für Objektmodule, C-Quelltexte und Assemblertexte existieren.
Dabei müssen dann auch noch Bildungsregeln festgelegt werden.
Hierfür gibt es folgende Syntax:

```
   "."<von>"."<nach>":"
    <TAB>          <Kommando>       # für spezielle Sachverhalte
            bzw.
    "."<von>":"
     <TAB>          <Kommando>      # für das Erzeugen von fertigen Objekten

.c:
       $(CC) $(CFLAGS) $< $(LDFLAGS) -o $@
.c.o:
       $(CC) $(CFLAGS)  -O2 -c -o $@ $<
```

Durch
    .SUFFIXES:
werden die vordefinierten Suffixregeln gelöscht.

m10

```
Spezielle Zielangaben:

.SUFFIXES:    - Definition neuer Suffixe

.DEFAULT:     - Das Scheinobjekt .DEFAULT wird immer dann benutzt,
                wenn "make" ein Objekt erzeugen soll und es keine Regel
                für die Bildung diese Objekts gibt.  Ohne das
                Scheinobjekt .DEFAULT wird "make" abgebrochen.
                                                                   m11
                 .DEFAULT:
                      cp ../m1/$@ .
              - Definition des Verhaltens, wenn benötigte Objekte
                nicht vorhanden sind

.IGNORE:      - Fehler ignorieren

.SILENT:      - Ausgabe auf Standardausgabe abschalten
                                                                   m12

.PRECIOUS:    - Definition von Zielobjekten, die bei Fehlern
                nicht gelöscht werden (für Bibliotheken).


include-Anweisung:

   include <Dateiname>

Rekursiver Make-Aufruf mit pseudo-Ziel

                                                                  Make
```



```
Syntax: make [Optionen] [Target] ...

Optionen:
  -b, -m                      Aus Kompatibilitätsgründen ignoriert..
                              Alte Beschreibungsdateien
  -C VERZEICHNIS, --directory=VERZEICHNIS
                              Wechsle in das VERZEICHNIS bevor etwas anderes
                              ausgeführt wird.
  -d                          Gebe viele Informationen zur Fehlersuche aus.
  --debug[=FLAGS]             Gebe verschiedene Arten von Debug-Information
                              aus.
  -e, --environment-overrides
                              Umgebungsvariablen überschreiben  "make"-
                              Steuerdateien.
  -f DATEI, --file=DATEI, --makefile=DATEI
                              Lese die Datei DATEI als  "make"-Steuerdatei.
  -h, --help                  Gib diese Nachricht aus und beende.
  -i, --ignore-errors         Ignoriere Fehler in den Kommandos.
  -I VERZEICHNIS, --include-dir=VERZEICHNIS
                              Durchsuche das VERZEICHNIS nach eingebundenen
                              "make"-Steuerdateien.
  -j [N], --jobs[=N]          Erlaube N Jobs gleichzeitig; unbegrenzte Anzahl
von Jobs ohne Argument..
  -k, --keep-going            Weiterlaufen, auch wenn einige Targets nicht
                              erzeugt werden konnten.
  -l [N], --load-average[=N], --max-load[=N]
                              Nur bei Belastung unterhalb N mehrere Prozesse
                              starten.
```

```
-n, --just-print, --dry-run, --recon
                            Kommandos nur anzeigen, nicht ausführen.
-o DATEI, --old-file=DATEI, --assume-old=DATEI
                            Betrachte DATEI als sehr alt und erzeuge sie
                            nicht neu.
-p, --print-data-base       Gib die interne Datenbank von "make" aus.
-q, --question              Keine Kommandos ausführen; der Exit-Status gibt
                            an, ob die Dateien aktuell sind.
-r, --no-builtin-rules      Deaktivieren der eingebauten impliziten Regeln.
-R, --no-builtin-variables  Deaktivieren der eingebauten Variablenbe-
                            legungen..
-s, --silent, --quiet       Gebe die Kommandos nicht aus.
-S, --no-keep-going, --stop
                            Schaltet -k ab..
-t, --touch                 Die Targets werden nur als aktualisiert
                            markiert,
                            nicht tatsächlich erneuert.
-v, --version               Gib die Versionsnummer von  "make"  aus und
                            beende.
-w, --print-directory       Gib das aktuelle Verzeichnis aus.
--no-print-directory        Schalte -w aus, selbst wenn es implizit
                            eingeschaltet wurde.
-W DATEI, --what-if=DATEI, --new-file=DATEI, --assume-new=DATEI
                            Betrachte die DATEI stets als neu.
--warn-undefined-variables  Gib eine Warnung aus, wenn eine undefinierte
                            Variable referenziert wird.
```