Einführung in Make

Quelle: http://www.comnets.rwth-aachen.de/doc/einmanual/shortintro_toc.html#SEC47
(aber dort nicht mehr verfügbar)

Programm-Verwaltung mit make

Autor : Matthias Froelich

Einleitung

Das Programm `make` erleichert die Entwicklung von Programmen. Es erlaubt, bei Aenderungen in einer Datei, automatisch die davon betroffenen Programmteile neu zu kompilieren und zu binden. Dazu generiert es anhand einer Beschreibungsdatei (meist mit Namen `makefile`) automatisch die notwendigen Kommandos und fuehrt diese aus.

Die im folgenden benutzten Beispiele sind im wesentlichen der Datei

- `/project/mobilise/examples/make/makefile` (1)

entnommen. Sie wird als Vorlage fuer die Entwicklung des **Mobilise**- Demonstrators verwendet und enthaelt Kommentare, die diesen Text ergaenzen. Im gleichen Verzeichnis befinden sich auch Beispiel-Dateien, anhand derer das unten beschriebene Beispiel nachvollziehbar sein sollte. Die Datei make.README enthaelt Hinweise neueren Datums, die in diesen Text nicht mehr aufgenommen werden konnten.
Bei Unstimmigkeiten und Verbesserungsvorschlaegen bzgl. dieser Beschreibung und der Beispiel-Dateien bitte eine Nachricht an `maf`.

Die im folgenden beschriebene Struktur wurde in Analogie zu dem ueblicherweise in Verbindung mit der `CNCL` (2) verwendeten `makefile` angelegt. Insoweit sollte dieser Text auch dafuer zutreffen.

Aufruf von make

Der Befehl zum Aufruf von make lautet im wesentlichen (3):

- `make [`**`Optionen`**`]` **`Ziel`**

Beispiel:

- `make -n login.o`

Die Option `-n` veranlasst `make` die Befehle zum Erreichen des **Ziels** `login.o` nur zu generieren und anzuzeigen, aber nicht direkt auszufuehren. Diese Option ist nuetzlich zum Debuggen von `makefile`s.

Das Programm `make` sucht jetzt im aktuellen Verzeichnis nach der Datei `makefile` und sucht dort nach einem Eintrag aus dem hervorgeht, wie zu dem angegebenen **Ziel** gelangt werden kann. Die Datei `makefile` ist eine ASCII-Textdatei, die vom Benutzer (z.B. mit `emacs`) erstellt wird. Zeilen mit einem `#` Zeichen am Anfang sind Kommentarzeilen. Ist das letzte Zeichen einer Zeile ein \ so wird diese fortgesetzt.

Das Ziel des makefiles

Herzstueck eines jeden `makefile` sind Eintraege der folgenden Form:

- `Ziel: Abhaengigkeit(en)`
- `<TAB> Regel`

Beispiel:

```
login.o:      login.c login.h announce.h
          gcc -ansi -c login.c
```

Die drei Teile eines solchen Eintrages koennen kurz als **was**, **warum** und **wie** bezeichnet werden:

- *Ziel*: **Was** von `make` neu angefertigt werden soll. In diesem Fall die Datei `login.o` (auch neudeutsch als **Target** bezeichnet).
- *Abhaengigkeit(en)*: **Warum** `login.o` neu angefertigt werden muss. Falls sich also an den Dateien `login.c`, `login.h` oder `announce.h` etwas geaendert hat geht `make` davon aus, dass das **Ziel** `login.o` neu zu erzeugen ist. Dazu vergleicht `make` einfach das Erstellungsdatum von `login.o` mit dem jeder Datei aus den **Abhaengigkeiten** (neudeutsch: *Dependencies*).
- *Regel*: **Wie** aus den **Abhaengigkeiten** das **Ziel** erzeugt werden kann (auch **Rule** genannt). In diesem Fall also durch Aufruf des `gcc` Compilers fuer die Datei `login.c`. Vermutlich werden in `login.c` die Dateien `login.h` und `announce.h` eingebunden. **Achtung!:** Es ist unabdingbar, dass diese Zeile (und ggf. jede Folgezeile!) tatsaechlich mit einem Tabulator-Zeichen <TAB> anfaengt.

Jede **Abhaengigkeit** ist fuer `make` selbst wieder ein moegliches **Ziel**. Daher wird `makefile` so lange ausgewertet, bis alle **Abhaengigkeiten** erkannt sind. Ein weiterer moeglicher Eintrag waere z.B.:

```
login:     login.o announce.o
         gcc -o login login.o announce.o
```

Jetzt wird mit dem Aufruf `make login` das Programm `login` aus den Objektdateien `login.o` und `announce.o` neu zusammengebunden, falls mindestens eine dieser Dateien juengeren Datums als `login` ist. Sollte auch `login.o` aelter als eine seiner **Abhaengigkeiten** sein wuerde es zuerst neu erzeugt.

Bei diesem Beispiel bleibt nur die Frage, wie `make` das **Ziel** `announce.o` herstellen wuerde, falls es noch nicht vorhanden waere? Schau'n wir mal!

1.1.1 Die voreingestellten Regeln

Um nicht fuer jede Datei einen eigenen Eintrag im `makefile` vornehmen zu muessen, besitzt `make` zahlreiche vorgegebene **Regeln**, wie von einer Datei zu einer anderen gelangt werden kann. Dabei orientiert sich make an den Endungen (suffix) der Dateien. Die Voreinstellungen koennen auch im `makefile` selbst durch Eintraege der folgenden Art umgesetzt werden:

```
.c.o:
        gcc -ansi -c $<
```

Hiermit wird eine allgemeine **Regel** (4) angegeben, wie ein **Ziel** mit Endung `.o` aus einer Datei mit **gleichem** Namenspraefix und der Endung `.c` hergestellt wird. Diese **Regel** wird nur angewandt, falls es keinen expliziten Eintrag fuer das gewuenschte **Ziel** gibt. Im obigen Beispiel also beim Aufruf von `make announce.o`. Ausdruecke wie `$<` haben fuer `make` eine beondere Bedeutung, in diesem Fall der Praefix des **Ziels** plus die Endung der **Abhaengigkeit** (5).

Da es eine enorme Menge von vordefinierten sog. **Suffix-Regeln** existiert, kann es leicht zu Konflikten mit den Voreinstellungen kommen. Daher wird haeufig die Liste der von `make` erkannten Endungen

zuerst geloescht und dann neu definiert. Dies geschieht durch Definition des voreingestellten **Ziels** `.SUFFIXES` wobei die von `make` zu erkennenden Endungen danach eingegeben werden und keine **Regel** angegeben wird:

- `.SUFFIXES:`
- `.SUFFIXES: .c .o`

Damit wuerde `make` nur noch die zwei angegebenen Endungen beruecksichtigen. In `makefile` sind zwei Regeln definiert, die automatisch aus einer `.c` bzw `.cc` Datei ein ausfuehrbares Programm erzeugen. Das geht natuerlich nur, wenn keine weiteren Objekt-Dateien dazugebunden werden muessen. Nuetzlich ist das, wenn ein kleines Testprogramm (6) schnell kompiliert werden soll. Dann muss **keinerlei** Eintrag oder Aenderung in `makefile` durchgefuehrt werden, sondern es kann direkt `make <`*Programmname_ohne_Endung*`>` aufgerufen werden. Den Rest macht `make` dann alleine. `(`Wir arbeiten daran, dass make **immer** so funktioniert `;-)`

Makro-Definitionen

Das letzte wesentliche Element eines `makefile`s sind Makro-Definitionen die als Textplatzhalter fungieren:

```
SOURCE = login.c
HEADERS = login.h announce.h
DEPENDCFLAGS =  -I/usr/global/lib/g++-include \
      -I/project/mobilise/genlib/src
```

Die Dereferenzierung eines Makros erfolgt mittels `$(Makro)`, im obigen Beispiel also:

```
login.o:      $(SOURCE) $(HEADERS)
          gcc -ansi -c -o login.o login.c
```

Aufgrund der Definition der allgemeinen **Regel** (fuer `.c.o:`) wuerde es jetzt auch ausreichen, nur die **Abhaengigkeiten** anzugeben:

```
login.o:      $(SOURCE) $(HEADERS)
```

Mit diesem Eintrag wird sichergestellt, dass `login.o` auch dann neu erzeugt wird, falls nur eine header-Datei geaendert ist. Die **allgemeine Regel** kann das nicht erkennen.

Haeufig werden von Header-Dateien weitere Header-Dateien eingezogen. Dadurch kann es zu komplexen **Abhaengigkeiten** kommen. Das Programm `makedepend` dient dazu diese herauszufinden und das `makefile` entsprechend anzupassen. Wie das geht kommt jetzt!

Abhaengigkeiten finden mit makedepend

Das Programm `makedepend` dient dazu von einer oder mehreren C/C++-Quelldateien automatisch die **Abhaengigkeiten** zu erkennen und entsprechende Eintraege im `makefile` vorzunehmen. Haeufig wird der Aufruf von `makedepend` wieder selbst ueber `make` vorgenommen. (7) Dazu dient der Eintrag:

```
depend:
      makedepend -- $(DEPENDCFLAGS) -- $(SOURCE)
```

Weiter unten im `makefile` muss sich dann die Zeile:

```
# DO NOT DELETE THIS LINE -- make depend depends on it.
```

befinden. Der Bereich hinter dieser Zeile wird durch `makedepend` verwaltet und kann daher nicht fuer eigene Eintraege verwendet werden. Das Kommando `make depend` veranlasst `make` das Programm `makedepend` aufzurufen. `makedepend` durchsucht daraufhin die Datei `login.c` und fuegt fuer jede gefundene `#include ...` Direktive eine entsprechende Zeile in `makefile` hinzu und zwar hinter der oben angegebenen Kommentarzeile. Dabei ist zu beachten, dass `makefile` veraendert wird. (8) Die urspruengliche Datei wird in `makefile.bak` umbenannt. Im Makro `DEPENDCFLAGS` sind dabei die nach

Header-Dateien zu durchsuchenden Verzeichnisse angegeben. Im obigen Beispiel werden folgende Zeilen hinzugefuegt:

```
login.o: /project/mobilise/genlib/src/ansi_compat.h
login.o: /usr/global/lib/g++-include/stdio.h
login.o: /usr/global/lib/g++-include-2.3/_G_config.h login.h announce.h
login.o: /usr/global/lib/g++-include/stdlib.h
login.o: /usr/global/lib/g++-include/stddef.h
```

Im Ablauf von `makedepend` sind zur Erhoehung der Laufzeit-Effizienz einige subtile Annahmen gemacht, die dazu fuehren koennen, dass sich das Programm nicht so verhaelt wie vermutet. Eine wesentliche implizite Voraussetzung besteht in der Annahme, dass alle im Aufruf von `makedepend` angegebenen Quell-Dateien im wesentlichen die gleichen Header-Dateien einschliessen. Daher ist eine gewisse Vorsicht bei der Verwendung von `makedepend` angebracht.

Mittlerweile wird anstelle des Programms `makedepend` die entsprechende Funktionalitaet des GNU-Compilers verwendet: Der wesentliche Unterschied zu `makedepend` besteht darin, dass die entsprechenden Eintraege nicht mehr direkt in `makefile` hinzugefuegt werden, sondern in die Datei `.depend` geschrieben werden. Diese Datei wird dann im `makefile` durch die Anweisung:

```
include .depend
```

eingeladen. Daher kommt es zu einer Fehlermeldung, falls eine solche Datei nicht existiert. Dies kann durch ein vorheriges `touch .depend` vermieden werden.

Anpassung des makefiles

Zur Anpassung des makefiles fuer ein eigenes Projekt muessen im wesentlichen die folgenden Aenderungen vorgenommen werden:

1. Definition (oder Aenderung) eines Makros zur Bezeichnung der Quelldateien:

   ```
   MY_SOURCE = file1.c file2.c file3.c
   ```

2. Entsprechend fuer die Objektdateien:

   ```
   MY_OBJECTS = file1.o file2.o file3.o
   ```

3. Falls erforderlich muessen die Macros INCLUDE_DIRS, LINK_DIRS und LINK_LIBS mit den benoetigten Verzeichnissen und Bibliotheken definiert werden. Das Macro DEPEND_SRCS sollte um die neu definierten Quelldateien ergaenzt werden, z.B.:

   ```
   INCLUDE_DIRS = -I/home/maf/special_includes
   LINK_DIRS = -L/home/maf/special_libs
   LINK_LIBS = -lmaf        Damit wird die Bibliothek libmaf.a aus dem
                            Verzeichnis /home/maf/special_libs mit
                            gebunden. Die dazu gehoerenden Header Dateien
                            befinden sich wahrscheinlich im Verzeichnis
                            /home/maf/special_includes
   DEPEND_SRCS = $(SOURCE) $(MY_SOURCE)
   ```

4. Hinzufuegen der Definition des gewuenschten Ziels (meist der Name des ausfuehrbaren Programms), z.B.:

   ```
   exefile:      $(MY_OBJECTS)
                 $(GXX) $(LFLAGS) $(LDIRS) $(LINK_DIRS) -o $@ \
                 $(MY_OBJECTS) $(LLIBS) $(LINK_LIBS)
   ```

   Dazu muessen im Eintrag fuer das Ziel login nur drei Aenderungen

```
      vorgenommen werden: Eintrag des neuen Ziels und Anpassung des
      $(OBJECTS) Macro in Abhaengigkeiten und Regel.

   5. Nach Aufruf von make depend kann mit make exefile das neue Ziel erzeugt
      werden. make depend muss immer dann neu aufgerufen werden falls sich
      Aenderungen an der #include Struktur der Quell- und Header-Dateien
geaendert
      hat.
```

Falls die Objekt-Dateien **nicht** mittels der **voreingestellten Regeln** erzeugt werden koennen, muessen noch entsprechende Eintraege fuer deren Erzeugung ergaenzt werden (z.B. fuer automatisch generierte Programme des SDT).

Fußnoten

1.1.2 (1)

Beim Kopieren dieser Datei bitte auch `.depend` aus dem gleichen Verzeichnis kopieren (s. Abschnitt sec_makedepend).

1.1.3 (2)

ComNets object-oriented Class Library, siehe eigene Dokumentation.

1.1.4 (3)

Fuer die genaue Syntax s. `man` pages bzw. Unix-Buecher.

1.1.5 (4)

<TAB> nicht vergessen!

1.1.6 (5)

Genauere Erklaerungen zu diesen **dynamischen Makros** finden sich wieder unter `man make`

1.1.7 (6)

Testprogramme duerfen **nie** den Namen `test` bekommen. Dies fuehrt zu Konflikten mit dem in der **Shell** eingebauten Kommando `test`.

1.1.8 (7)

Merke: mit make koennen auch andere Programme als nur der Compiler aufgerufen werden!

1.1.9 (8)

Dies ist besonders zu beruecksichtigen, falls sich `makefile` gerade z.B. im `emacs` befindet und von dort aus `make depend` aufgerufen wird!

1.1.10 (9)

Im Fenster mit der hoechsten Nummer wird dann natuerlich in das erste Fenster gewechselt!

1.1.11 (10)

Oder selbstverstaendlich auch im gleichen!

1.1.12 (11)

Eine `vi` aehnliche Suchfunktion ist ebenfalls implementiert, siehe Dokumentation.

1.1.13 (12)

oder nur einigen davon -- je nachdem wie viele Zeilen gerade auf den Monitor passen

1.1.14 (13)

erscheinen dann invers