Anwenderseminar

Das Programmierwerkzeug make

Günter Egerer
JSC
g.egerer@fz-juelich.de

make

- ist ein für die Programmentwicklung außerordentlich hilfreiches Unix-Programm.
- generiert selbsttätig Kommandos zur Erzeugung bestimmter Dateien (z.B. Aufruf des Compilers, um Objektdatei zu erzeugen, oder Aufruf des Binders (Linkers) zur Erzeugung eines ausführbaren Programmes).
- führt die generierten Kommandos automatisch aus.
- ist in der Lage, nach einer Modifikation an einem größeren Programmsystem, selbständig herauszufinden, welche Quelldateien neu übersetzt werden müssen.

Damit **make** z.B. ein ausführbares Programm erzeugen kann, benötigt es Informationen über das Programmsystem. Diese werden **make** in einer Beschreibungsdatei (*Makefile*) mitgeteilt. Ein Makefile kann folgende Informationen beinhalten:

- Beschreibung der gegenseitigen Abhängigkeiten, der an der Erzeugung einer Datei (z.B. eines ausführbaren Programmes) beteiligten Dateien.
- Kommandos für die Erzeugung dieser Dateien.
- Makrodefinitionen, z.B. um Compiler-Optionen festzulegen.
- Regeln, die festlegen, wie aus einer Datei mit einer bestimmten Endung (z.B. *.c*) eine Datei mit einer anderen Endung (z.B. *.o*) erzeugt werden kann (*Suffix rule*).
- evtl. **include**-Anweisungen

I. allg. braucht die Beschreibung durch den Makefile nicht vollständig zu sein, da eine Reihe von Abhängigkeiten und Kommandos bereits **make**-intern definiert sind. Ebenso gibt es **make**-interne Standarddefinitionen für Makros.

Beachte: Ein Makefile ist **kein** Programm! Die Abarbeitung eines Makefiles erfolgt nicht-prozedural. (Daher ist auch die Reihenfolge, in der Abhängigkeiten und Makrodefinitionen aufgeführt sind, in der Regel ohne Bedeutung.)

Dependency lines

$targets$**:** $prerequisites_{opt}$

dienen zur Beschreibung von Abhängigkeiten.

$targets$	Dateien, die erzeugt werden sollen.
$prerequisites$	Dateien, die benötigt werden, um die $target$-Dateien zu erzeugen.

Einer $dependency\ line$ folgen häufig Kommandos, die ausgeführt werden sollen, um aus den $prerequisites$ die $target$-Dateien zu erzeugen:

$targets$**:** $prerequisites_{opt}$
$tab\ command_1$
tab . . .
$tab\ command_n$

Beachte: Jede Befehls**zeile** ist so zu betrachten, als ob sie in einer eigenen Shell ausgeführt wird.

Die Bearbeitung mehrerer in einer $dependency\ line$ angegebener $prerequisites$ erfolgt in der Reihenfolge, in der sie aufgeführt sind.

Beispiele zur Benutzung von make:

☛ Beschreibungsdatei (*Makefile*): `bsp1.mk`

```
pgm: main.c vector.c
     cc -o pgm main.c vector.c
```

make -f bsp1.mk

```
cc -o pgm main.c vector.c
```

make -f bsp1.mk

```
make: 'pgm' is up to date.
```

touch vector.c

make -f bsp1.mk

```
cc -o pgm main.c vector.c
```

Nachteil: `main.c` wird erneut übersetzt, obwohl es nicht verändert wurde.

☛ bsp2.mk

```
pgm: main.o vector.o
     cc -o pgm main.o vector.o

main.o: main.c
     cc -c main.c

vector.o: vector.c
     cc -c vector.c
```

make -f bsp2.mk

```
cc -c main.c
cc -c vector.c
cc -o pgm main.o vector.o
```

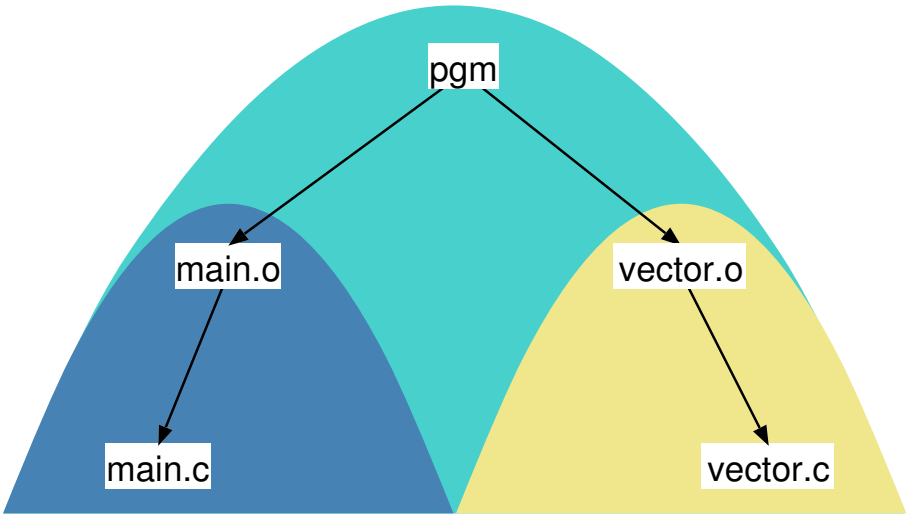
make -f bsp2.mk

```
make: 'pgm' is up to date.
```

touch vector.c

make -f bsp2.mk

```
cc -c vector.c
cc -o pgm main.o vector.o
```



Abarbeitung des Kommandos **make -f bsp2.mk**:

make pgm

→ make main.o

→ make main.c

Examining main.c...modified 9:55:22 Apr 23, 1997...
up-to-date.
Examining main.o...modified 16:43:46 Apr 23, 1997...
up-to-date.

→ make vector.o

→ make vector.c

Examining vector.c...modified 11:44:47 Apr 24, 1997...
up-to-date.
Examining vector.o...modified 16:45:12 Apr 23, 1997...
modified before source...out-of-date.

```
cc -c vector.c
```

update time: 11:45:03 Apr 24, 1997

Examining pgm...modified 16:45:13 Apr 23, 1997...modified before source...out-of-date.

```
cc -o pgm main.o vector.o
```

make -f bsp2.mk main.o

```
make: 'main.o' is up to date.
```

make -f bsp2.mk vector.o

```
make: 'vector.o' is up to date.
```

touch vector.c

make -f bsp2.mk main.o vector.o

```
make: 'main.o' is up to date.
cc -c vector.c
```

mv main.c main.c.bak

make -f bsp2.mk

```
make: *** No rule to make target 'main.c',
  needed by 'main.o'.  Stop.
```

mv main.c.bak main.c

☛ bsp3.mk

```
pgm: main.o vector.o
     cc -o pgm main.o vector.o

main.o: main.c
     cc -c main.c

vector.o: vector.c
     cc -c vector.c

.PHONY:[1] clean
clean:
     rm -f *.o core
```

make -f bsp3.mk

```
cc -o pgm main.o vector.o
```

make -f bsp3.mk

```
make: 'pgm' is up to date.
```

make -f bsp3.mk clean

```
rm -f *.o core
```

make -f bsp3.mk

```
cc -c main.c
cc -c vector.c
cc -o pgm main.o vector.o
```

[1] Nur bei GNU make: Ein *target*, das keinen Dateinamen repräsentiert, sollte als *phony* vereinbart werden.

Makros

Makrodefinition

$name$=$string$

Makroaufruf

$(**$name$)** oder **${**$name$**}**

Eine Makrodefinition darf Makroaufrufe enthalten. Die Definition eines Makros darf jedoch keine Bezugnahme auf sich selbst beinhalten. Die folgende Definition des Makros `CCOPTS` ist daher unzulässig.

```
CCOPTS = -DFIND2
DEBUGOPTS = -g
CCOPTS = $(CCOPTS) $(DEBUGOPTS)
```

Die den Namen umschließenden Klammern dürfen weggelassen werden, falls der Name nur aus einem Zeichen besteht:

`$X` $\Leftrightarrow$ `$(X)`

Möglichkeiten zur Definition von Makros:

(Reihenfolge entspricht der Priorität:

1 = höchste Priorität

4 = geringste Priorität

Die Angaben in Klammern gelten für **make**-Aufrufe mit der Option -**e**.)

1. (1.) Makrodefinition auf der Kommadozeile (**make**-Aufruf)
2. (3.) Makrodefinition innerhalb der Beschreibungsdatei (Makefile)
3. (2.) Shell-Variablen, z.B. $(HOME)
4. (4.) interne Standarddefinitionen von **make**

Der Aufruf eines Makros, für das keine Definition existiert, wird durch einen leeren String ersetzt.

☛ bsp4.mk

```
CCOPTS = -g -Wall
#CCOPTS = -O

pgm:    main.o vector.o
     cc -o pgm main.o vector.o

main.o: main.c
     cc $(CCOPTS) -c main.c

vector.o: vector.c
     cc $(CCOPTS) -c vector.c

clean:
     rm -f *.o core
```

make -f bsp4.mk clean pgm

```
rm -f *.o core
cc -g -Wall -c main.c
...
```

make -f bsp4.mk CCOPTS="-ansi -g" clean pgm

```
rm -f *.o core
cc -ansi -g -c main.c
...
```

export CCOPTS=-O; make -f bsp4.mk -e clean pgm

```
rm -f *.o core
cc -O -c main.c
...
```

interne Makros

$? Liste der *prerequisites*, die neueren Datums als das aktuelle *target* sind. (In Kommandos einer *suffix rule* nicht zulässig.)

$@ Name des aktuellen *targets*.

$$@ Name des aktuellen *targets*. (Nur rechts vom Doppelpunkt innerhalb einer *dependency line* erlaubt.)

Beispiel:
```
.SECONDEXPANSION:[2]
prg: $$@.c
```

$< aktueller *prerequisite*-Name. Es handelt sich um den Namen einer Datei, die neueren Datums als das aktuelle *target* ist.
(nur in *suffix rules* und im **.DEFAULT**-Abschnitt!)

$* aktueller *prerequisite*-Name ohne Suffix.
(nur in *suffix rules*!)

$% steht für *member*.**o**, sofern das aktuelle *target* die Form *libname*(*member*.**o**) hat. (*target* ist in diesem Fall eine in einer Bibliothek enthaltene Objektdatei. **$@** expandiert hier zu *libname*.)

$$ das Zeichen $

Einige Varianten von **make** erlauben die Verwendung dieser Makros mit angehängtem **D** oder **F**:

D (***d****irectory part*) Pfadname der jeweiligen Datei

F (***f****ile name part*) Dateiname ohne Pfadangabe

Verwendung nur in Verbindung mit Klammern: z.B. `${@F}`

[2] Bei GNU make muss diese Zeile der Verwendung von **$$@** vorausgehen.

☛ bsp5.mk

```
CCOPTS = -g -Wall
#CCOPTS = -O

pgm:    main.o vector.o
     cc -o $@ main.o vector.o

main.o: main.c
     cc $(CCOPTS) -c $?

vector.o: vector.c
     cc $(CCOPTS) -c $?

clean:
     rm -f *.o core
```

make -f bsp5.mk clean

```
rm -f *.o core
```

make -f bsp5.mk

```
cc -g -Wall -c main.c
cc -g -Wall -c vector.c
cc -o pgm main.o vector.o
```

Makro String Substitution

nicht bei allen Varianten von **make** verfügbar.

${*macroname*:*s1*=*s2*}

Beachte: Ersetzungen finden nur am Ende von (durch Zwischenraum oder Zeilenende begrenzten) Wörtern statt.

☛ bsp6.mk

```
CCOPTS = -g -Wall
#CCOPTS = -O

SRCS = main.c vector.c
# ----------------------------------------
OBJS = ${SRCS:.c=.o}

pgm:    $(OBJS)
     cc -o $@ $(OBJS)

main.o: main.c
     cc $(CCOPTS) -c $?

vector.o: vector.c
     cc $(CCOPTS) -c $?

clean:
     rm -f *.o core
```

make -f bsp6.mk clean

```
rm -f *.o core
```

make -f bsp6.mk

```
cc -g -Wall -c main.c
cc -g -Wall -c vector.c
cc -o pgm main.o vector.o
```

Suffix rules

.$suffix_1$.$suffix_2$:

Eine *suffix rule* teilt **make** mit, dass eine Datei

$name$.$\mathbf{suffix_1}$

prerequisite einer Datei $name$.$\mathbf{suffix_2}$ sein kann (implizite Abhängigkeit).

Es sollte mindestens ein Kommando folgen, das auszuführen ist, um aus der Datei $name$.$\mathbf{suffix_1}$ die Datei $name$.$\mathbf{suffix_2}$ zu erzeugen.

Beispiel:

```
.c.o:
tab$(CC) $(CFLAGS) -c $<
```

Pattern rules (nur GNU make)

- gut lesbar und vielseitig einsetzbar (z.B. als Alternative zu *Suffix rules*)
- *target* enthält genau ein Prozentzeichen, das für einen beliebigen (nicht-leeren) Text steht.

Beispiel:

```
%_dbg.o: %.c
tab$(CC) $(CFLAGS) $(DBGFLAGS) -c $< -o $@
```

☛ bsp7.mk

```
CCOPTS = -g -Wall
#CCOPTS = -O

SRCS = main.c vector.c
# ----------------------------------------------
OBJS = ${SRCS:.c=.o}

.c.o:
     cc $(CCOPTS) -c $<

pgm:    $(OBJS)
     cc -o $@ $(OBJS)

clean:
     rm -f *.o core
```

make -f bsp7.mk clean

```
rm -f *.o core
```

make -f bsp7.mk

```
cc -g -Wall -c main.c
cc -g -Wall -c vector.c
cc -o pgm main.o vector.o
```

make-Standardregeln erfragen:

make -p >make.defaults

```
make: *** No targets specified and no makefile
  found.  Stop.
```

cat make.defaults

```
...
# Variables
...
OUTPUT_OPTION = -o $@
...
CC = cc
...
COMPILE.c = \
  $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
...
# Implicit Rules
...
.c.o:
        $(COMPILE.c) $(OUTPUT_OPTION) $<
...
```

☛ bsp8.mk

```
CFLAGS = -g -Wall
#CFLAGS = -O

SRCS = main.c vector.c
# ----------------------------------------
OBJS = ${SRCS:.c=.o}

pgm:    $(OBJS)
     $(CC) -o $@ $(OBJS)

clean:
     rm -f *.o core
```

touch main.c

make -f bsp8.mk

```
cc -g -Wall   -c -o main.o main.c
cc -o pgm main.o vector.o
```

make -f bsp8.mk clean

```
rm -f *.o core
```

make -f bsp8.mk CFLAGS=-O "CC = icc"

```
icc -O   -c -o main.o main.c
icc -O   -c -o vector.o vector.c
icc -o pgm main.o vector.o
```

Beachte: Eine Aktualisierung des Makefiles bzw. geänderte Makrodefinitionen führen nicht dazu, dass *targets* erneut erzeugt werden.

touch bsp8.mk
make -f bsp8.mk CFLAGS=-g

```
make: 'pgm' is up to date.
```

Abhängigkeiten von Header-Dateien werden noch nicht berücksichtigt:

grep 'vector.h' *.c

```
main.c:#include "vector.h"
vector.c:#include "vector.h"
```

touch vector.h
make -f bsp8.mk

```
make: 'pgm' is up to date.
```

Abhängigkeiten von Header-Dateien hinzufügen:

☛ `bsp9.mk`

```
CFLAGS = -g -Wall
#CFLAGS = -O

SRCS = main.c vector.c
# ----------------------------------------
OBJS = ${SRCS:.c=.o}

pgm:    $(OBJS)
     $(CC) -o $@ $(OBJS)

main.o vector.o: vector.h
# oder umstaendlicher:
# main.c vector.c: vector.h
#       touch $@

clean:
     rm -f *.o core
```

make -f bsp9.mk

```
cc -g -Wall   -c -o main.o main.c
cc -g -Wall   -c -o vector.o vector.c
cc -o pgm main.o vector.o
```

Abhängigkeiten können mit **cc**-Option -**M** oder **makedepend** automatisch generiert werden.

☛ bsp10.mk

```
CFLAGS = -g -Wall
#CFLAGS = -O

SRCS = main.c vector.c
# -----------------------------------------
OBJS = ${SRCS:.c=.o}

pgm:    $(OBJS)
     $(CC) -o $@ $(OBJS)

clean:
     rm -f *.o core

depend:
     makedepend -- $(CFLAGS) -- $(SRCS)
```

cp bsp10.mk makefile; make depend

```
   makedepend -- -g -Wall -- main.c vector.c
```

diff makefile bsp10.mk

```
< # DO NOT DELETE
<
< main.o: /usr/include/stdio.h /usr/include/feature\
s.h /usr/include/sys/cdefs.h
< main.o: /usr/include/gnu/stubs.h
...
< main.o: vector.h
...
```

Single-suffix rules

nicht bei allen Varianten von **make** verfügbar.

.*suffix*:

Eine solche Regel spezifiziert, wie aus einer Datei *name*.***suffix*** eine Datei *name* erzeugt werden kann.

Beispiel:

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

Da es sich bei dieser Regel um eine interne Standardregel handelt, wird kein Makefile benötigt, um aus einer Quelldatei *name*.**c** ein ausführbares Programm *name* zu erzeugen. Es genügt das Kommando

make *name*

☛ Beispiel 11

ls makefile Makefile prog*

```
ls: makefile: No such file or directory
ls: Makefile: No such file or directory
prog.c  prog.f
```

make prog

```
cc   prog.c -o prog
```

Spezielle $target$-Namen

.SUFFIXES: **.**$suffix_{1\,opt}$ **.**$suffix_{2\,opt}$ . . . **.**$suffix_{n\,opt}$

erweitert die Liste der $suffixes$, die von **make** als signifikant betrachtet werden und somit in $suffix\ rules$ verwendet werden können, um die hier angegebenen. Falls keine $suffixes$ angegeben sind, wird die Liste der $suffixes$ gelöscht.
(Die Reihenfolge der $suffixes$ ist insofern von Bedeutung, als durch sie festgelegt wird, welche implizite Abhängigkeit angenommen wird, wenn mehrere $suffix\ rules$ gleichenmaßen anwendbar sind.)

.DEFAULT:

die mit **.DEFAULT** verbundenen Befehle werden immer dann ausgeführt, wenn ein $target$ (, das ausschließlich als $prerequisite$ verwendet wird), nicht auf andere Weise erzeugt werden kann.

.SILENT: $targets_{opt}$

die zur Erzeugung der angegebenen $targets$ dienenden Kommandos werden bei ihrer Ausführung nicht auf der Standardausgabe angezeigt. Falls keine $targets$ spezifiziert sind, gilt dies für alle Kommandos des Makefiles. Dies entspricht der Option **-s**. Alternativ kann einzelnen Kommandozeilen das Zeichen **@** vorangestellt werden, um das Auflisten zu unterdrücken.

.IGNORE: $targets_{opt}$

veranlasst **make** alle Fehler zu ignorieren, die bei der Ausführung von Kommandos auftreten, welche zur Erzeugung der angegebenen $targets$ dienen. Falls keine $targets$ angegeben sind, erstreckt sich die Wirkung auf alle Kommandos des Makefiles. Dies entspricht der Option **-i**. Sollen Fehler nur bei einzelnen Kommandos ignoriert werden, kann einer Kommandozeile das Zeichen – vorangestellt werden.

.PRECIOUS: $targets_{opt}$

die angegebenen $target$-Dateien sollen erhalten bleiben, falls **make** (z.B. durch <Ctrl>-c) abgebrochen wird. Sind keine $targets$ angegeben, dann bezieht sich **.PRECIOUS** auf alle $targets$ des Makefiles.

Reihenfolge der *suffixes* verändern:

.f-Datei soll vor **.c**-Datei in Betracht gezogen werden.

☛ `makefile` (Beispiel 12)

```
.SUFFIXES:
.SUFFIXES: .f .c .o

# unter IBM AIX - jedoch nicht bei DEC, Sun,
# SGI, Cray oder GNU make(gmake) - zusaetzlich
# erforderlich:
.c:
     $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
.f:
     $(FC) $(FFLAGS) $(LDFLAGS) $< -o $@
```

ls prog*

```
prog.c  prog.f
```

make prog

```
f77    prog.f   -o prog
```

rm prog
mv prog.f prg.f
make prog

```
cc     prog.c   -o prog
```

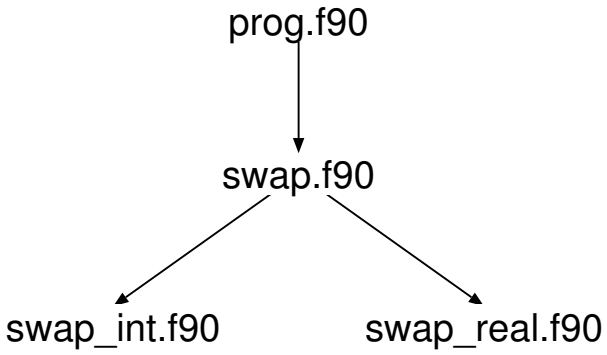
mv prog.c prg.c

neuen *suffix* **.f90** hinzufügen:

☛ `makefile` (Beispiel 13)

```
F90FLAGS = -g

SRCS = prog.f90 swap.f90 swap_int.f90 \
       swap_real.f90
# -----------------------------------------
OBJS = ${SRCS:.f90=.o}

.SUFFIXES: .f90

F90C = f90
.f90.o:
     $(F90C) $(F90FLAGS) -c $<

prog: $(OBJS)
     $(F90C) -o $@ $(OBJS)

clean:
     rm -f *.o core
# -----------------------------------------
# Modul-Abhaengigkeiten:

prog.o: swap.o
swap.o: swap_int.o swap_real.o
```

Modul-Abhängigkeiten:



make

```
f90 -g -c swap_int.f90
f90 -g -c swap_real.f90
f90 -g -c swap.f90
f90 -g -c prog.f90
f90 -o prog prog.o swap.o swap_int.o swap_real.o
```

☛ `makefile` (Beispiel 14)

```
...
.SUFFIXES: .f90

F90C = f90
.f90.o:
     $(F90C) $(F90FLAGS) -c $<

.DEFAULT:
     @echo "searching for $< ..."
     @result=‘find . -name $< -print‘; \
      if test ‘echo $$result | wc -w‘ -eq 1;\
      then \
        ln -s $$result $< ; \
        echo " ln -s $$result $<"; \
      else \
        echo "Don’t know how to make ’$<’.";\
        echo -e "files which may be used "\
          "instead:\n$$result"; \
      fi
     @test -f $<

prog: $(SRCS) $(OBJS)
     $(F90C) -o $@ $(OBJS)
...
```

mkdir backup
cp *.f90 backup/
rm swap.f90
make

```
searching for swap.f90 ...
        ln -s ./backup/swap.f90 swap.f90
f90 -g -c swap.f90
f90 -g -c prog.f90
f90 -o prog prog.o swap.o swap_int.o swap_real.o
```

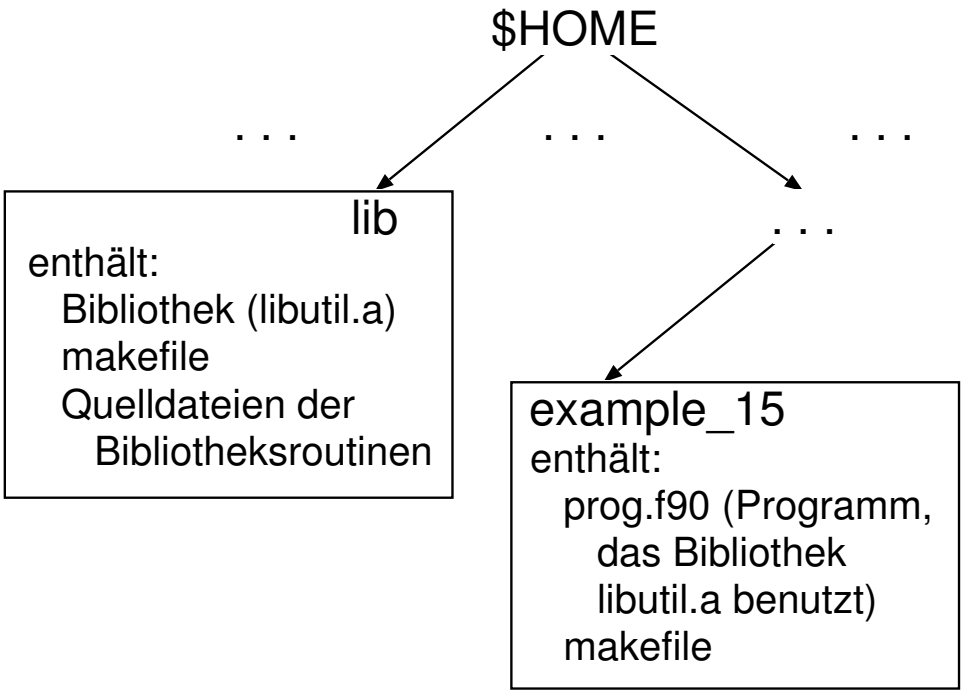
rm swap.f90
mkdir tmp
cp backup/swap.f90 tmp/
make

```
searching for swap.f90 ...
Don't know how to make 'swap.f90'.
files which may be used  instead:
./tmp/swap.f90
./backup/swap.f90
make: *** [swap.f90] Error 1
```

rekursive Verwendung von make

- ist hilfreich bei einem größeren Programmsystem, dessen Dateien auf mehrere Verzeichnisse (*Directories*) verteilt sind (→ Beispiel 15),
- oder zum Angeben bzw. Verändern von Makrodefinitionen (→ Beispiel 16).

☛ Beispiel 15



./makefile

```
F90FLAGS = -g -I$(LIBDIR)

SRCS   = prog.f90
LIBDIR = $(HOME)/lib
LIB    = $(LIBDIR)/libutil.a
# ----------------------------------------
OBJS = ${SRCS:.f90=.o}

.SUFFIXES: .f90

F90C = f90
.f90.o:
     $(F90C) $(F90FLAGS) -c $<

all:    library prog
     @echo "'prog' is now up-to-date."

prog:   $(OBJS)
     $(F90C) -o $@ $(OBJS) -L$(LIBDIR) -lutil

library:
     cd $(LIBDIR); $(MAKE)
# ----------------------------------------
# Modul-Abhaengigkeiten:

prog.o: $(LIB)(swap.o)
```

$HOME/lib/makefile

```
F90FLAGS = -O

LIB = libutil.a
# ----------------------------------------
.SUFFIXES: .f90

F90C = f90
# durch folgende Regel ausgedrueckte implizite
# Abhaengigkeit:
# libname(member.o): member.f90
#  $@ = libname      $< = member.f90
#  $% = member.o     $* = member
.f90.a:
     $(F90C) $(F90FLAGS) -c $<
     ar rv $@ $%
     rm -f $%

$(LIB): $(LIB)(swap.o) $(LIB)(match.o)
     @echo "'$@' is now up-to-date."

# ----------------------------------------
# Modul-Abhaengigkeiten:

$(LIB)(swap.o): $(LIB)(swap_int.o)
$(LIB)(swap.o): $(LIB)(swap_real.o)
```

ls $HOME/lib

```
makefile        swap.f90        swap_real.f90
match.f90       swap_int.f90
```

make

```
cd /home/egerer/lib; make
make[1]: Entering directory ‘/home/egerer/lib’
f90 -O -c swap_int.f90
ar rv libutil.a swap_int.o
a - swap_int.o
rm -f swap_int.o
f90 -O -c swap_real.f90
ar rv libutil.a swap_real.o
a - swap_real.o
rm -f swap_real.o
f90 -O -c swap.f90
ar rv libutil.a swap.o
a - swap.o
rm -f swap.o
f90 -O -c match.f90
ar rv libutil.a match.o
a - match.o
rm -f match.o
’libutil.a’ is now up-to-date.
make[1]: Leaving directory ‘/home/egerer/lib’
f90 -g -I/home/egerer/lib -c prog.f90
f90 -o prog prog.o -L/home/egerer/lib -lutil
’prog’ is now up-to-date.
```

make

```
cd /home/egerer/lib; make
make[1]: Entering directory ‘/home/egerer/lib’
make[1]: ‘libutil.a’ is up to date.
make[1]: Leaving directory ‘/home/egerer/lib’
’prog’ is now up-to-date.
```

(cd $HOME/lib; touch swap_real.f90)
make

```
cd /home/egerer/lib; make
make[1]: Entering directory ‘/home/egerer/lib’
f90 -O -c swap_real.f90
ar rv libutil.a swap_real.o
r - swap_real.o
rm -f swap_real.o
f90 -O -c swap.f90
ar rv libutil.a swap.o
r - swap.o
rm -f swap.o
’libutil.a’ is now up-to-date.
make[1]: Leaving directory ‘/home/egerer/lib’
f90 -g -I/home/egerer/lib -c prog.f90
f90 -o prog prog.o -L/home/egerer/lib -lutil
’prog’ is now up-to-date.
```

(cd $HOME/lib; touch match.f90)
make

```
cd /home/egerer/lib; make
make[1]: Entering directory ‘/home/egerer/lib’
f90 -O -c match.f90
ar rv libutil.a match.o
r - match.o
rm -f match.o
’libutil.a’ is now up-to-date.
make[1]: Leaving directory ‘/home/egerer/lib’
’prog’ is now up-to-date.
```

touch prog.o
make

```
cd /home/egerer/lib; make
make[1]: Entering directory ‘/home/egerer/lib’
make[1]: ‘libutil.a’ is up to date.
make[1]: Leaving directory ‘/home/egerer/lib’
f90 -o prog prog.o -L/home/egerer/lib -lutil
’prog’ is now up-to-date.
```

☛ `makefile` (Beispiel 16)

```
CFLAGS = -g -Wall
#CFLAGS = -O

# ----------------------------------------
SRCS = `echo *.c`
# Aufruf von Shell: echo '$(SRCS)' -> `echo *.c`
# rekursiver Aufruf: echo '$(SRCS)' -> main.c
#                                      vector.c
OBJS = ${SRCS:.c=.o}
# Aufruf von Shell: echo '$(OBJS)' -> `echo *.c`
# rekursiver Aufruf:
#   echo '$(OBJS)' -> main.o vector.o

all:
     @$(MAKE) "SRCS = $(SRCS)" pgm

pgm: $(OBJS)
     $(CC) -o $@ $(OBJS)

clean:
     rm -f *.o core

depend:
     makedepend -- $(CFLAGS) -- $(SRCS)
```

Spezielle Makros

Beachte: Die Handhabung der folgenden Makros kann bei verschiedenen **make**-Implementierungen unterschiedlich sein.

MAKE wird durch **make** gesetzt und enthält den Namen des Kommandos zum Aufruf von **make**.

MAKEFLAGS wird durch **make** gesetzt und enthält bestimmte Optionen, die in der Kommandozeile des gerade aktuellen **make**-Aufrufs angegeben sind.

SHELL bestimmt die Kommando-Shell, die aus **make** heraus gestartet wird, um Befehle auszuführen. Falls das Makro nicht im Makefile definiert wird, übernehmen einige **make**-Varianten die Definition der entsprechenden Shell-Variable, andere benutzen (zumindest für *top-level*-Aufrufe) standardmäßig die Bourne-Shell (`/bin/sh`).

VPATH Spezifiziert eine Liste von (durch **:** getrennten) Verzeichnissen, in denen nach *prerequisite*-Dateien gesucht werden soll, wenn diese nicht im aktuellen Verzeichnis vorliegen.

Standard-Optionen

-d (***d****ebug*) veranlasst die Ausgabe detaillierter Informationen (z.B. betrachtete Dateien und deren Modifikationszeiten), die es erlauben, die von **make** ausgeführten Aktionen zu verfolgen.

-e (***e****nvironment*) Shell-Variablen sollen Makrodefinitionen innerhalb des Makefiles überschreiben.

-f *file* spezifiziert eine Beschreibungsdatei (Standardvorgaben: `makefile`, `Makefile`). Die Option kann mehrfach angegeben werden. In diesem Fall bilden die in der Reihenfolge ihres Auftretens aneinandergefügten Dateien den Makefile. Soll der Makefile von **stdin** gelesen werden, dann ist für *file* ein Minuszeichen (-) anzugeben.

-i (***i****gnore*) gleiche Wirkung wie **.IGNORE**-*target*: Fehler (Kommandos mit Exit-Status $\neq$ 0) sollen nicht zum Abbruch führen.

-k nützlich bei mehreren *targets* auf der Kommandozeile: Tritt bei der Erzeugung eines *targets* ein Fehler auf, werden die folgenden *targets* trotzdem noch erzeugt.

-n listet alle generierten Kommandos auf, ohne sie auszuführen. (Manche Varianten von **make** führen mit dem Zeichen + beginnende Kommandozeilen trotzdem noch aus.)

-p alle Makrodefinitionen, die Liste der *suffixes*, *suffix rules* und *target*-Beschreibungen sollen aufgelistet werden.

-q (***q****uestion*) veranlasst **make** über den Exit-Status anzuzeigen, ob eine *target*-Datei neu erzeugt werden muss (Exit-Status $\neq$ 0) oder nicht (Exit-Status 0). (Die *target*-Datei wird dabei nicht neu generiert. Mit **+** beginnende Kommadozeilen werden jedoch ausgeführt.)

-r **make**-interne Abhängigkeiten (Standardregeln) sollen ignoriert werden.

-s (***s****ilent*) gleiche Wirkung wie **.SILENT**-*target*: ausgeführte Kommandos sollen nicht aufgelistet werden.

-t (***t****ouch*) *target*-Dateien erhalten neues Änderungsdatum, ohne dass irgendwelche weiteren Kommandos ausgeführt werden. (Mit **+** beginnende Kommadozeilen werden trotzdem ausgeführt.)

f90depend

- Tool zur Analyse von Abhängigkeiten und zur automatischen Erzeugung von Makefiles für Fortran90/95-Quellprogramme (nur *free source form*)
- Eigenentwicklung des JSC (Basis: Scanner/Lexer des NAGWare f90 Compilers)
- Verfügbarkeit:
  - Linux-Systeme, AIX (IBM), ...
- Anwendungsmöglichkeiten:

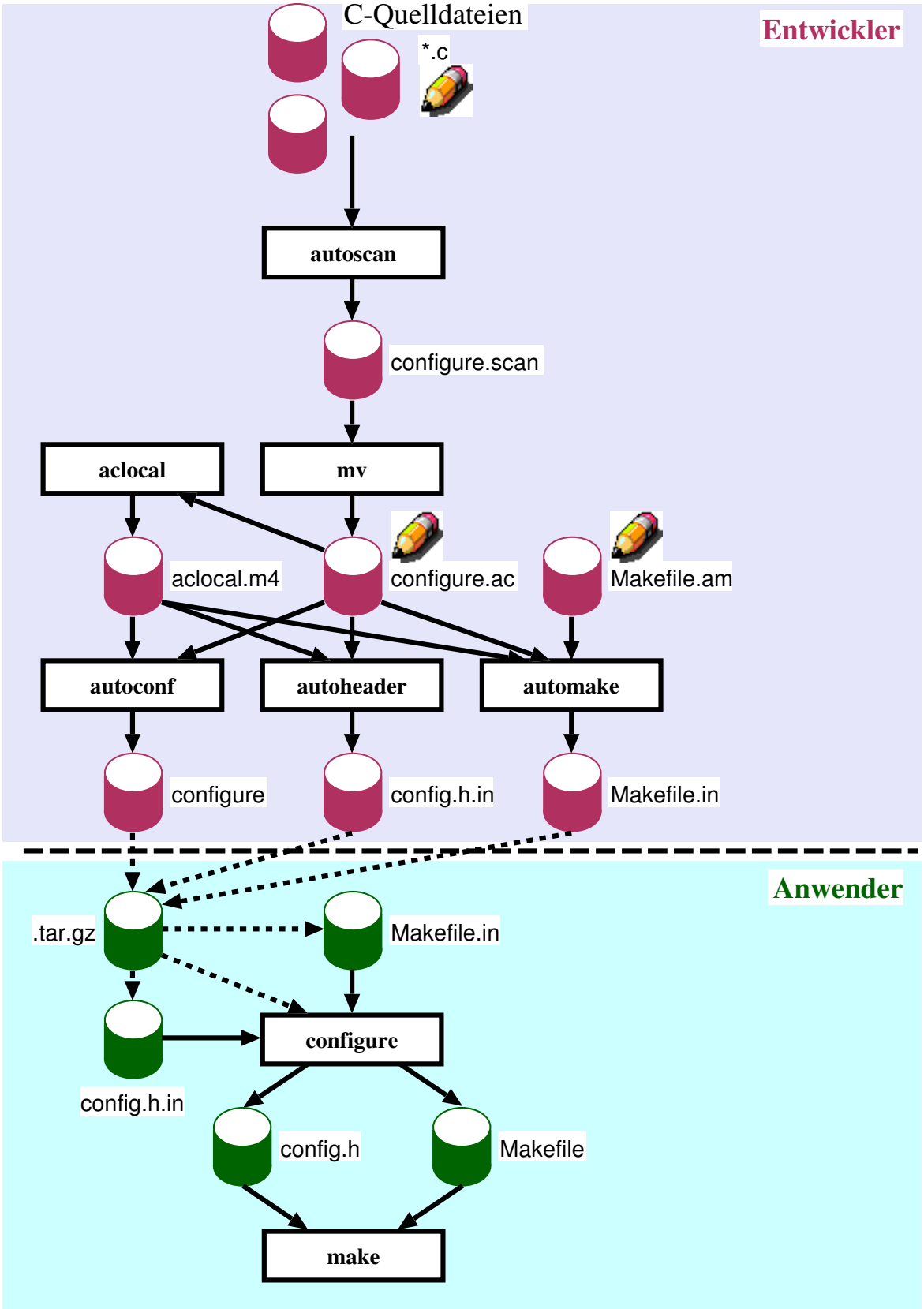
  -u Erzeugen von Abhängigkeiten in Makefiles (ähnlich wie **makedepend**)

  -c Automatische Generierung von einfachen Makefiles für die Erzeugung ausführbarer Programme

  -s Bestimmen einer Reihenfolge, in der die Quelldateien eines Programmsystems übersetzt werden können.
- weitere Informationen: **www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/f90depend/_node.html**

  man f90depend

Das GNU-Build-System



Beispiel zur Benutzung:

`Makefile.am`

```
bin_PROGRAMS = pgm

pgm_SOURCES = main.c vector.c vector.h

## The 'include_HEADERS' line lists  p u b l i c
## headers present in this directory that you
## want to install in /prefix/include.

## include_HEADERS = vector.h
```

`pgm` ist der frei wählbare Name des ausführbaren Programms. Dieses wird beim Installieren mit **make install** in das Verzeichnis `/usr/local/bin` (bzw. */prefix*`/bin`) kopiert.

- `configure.ac` erstellen:

  ls *.c *.h

  ```
  main.c  vector.c  vector.h
  ```

  autoscan
  mv configure.scan configure.ac

- `configure.ac` editieren (Die grau hinterlegte Information wird hinzugefügt):

```
...
AC_PREREQ(2.59)
## AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT
-ADDRESS)
AC_INIT(mytool, 0.0, g.egerer@fz-juelich.de)

AM_INIT_AUTOMAKE([foreign])
AC_CONFIG_SRCDIR([vector.h])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC
...
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

`AM_INIT_AUTOMAKE` ist Voraussetzung für die Verwendung von **automake**. `foreign` gibt an, dass bestimmte Dokumentationsdateien (NEWS, README, AUTHORS, ChangeLog), die in einem wirklichen GNU-Distributionspaket vorhanden sein müssen, im aktuellen Verzeichnis fehlen dürfen.

- Dateien für den Anwender erzeugen:

  aclocal
  autoconf
  autoheader
  automake -a

  Die Option -a zeigt an, dass fehlende Hilfsdateien zum Verzeichnis hinzugefügt werden sollen.

  ```
  configure.ac:7: installing ‘./missing’
  configure.ac:7: installing ‘./install-sh’
  Makefile.am: installing ‘./depcomp’
  ```

- `Makefile` und ausführbares Programm generieren und testen:

  ./configure
  make
  ./pgm

- Distributionsdatei (`mytool-0.0.tar.gz`) erzeugen:

  make distcheck

  Hierbei wird auch überprüft, ob die Distribution entpackt und installiert werden kann.