GNU Make — Kurzreferenz

23. Februar 2005

1 Aufruf

Der Make-Aufruf sieht wiefolgt aus: make [*options*] [*VAR1=val1 VAR2=val2 . . .*] *goal*.
Die wichtigsten Optionen (*options*) sind dabei:

–directory=*dir* (-C *dir*) Make wechselt vor der Ausführung in das Verzeichnis *dir*.

–environment-overrides (-e) Variablen aus der Umgebung haben Vorrang.

–makefile=file (-f *file*) Verarbeitet file als Makefile.

–help (-h) Hilfe

–ignore-errors (-i) Make ignoriert alle Fehler während der Verarbeitung.

–include-dir=dir (-I *dir*) Spezifiziert weitere Verzeichnisse für die Suche nach Makefiles für include.

–keep-going (-k) make soll trotz Fehler versuchen die Ausführung fortzusetzen.

–no-builtin-rules (-r) Alle eingebauten impliziten *Rules* werden von Make ignoriert.

–no-builtin-variables (-R) Alle vordefinierten Variablen von Make werden ignoriert.

–touch (-t) Aktualisiere Änderungszeit der Dateien (touch) anstatt die zugehörigen *Rules* auszuführen.

–version (-v) Ausgabe der Version von Make.

–warn-undefined-variables Make gibt eine Warnung bei undefinierten Variablen aus.

2 Rules

2.1 Explizite Rules

Jedes Makefile besteht aus sogenannten *Rules*, zu denen jeweils *Targets*, *Prerequisites* und *Commands* gehören. Eine explizite Rule wird folgendermaßen geschrieben:

```
target(s) : [prerequisites]
    [TAB shell-command]
    [TAB shell-command]
```

Diejenigen (abhängigen) *Targets*, die für das Erreichen eine "Ziel"-*Targets* aktualisiert werden müssen, nennt man *Goals*. Ob eine Aktualisierung notwendig ist, entscheidet die Änderungszeit einer gleichnamigen Datei — es sei denn das *Goal* oder auch die *Prerequesites* sind als .PHONY deklariert. Im letzteren Fall wird eine Datei weder vorausgesetzt noch erzeugt, d. h. das *Target* gilt immer als veraltet. Wie ein *Goal* bzgl. seiner *Prerequesites* aktualisiert wird, beschreiben die mit TAB beginnenden Shell-Commands.

Beim Make-Aufruf kann ein spezifisches *Goal* übergeben werden, wird es weggelassen dann findet immer die erste gültige *Rule* im Makefile Anwendung. Sind *Prerequesites* selbst wieder *Targets* in anderen *Rules*, dann setzt sich der Aktualisierungsprozeß rekursiv fort. Die einzelnen Zeilen einer *Rule* werden immer in separaten Sub-Shells ausgeführt, was bei der Verwendung von Variablen zu beachten ist.

2.2 Implizite Rules

Implizite *Rules* in Make-Files werden von GNU mittlerweile als *obsolet* bezeichnet, da sie von den (allgemeineren) *Pattern-Rules* abgelöst werden. Im Original beschreiben solche *Rules* die Beziehung zwischen Dateien bestimmter Extensions (im SInne von *Prerequisite* und *Target*) und ihre Bildungsregeln als Shell-Command, also z. B. :

```
.c.o:
    [TAB shell-command]
    [TAB shell-command]
```

Zahlreiche solcher *Rules* werden von Make durch Built-In *Rules* bereitgestellt. Die wichtigsten davon sind:

```
.c.o:
    $(CC) -c $(CFLAGS) $<
.cc.o:
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
.s.o:
    $(AS) $(ASFLAGS)
.y.c:
    $(YACC) $(YFLAGS)
.l.c:
    $(LEX) $(LFLAGS)
.tex.dvi:
    $(TEX)
```

2.3 Pattern-Rules

Oftmals führen explizite *Rules* zu vielen gleichartigen Zeilen, in den sich z. B. nur Teile von *Targets* bzw. Dateinamen ändern, die Verarbeitung aber immer gleich ist. Eine *Pattern-Rule* schafft die Möglichkeit der (gleichartigen) Verarbeitung für alle *Targets*, die zu einem *Pattern* passen. Die Schreibweise einer solchen *Rule* ist die folgende:

```
targets : target-pattern: dep-patterns
commands
```

Das *Target-Pattern* setzt sich aus einem Präfix gefolgt von einem Prozentzeichen "%" und (zuletzt) einem Suffix zusammen. Das Prozentzeichen ist ähnlich einem Wildcard zu interpretieren, wobei der Teil des *Targets*, zu dem es paßt, als auch als *Stamm* bezeichnet wird. Zum Beispiel paßt "%.o" zu allen *Targets* (oder Dateien), die auf ".o" enden.

```
$(objects): %.o: %.c
    $(CC) -c $(CFLAGS) $< -o $@
```

Make führt das Kommando für jedes Target aus, daß der *Pattern-Rule* entspricht — das aber nur dann, wenn das Target nicht noch (irgendwo) ein Kommando zur Generierung (z. B. aus einer expliziten *Rule*) hat[1].

2.4 Built-In Targets

.PHONY *Phony Target*, d. h. unabhängig davon ob eine gleichnamige Datei existiert wird es immer aktualisiert

.IGNORE Fehler bei der Ausführung zugehöriger Kommandos werden ignoriert

.PRECIOUS Stopt Make die Programmausführung dann wird die zum *Target* gehörige Datei nicht gelöscht

.DEFAULT Kann zu einem Target keine Rule ermittelt werden, dann werden die hier spezifizierten *Commands* angewendet.

.EXPORT_ALL_VARIABLES Hierbei handelt es sich weniger um ein *Target* als vielmehr eine Anweisung alle Variablen zu den Child-Prozessen (u.U. Sub-Makes) zu exportieren.

2.5 Bedingte Ausführung

Bedingte Ausführung ist durch Verwendung der folgende Schlüsselwörter möglich:

ifdef *VAR* TRUE, wenn Variable *VAR* definiert ist.

ifndef *VAR* TRUE, wenn Variable *VAR* nicht definiert ist.

ifeq (*VAR,val*) TRUE, wenn Variable *VAR* gleich *val* ist (Alternativen: ifeq '*VAR*' '*val*', ifeq "*VAR*" "*val*")

ifneq (*VAR,val*) TRUE, wenn Variable *VAR* ungleich *val* ist

Beispiel:

```
ifeq ($(CC),gcc)
    $(CC) -o foo $(objects) $(libs_for_gcc)
else
    $(CC) -o foo $(objects) $(normal_libs)
endif
```

Variablen können auf der Kommandozeile definiert (und vordefinierte sogar überschrieben) werden, Beispiel: make CFLAGS='-g -O'.

[1] Außerdem müssen die *Prerequisites* gefunden werden.

3 Variablen

3.1 Anwender-Variablen

Ähnlich wie in Shell-Scripten können Variablen definiert und verwendet werden, also z. B.

```
VAR="Hello World"
@echo $(VAR)
```

Weitere Formen der Variablenzuweisung sind: *VAR*:=, *VAR*?= , *VAR*+= , für die auf das GNU Make http://www.gnu.org/software/make/manual/html_mono/make.html Manual verwiesen wird. Sie unterscheiden sich in der Zuweisung des Wertes entweder zum Zeitpunkt der Deklaration oder der Verwendung[2].

Die Bezugnahme auf Shell-Variablen muß durch Doppel-Dollar, wie in $$*var* erfolgen.

3.2 Auto-Variablen[3]

$< (erstes/einziges) *Prerequisite*

$@ *Target*

$? alle *Prerequesite*, die neuer als das *Target* sind

$^ alle *Prerequesites*

$+ alle *Prerequesites*, aber doppeltes Vorkommen bleibt erhalten

$* Stamm der impliziten *Rule* oder *Pattern-Rule*

Varianten dieser Variablen können folgendermaßen (hier am Beispiel von $@) gebildet werden:

$(@D) Verzeichnis-Teil des Pfades von $@

$(@F) Datei-Teil des Pfades von $@

3.3 Umgebungsvariablen

In einem Makefile ist die Verwendung von Umgebungsvariablen möglich. Soche Variablen (sowie die auf der Kommandozeile übergebenen) werden auch an Sub-Makes exportiert. Sollen Variablen des aktuellen Makefiles exportiert werden, dann sind sie wiefolgt zu deklarieren:

```
VAR = value
export VAR
```

oder

```
export VAR = value
```

Make wertet insbesondere die folgenden Umgebungsvariablen aus:

[2] Dazu muß man wissen, daß GNU Make eine Zwei-Pass Anwendung ist.

[3] Ohne Berücksichtigung von Variablen, die beim Handling von Archiven eine Rolle spielen.

MAKE Name des Make-Programms

MAKEFILES Inhalt wird als Makefiles interpretiert, die vor der gesamten Verarbeitung ausgeführt werden sollen.

MAKEFLAGS Flags, so als wären sie beim Aufruf übergeben worden.

MAKECMDGOALS Liste von *Goals* die auf der Kommandozeile übergeben wurden.

SHELL Name der Shell.

CC Name des C/C++ Compilers

4 Funktionen

$(subst *from*,*to*,*text*) Ersetzt *from* durch *to* in *text*.

$(patsubst *pat*,*new*,*text*) Ersetzt in *text* alle Worte, die mit *pat* übereinstimmen durch *new*.

$(strip *string*) Entfernt alle Whitespaces aus *string*.

$(findstring *find*,*text*) Sucht den String *find* in text.

$(filter *pattern*...,*text*) Selektiert in *text* die Wörter, die mit einem der *pattern* übereinstimmen.

$(filter-out *pattern*...,*text*) Selektiert in *text* die Wörter, die mit keinem *pattern* übereinstimmen.

$(sort *list*) Sortiert die Wörter in *list*, wobei Doppelte entfernt werden.

$(dir *fnames*...) Extrahiert den Verzeichnis-Teil aus *fnames*.

$(notdir *fnames*...) Extrahiert den Dateinamen-Teil aus *fnames*.

$(suffix *fnames*...) Extrahiert die Dateinamen-Erweiterung aus *fnames*.

$(basename *fnames*...) Extrahiert den Dateinamen-Teil (ohne Erweiterung) aus fnames.

$(addsuffix *suffix*,*names*...) Hängt *suffix* an jedes Wort in *names* an.

$(addprefix *prefix*,*names*...) Fügt *prefix* vor jedes Wort in *names*.

$(join *list1*,*list2*) Vereinigt die Wortlisten *list1* und *list2*.

$(word *n*,*text*) Extrahiert das n-te (bei Eins beginnend) aus *text*.

$(words *text*) Ermittelt die Anzahl der Worte in *text*.

$(wordlist *s*,*e*,*text*) Gibt eine Liste von Worten aus *text* zurück, die von Position *s* bis *e* reicht.

$(firstword *names*...) Extrahiert das erste Wort aus *names*.

$(wildcard *pat*...) Sucht Dateien, die mit dem Shell-Pattern *pat* übereinstimmen.

$(error *text*...) Fehlerausgabe von *text* und Ende des Make-Prozesses.

$(warning *text*...) Ausgabe der Warnung *text*.

$(shell *cmd*) Ausführung von Shell-Command *cmd*.

$(origin *var*) Teilt mit woher die Variable *var* stammt (z. B. environment, file, command line, . . .)

$(foreach *var*,*list*,*text*) Ersatz der Shell FOR-Schleife (für Details siehe GNU Make Manual, 8.4).

$(call *var*,*param*,...) Berechne *var* neu, indem alle Referenzen $(1), $(2), . . . darin durch die Werte von *param*,. . . ersetzt werden.

Hinweis: Enthält eine Variable *VAR* einen Funktionsnamen, dann ist zum Aufruf das Konstrukt $(call *VAR*) zu verwenden.

5 Spezialzeichen

Kommentar

**** Zeilenfortsetzung (Verwendung am Zeilenende)

@cmd Unterdrückt Echo der Kommandozeile bei Ausführung von *cmd.*

-cmd Sollte *cmd* einen Fehler zurückgeben, wird der Make-Prozeß nicht gestopt.

+cmd *cmd* wird rekursiv ausgeführt.