Standard– und Programmiertools für Unix

Stephan Thesing

5. GNU `make`: Verwaltung von Projekten

Advanced Features:

- Bedingte Ausführung von `Makefiles`
- Funktionen zur Textersetzung
- Abhängigkeiten generieren
- Configurationsscript automatisch erzeugen

`VPATH`: Files suchen

```
VPATH=../include:../cc/include:../src

foo.o: foo.c header.h
	$(CC) -c $< -o $@
```

GNU `make`: `vpath` Direktive `vpath` *`pattern dirlist`*

```
vpath %.h ../include
vpath %.c ../src

foo.o: foo.c header.h
```

5.1. Bedingte Ausführung

Nur bestimmte Teile eines `Makefiles` werden gelesen.

⇒ Auflösung zum Zeitpunkt des Einlesens des Makefiles.

```
ifeq ($CC,)
 CC=gcc
fi

ifeq ($CC,gcc)
 CFLAGS+=-Wall -Werror -g
else
 CFLAGS+=-g
endif
```

Es gibt insgesamt vier Direktiven für bedingte Ausführung:

ifeq (s_1,s_2)

ifeq 's_1' 's_2'

ifeq "s_1""s_2" Wenn die beiden Zeichenfolgen gleich sind, wird der folgende Teil gelesen.

ifneq (s_1,s_2) Wenn die beiden Zeichenfolgen ungleich sind, wird der folgende Teil gelesen.

Man beachte: Zeichenfolgen, die nur aus Leerzeichen bestehen, sind nicht äquivalent zu leeren Zeichenfolgen.

ifdef v Wenn die Variable v definiert ist, wird der folgende Teil gelesen, z.B.

```
#ifdef CC
 CFLAGS+=-g
#endif
```

ifndef v Wenn die Variable v nicht definiert ist, wird der folgende Teil gelesen.

Eine leere Variable ist definiert, z.B.:

```
CFLAGS=
ifdef CFLAGS
 CFLAGS+=-g
endif
```

Andere (nützliche) Direktiven:

```
include makefile.inc ../global/Makefile.inc

override CFLAGS+=-g

export CFLAGS
export CC=gcc
export FLAGS:=$(FLAGS) -g
export FLAGS+=g

unexport CFLAGS
```

5.2. Funktionen

Funktionen dienen dazu,

- Text umzuformen
- Patternmatching zu machen
- Schleifen zu realisieren
- Kommandos in einer externen Shell aufzurufen

Werden ähnlich wie Variablenreferenzen geschrieben:

`$(`*`Func-name Args`*`...)`

Textmanipulation:

`$(subst` *from*`,`*to*`,`*text*`)`

`$(patsubst` *pattern*`,`*repl*`,`*text*`)`

```
FILES=a.c b.c c.c
OFILES=$(patsubst %.c,%.o,$(FILES))

AFILES=yacc.tab.c yacc.tab.h
ACFILES=$(subst .tab,,$(AFILES))
```

`$(strip text)` Entfernt Whitespace vom Anfang und Ende.

`$(findstring find,text)` Liefert *find*, wenn *text* *find* enthält, den leeren String sonst.

```
SOPTS=$(strip $(CFLAGS))
ifneq ($(SOPTS),)
ISWALL=$(find -Wall,$(SOPTS))
ifneq ($(ISWALL),)
# -Wall enabled
endif
endif
```

`$(filter` ***pattern...*,*text*** `)` Filtert Worte aus *text*, die auf keins der Pattern *pattern...* passen.

`$(filter-out` ***pattern...*,*text*** `)` Filtert Worte aus *text*, die auf eins der *pattern...* passen.

`$(sort` ***text*** `)` Sortiert alle Worte in *text*.

```
FILES=a.c foo.h b.c baz.y
CFILES=$(filter %.c,$(FILES))
NONCHFILES=$(filter-out %.c %.h,$(FILES))
SFILES=$(sort $(FILES))
```

Filenamen-Manipulation:

`$(dir` *`Namen`*`...)` Liefert den Directory-Teil für jeden Namen

`$(notdir` *`Namen`*`...)` Liefert den Filenamen-Teil für jeden Namen

```
FILES=../src/foo.c bar.c nix/

DFILES=$(dir $(FILES))
# Wird zu ../src ./ nix/

FFILES=$(notdir $(FILES))
# Wird zu foo.c bar.c
```

`$(suffix` *`Namen`*`...)` Liefert den Suffix jeden Namens

`$(basename` *`Namen`*`...)` Liefert den Namen ohne Suffix

```
FILES=../src/foo.c bar.c nix/

SFILES=$(suffix $(FILES))
# Wird zu .c .c

BFILES=$(basename $(FILES))
# Wird zu ../src/foo bar nix/
```

`$(addsuffix` ***Suffix*,*Namen*...**`)` Fügt *Suffix* an jeden Namen in *Namen* an.

`$(addprefix` ***Prefix*,*Namen*...**`)` Fügt *Prefix* vor jeden Namen an.

`$(join` ***Liste***$_1$,***Liste***$_2$`)` Fügt Worte aus den Listen aneinander.

```
FILES=a b c

CFILES=$(addsuffix .c,$(FILES))
# Liefert a.c b.c c.c

XFILES=$(addprefix foo_,$(CFILES))
# Liefert foo_a.c foo_b.c foo_c.c

YFILES=$(join $(FILES),.c .h .c)
# Liefert a.c b.h c.c
```

`$(word` *N*`,`*Text*`)` Liefert das N-te Wort aus *Text*.

`$(wordlist` *S*`,`*E*`,`*Text*`)` Liefert das S-te bis E-te Wort.

`$(words` *Text*`)` Anzahl Worte in *Text*.

```
FILES=a b c d

NFILES=$(words $(FILES))
# Liefert 4

SECOND=$(word 2,$(FILES))
# Liefert b

SECOND_TO_LAST=$(wordlist 2,4,$(FILES))
# Liefert b c d
```

`$(wildcard` *`Pattern`*`)` Expandiert Pattern im akt. Directory

Schleifen:

`$(foreach` *`Var`*`,`*`Liste`*`,`*`Body`*`)`

Shell & Sonstiges:

`$(origin` *`var`*`)` Liefert eine Zeichenkette, die den Typ der Variablen *var* angibt.

`$(shell` *`Args`*`)` Führt das Kommando *Args* in einer Subshell aus und liefert dessen Ausgabe als Ergebnis.

```
FILES=$(wildcard *.c)

XFILES=$(foreach v,$(FILES),$(v).nix)

ifneq ($(origin PATH),environment)
PATH=$(shell echo $$PATH)
endif
```

5.3. Abhängigkeiten generieren

Problem: Viele Abhängigkeiten (z.B. rekursiv via Header-Files).

Wie kann man sicher sein, alle zu erwischen?

Lösung: Man benutzt ein Tool, das die Abhängigkeiten eines C-Sourcefiles automatisch findet.

Z.B.

```
makedepend foo.c bar.c baz.c
```

Fügt am `Makefile` alle Abhängigkeiten der Files an, etwa

```
foo.o: baz.h foobar.h
baz.o: ../include/nix.h
```

5.4. Configurationsfiles erstellen

Problem: Software soll portabel sein

Man muß viele Architekturen & OSs unterstützen

Soll möglichst automatisch erfolgen

Lösung: man benutzt

- C-Präprozessor `#defines`'s (z.B. `#ifdef HAVE_VFPRINTF`)
- Variablenersetzungen in `Makefiles` (z.B. `$(RANLIB)`)

Um diese Informationen zu erhalten, läßt man ein Configurations-Script laufen:

- Dieses führt Tests auf die in Frage kommenden Eigenschaften aus
- erzeugt `Makefiles` aus Schablonen `Makefile.in`
- erzeugt ein Header-File `config.h` aus `config.h.in`

```
> ./configure --prefix=$HOME --enable-shared
...
creating Makfile
creating src/Makefile
creating config.h
```

Beispiel für ein `Makefile.in`

```
srcdir = @srcdir@
VPATH = @srcdir@

CC = @CC@

INSTALL = @INSTALL@
INSTALL_PROGRAM = @INSTALL_PROGRAM@
INSTALL_DATA = @INSTALL_DATA@

install: all
        ${INSTALL_PROGRAM} indent ${bindir}/gindent
```

Beispiel für ein `config.h.in`:

```
/* config.h.in. Generated automatically from configure.in by autoheader.*/

/* Define if on AIX 3.
   System headers sometimes define this.
   We just want to avoid a redefinition error message.  */
#ifndef _ALL_SOURCE
#undef _ALL_SOURCE
#endif

/* Define if using alloca.c.  */
#undef C_ALLOCA

/* Define to empty if the keyword does not work.  */
#undef const
```

Das `configure` ist ein normales Shell-Script.

Es wird aus einer Beschreibung `configure.in` generiert, in welcher verschiedene Tests (und normale Shell-Kommandos) angegeben werden.

Die Generierung erfolgt mittels `autoconf`.

Wirkungsweise: `m4` Macros werden expandiert. `sed` Kommandos für direkte Ersetzungen werden generiert (`Makefile.in` →`Makefile`)

`config.h.in` wird zu `config.h`

`autoheader` kann automatisch ein `config.h.in` aus den Tests in `configure.in` erzeugen!

Beispiel für ein `configure.in`:

```
dnl Process this file with autoconf to produce a configure script.
AC_INIT(indent.c)
AC_PROG_CC
AC_PROG_CPP
AC_PROG_INSTALL
AC_AIX
AC_ISC_POSIX
AC_REPLACE_FUNCS(memcpy)
AC_UNISTD_H
AC_DIR_HEADER
if test -z "$dirheader"; then
  AC_DEFINE(NODIR)
fi
AC_XENIX_DIR
AC_OUTPUT(makefile)
```

`autoconf` bietet unzählige Tests:

- Test auf Vorhandensein & Ort verschiedenster Utilities
- Test auf Libraries
- Test auf bestimmte Funktionen in Libs
- Systemaufruftest
- Größe von Datentypen
- Header-Files
- typedefs, Definitionen in Headern
- ...

`automake` geht noch einen Schritt über `autoconf` hinaus.

Hier wird auch die Schablone `Makefile.in` aus einer Shcablone `Makefile.am` erzeugt.

Vorteil: Viele Standard-Ziele werden automatisch in das Makefile aufgenommen.