

8.2 Programmierung/make

8.2.1 Zweck

Das Programm `make` dient zur automatischen Erstellung und Aktualisierung von Software-Projekten, die aus mehreren Quelltextdateien bestehen.

Dabei wird in einer Datei mit dem Namen `Makefile` eine Beschreibung der Abhängigkeiten und der Erstellung des Projektes hinterlegt. Beim Aufruf von `make` werden die Abhängigkeiten durchsucht und nur die Projektteile neu erstellt, die seit dem letzten Aufruf verändert wurden.

8.2.2 Arbeitsweise

- Wird `make` aufgerufen, wird *im aktuellen Verzeichnis* nach einer Beschreibungsdatei namens `Makefile` oder `makefile` gesucht. Deshalb ist es sinnvoll, vor dem Aufruf von `make` in das entsprechende Verzeichnis zu wechseln. Mit der Option `-f dateiname` kann auch ein anderer Name für die Beschreibungsdatei angegeben werden.
- Ruft man `make` ohne Parameter auf, wird der erste Eintrag in der Beschreibungsdatei ausgeführt.
- Ruft man `make` mit einem Parameter auf (z.B. `make hauptprog`), so wird der Parameter als so genanntes *Ziel* aufgefasst. Das Ziel wird dann in der Beschreibungsdatei gesucht und mit Hilfe der angegebenen Regeln aufgebaut.
- Gibt man mehrere Parameter an (z.B. `make all install clean`), dann werden die Ziele in der Beschreibungsdatei gesucht und der Reihe nach aufgebaut.
- Jedes Ziel wird durch einen¹ Eintrag beschrieben; dieser besteht aus
 - einer Abhängigkeitszeile, beginnend mit dem Namen eines Ziels (eines Software-Produkts),
 - direkt anschließend optionalen Kommandozeilen².

Mit einer Leerzeile endet das Ziel. Hier ein Beispiel:

```
1 meinziel: datei1.o datei2.o
2          gcc -o meinziel datei1.o datei2.o
```

- Eine Abhängigkeitszeile hat stets die Form

```
1 meinziel: teil1 teil2 ...
```

Sie sagt aus: Um `meinziel` zu erstellen, sind `teil1` und `teil2` usw. notwendig. `teil1` und `teil2` können Dateien sein (wie im Beispiel) oder auch Ziele in der Beschreibungsdatei.

- Die Kommandozeile beginnt mit einem Tabulator-Zeichen. Dieses Zeichen ist unbedingt notwendig und kann nicht durch Leerzeichen ersetzt werden³. Die Kommandozeile beschreibt das Kommando, das ausgeführt werden muss, um `meinziel` zu erstellen.
- Das Kommando zu einem Ziel namens `meinziel` wird aufgerufen, falls
 - a) die Ziel-Datei `meinziel` nicht existiert,
 - b) die Ziel-Datei `meinziel` älter ist als die Dateien, von denen es abhängt (im Beispiel oben `datei1.o` und `datei2.o`) — dann muss es nämlich erneuert werden,

¹genau: mindestens einen

²Falls ein Ziel durch mehrere Einträge beschrieben wird, darf nur ein Eintrag Kommandozeilen aufweisen.

³Man kann dies testen mit `cat -e -vt Makefile`.

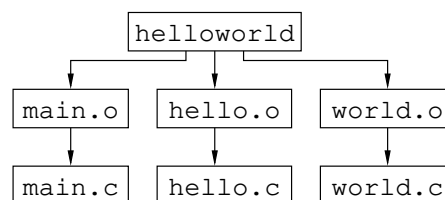
- c) oder wenn keine Dateien angegeben sind, von denen es abhängt. Dann wird es in jedem Fall erneuert.
- Aber: Falls eine der abhängigen Dateien an einer anderen Stelle der Beschreibungsdatei als Ziel genannt ist, wird *vorher* dieses Ziel ebenfalls überprüft (Existiert es? Ist es neuer als die Dateien, von denen es abhängt?) und, falls nötig, aufgebaut.
- So wird ein ganzer Abhängigkeitsbaum durchlaufen. Es werden aber immer nur die Teile erneuert, deren Quellen sich verändert haben.
- Das erspart in der Software-Entwicklung sehr viel Zeit und Nerven, denn so kann man eben schnell eine Änderung ausprobieren, ohne gleich alle Programmteile neu aufbauen zu müssen.

8.2.3 Weitere Regeln

- Alle Zeilen dürfen mit Backslash als letztem Zeichen umgebrochen werden. Gerade bei langen Kommandozeilen ist das wichtig.
- Wie bei der Shell dürfen Variablen benutzt werden; zur Unterscheidung heißen sie hier Makros (Definition wie bei Shell-Variablen; ihr Inhalt wird mit `${NAME}` gelesen). Makros werden sinnvollerweise nur *einmal* zu Beginn des Makefile gesetzt.
- Umgebungsvariablen können ebenfalls ausgewertet werden; soll eine Umgebungsvariable anstelle eines gleichnamigen Makros benutzt werden, erfolgt ihr Aufruf mit der Form `$$ {NAME}`.
- `make` kennt übrigens schon eine Reihe von vordefinierten Makros, die z.B. aus C-Quelltexten Objektdateien erstellen.

8.2.4 Beispiel

Im folgenden Beispiel wird ein Programm `helloworld` vorgestellt, das aus den drei Quelltextdateien `main.c`, `hello.c` und `world.c` erstellt werden soll. Dabei soll jede Quelltextdatei einzeln kompiliert werden. Das ergibt folgende Abhängigkeiten:



Im Makefile sieht das dann so aus:

```

1 # Makefile fuer Beispiel 1
2
3 # Objekte main, hello und world zu einer
4 # ausfuehrbaren Datei zusammenbauen
5 helloworld: main.o hello.o world.o
6     gcc -o helloworld main.o hello.o world.o
7
8 # Objekt main.o erstellen
9 main.o: main.c
10     gcc -c main.c
11
12 # Objekt hello.o erstellen
13 hello.o: hello.c
14     gcc -c hello.c

```

```
15 |
16 | # Objekt world.o erstellen
17 | world.o: world.c
18 |     gcc -c world.c
```

main.c:

```
1 | #include <stdio.h>
2 | void hello(void);
3 | void world(void);
4 | /*****/
5 |
6 | int main(void)
7 | {
8 |     hello();
9 |     world();
10 |    return 0;
11 | }
```

hello.c:

```
1 | #include <stdio.h>
2 |
3 | void hello(void)
4 | {
5 |     printf("Hello ,_\n");
6 | }
```

world.c:

```
1 | #include <stdio.h>
2 |
3 | void world(void)
4 | {
5 |     printf("World!\n");
6 | }
```

Wenn noch kein Modul vorhanden ist, wird zuerst jede Quelltextdatei einzeln zu einer Objektdatei kompiliert (`gcc -c`). Anschließend werden alle drei Module zusammengelinkt zu einer ausführbaren Datei (`gcc`):

```
$ make
gcc -c main.c
gcc -c hello.c
gcc -c world.c
gcc -o helloworld main.o hello.o world.o
$
```

Wird jetzt z.B. `world.c` editiert und verändert, braucht nur noch dieser Teil des Baums erneuert zu werden:

```
$ vi world.c
$ make
gcc -c world.c
gcc -o helloworld main.o hello.o world.o
$
```

Den Rest des Baums lässt `make` in Ruhe.