8 Make-Programm

Nach Änderungen in C-Source-Dateien ist es nötig, diese mit dem Compiler zu übersetzen und danach die Module des Projekts zusammen zu linken. Besteht ein Projekt aus mehreren Modulen, möchte man sichergehen, dass sich alle zu den C-Dateien gehörenden Objekte auf dem neuesten Stand befinden. Dies kann natürlich dadurch sichergestellt werden, dass bei der Änderung eines Moduls alle Module neu compiliert werden. Dies ist eine machbare, aber keine effektive Lösung, die bei großen Projekten viel Zeit in Anspruch nimmt. Das Compilieren kann dann schon einige Stunden dauern. Eine bessere Lösung wäre es, nur die Dateien zu übersetzen, die sich auch wirklich geändert haben und deren C-Code nicht mehr mit dem vorhandenen Compilat übereinstimmt. Ein fehlendes Compilat muss natürlich wie ein veraltetes bewertet werden. Das Programm *make* bewerkstelligt dies. Anhand des Datums der Dateien stellt es fest, ob eine C-Datei neuer als die dazugehörige Objektdatei ist und ruft dann den Compiler auf, der die C-Datei übersetzt. Die alte Objektdatei wird durch eine neue ersetzt.

Damit *make* dies tun kann, ist es nötig, ihm die Abhängigkeiten des Projekts mitzuteilen. Dies geschieht in einer *Make-Datei* (engl.: *makefile*). Da sich bei den Make-Programmen eine Art „Quasi-Standard“ etabliert hat, möchte ich hier kurz auf den Aufbau einer Make-Datei eingehen und auch einige Probleme aufzeigen. Unter Quasi-Standard ist **nicht** eine Standardisierung wie bei ANSI-C zu verstehen, sondern nur ein gemeinsames Funktionsprinzip. Make-Programme gibt es von vielen verschiedenen Herstellern und mit verschiedenen Funktionen. Die Hersteller der Make-Programme haben diese optimal an ihre jeweiligen Compiler und Linker angepasst. Da der Make des GNU-Projekts kostenlos zur Verfügung steht, habe ich diesen für meine nachfolgenden Beispiele ausgewählt. Der Nachteil des GNU-Makes ist, dass sich die verfügbaren Beispiele meist auf Linux-Projekte beziehen. Arbeitet man nicht unter Linux, müssen die Make-Dateien an das verwendete System angepasst werden. Ein typisches Problem ist die Verwendung von Linux-Betriebssystemkommandos wie *rm* (löschen) und *cp* (kopieren), die in einer Windows-Umgebung in *del* (für einzelne Dateien) oder *deltree* (für ganze Verzeichnisse; Vorsicht!) und *copy* geändert werden müssen.

In der Sprache C besteht der programmerzeugende Source-Code nicht nur aus der C-Datei, sondern auch aus den die Schnittstellen beschreibenden H-Dateien, die vom Präprozessor in den C-Code eingefügt werden. Eine C-Datei enthält meist mehrere H-Dateien. Eine Objektdatei ist also nicht nur von der C-Datei abhängig, sondern auch von den durch die *include*-Anweisung eingefügten H-Dateien.

Die Make-Datei ist eine Auflistung der Abhängigkeiten und der entsprechenden Aktionen, die geschehen sollen, um die Objektdatei und schließlich auch die ausführbare Datei auf den neuesten Stand zu bringen. Der Standardname einer Make-Datei ist *makefile*. Der Schalter -f*Dateiname* macht es möglich, einen alternativen Dateinamen zu spezifizieren, um z. B. mehrere Projekte in einem Verzeichnis zu verwalten.

Die einfachste Form des Eintrags in einer Make-Datei ist eine so genannte *explizite Regel* in der Form:

```
Ziel:   Abhängigkeit1 Abhängigkeit2
        Kommando1
        Kommando2
```

Das Ziel, also z. B. eine Objektdatei, hängt von den Dateien ab, die in *Abhängigkeit1* bis *N* aufgelistet sind. Ist eine Datei in der Abhängigkeitsliste neuer als die Zieldatei, so werden die Kommandos aus der Kommandoliste, hier *Kommando1* und *Kommando2,* gestartet. Soll eine Aktion unbedingt, das heißt ohne die Auswertung von Abhängigkeiten, ausgeführt werden, entfällt der Eintrag der Abhängigkeiten, und die Zeile bleibt leer.

Achtung: Kommandos im makefile (für die meisten Make-Programme) müssen mit dem Tabulator beginnen! Beginnt ein Kommando nicht mit dem Tabulatorzeichen, kommt es zu Fehlern. Leider sind die Fehlermeldungen bei *make* nicht sehr komfortabel, und ein fehlender Tabulator oder ein in Leerzeichen umgewandelter Tabulator kann schon einmal zu längeren Suchaktionen führen.

Ein typisches Beispiel für eine solche Fehlermeldung ist die Ausgabe:

```
makefile:11: *** missing separator. Stop.
```

Zugegeben nicht sehr aussagekräftig! Um diesen Fehler zu erzeugen, wurde nur der Tabulator durch die entsprechende Anzahl von Leerzeichen ersetzt. Optisch hätte sich das Aussehen der Datei in den meisten Editoren nicht geändert. Manche Editoren wandeln bei der entsprechenden Einstellung automatisch alle Tabulatoren in Leerzeichen um. Beim Bearbeiten einer Make-Datei wäre das ziemlich fatal. Der GNU-Make ist bei fehlenden Tabulatoren wesentlicher hilfreicher und weist explizit auf diesen Fehler hin.

In der Praxis sieht ein Eintrag in der Make-Datei wie nachfolgend aus:

```
main.o: calc.h mmi.h main.c
        cc main.c
```

Die Objektdatei main.o ist von der Datei calc.h mmi.h und main.c abhängig. Ist eine von diesen Dateien neuer als die Objektdatei main.o, so wird der Compiliervorgang ausgelöst, der hier durch „cc main.c“ erfolgt. Da die Objektdatei noch durch den Linker bearbeitet werden muss, muss eine weitere Zeile mit der Abhängigkeit der ausführbaren Datei von den einzelnen Objektdateien hinzugefügt werden. Da das Make-Programm beim Aufruf ohne besondere Parameter stets versucht, das erste Ziel (den ersten Targeteintrag) zu erstellen, wird das gewünschte Endergebnis als er-

stes Ziel eingetragen. Deshalb steht das Ziel projekt.x hier auch als erstes Ziel in der Datei. Die Ziele werden auch als *Targets* bezeichnet. Analog zur obigen Zeile wird für die Steuerung des Linkprozesses eine Zeile wie folgt aufgebaut:

```
projekt.x: main.o mmi.o
        lnk main.o mmi.o -o projekt.x
```

Das Gesamtergebnis projekt.x ist als Target eingesetzt. Die Dateien main.o und mmi.o bilden die Abhängigkeiten, und durch die Anweisungszeile lnk main.o mmi.o -o projekt.x wird bei Bedarf der Linker mit den erforderlichen Dateien und einem Steuerparameter zur Benamung des Ergebnis aufgerufenen. Genauso wie bei den Abhängigkeiten können auch bei den Zielen (Targets) mehrere Ziele durch Leerzeichen getrennt angegeben werden. Die komplette Make-Datei ist in **Beispiel 8.1** dargestellt. Das Projekt besteht aus den C-Dateien main.c und mmi.c, das Ergebnis des Link und Compiliervorgangs soll projekt.x heißen. Wie im Beispiel sichtbar, können Kommentare mit # gekennzeichnet werden.

```
# Einfache Beispielmakedatei fuer ein Projekt aus den Dateien
#  Sourcedateien: main.c mmi.c
#  Includedateien: calc.h mmi.h

projekt.x: main.o mmi.o
        lnk main.o mmi.o -o projekt.x
main.o: calc.h mmi.h main.c
        cc main.c
mmi.o:  mmi.h mmi.c
        cc mmi.c
```

Beispiel 8.1: Einfache Make-Datei

Schon in dem einfachen Beispiel fällt eine Wiederholung von Dateinamen auf. Das Ziel projext.x, die Sourcecodedateien main.c und mmi.c findet man jeweils zweimal in der Datei. Um solche Mehrfachnennungen zu vermeiden, die Übersichtlichkeit der Datei zu wahren und Tipparbeit zu vermeiden, werden *Variablen* verwendet. Variablen werden definiert durch den Variablennamen, gefolgt von einem Gleichheitszeichen mit der nachfolgenden Zuweisung des Variableninhalts:

```
Modul1 = main
```

mit $(Modul1) wird der Inhalt der Variablen an beliebigen Stellen wieder eingesetzt.

Die Make-Datei aus **Beispiel 8.1** ändert sich mit der Verwendung von Variablen wie im **Beispiel 8.2**. Die Erweiterung einer solchen Datei ist einfacher, da nicht die Modulnamen individuell hinzugefügt werden müssen, sondern die Zeilen kopiert werden können und die Ziffern zur Unterscheidung der Makros einfach hochgezählt werden. Ebenso wurden die Tools C-Compiler (cc) und Linker (lnk) durch Makros ersetzt, damit die entsprechenden Tools einfacher ausgetauscht werden können.

```
# Beispielmakedatei mit der Verwendung von Makros
#  Sourcedateien: main.c mmi.c
#  Includedateien: calc.h mmi.h
Modul1 = main
Modul2 =  mmi
Target = projekt.x
Linker =  lnk
Compiler = cc

$(Target): $(Modul1).o $(Modul2).o
                    $(Linker) $(Modul1).o $(Modul2) -o $(Target)
$(Modul1).o: calc.h mmi.h $(Modul1).c
                    $(Compiler) $(Modul1).c
$(Modul2).o:  mmi.h $(Modul2).c
                    $(Compiler) $(Modul2).c
```

Beispiel 8.2: Make-Datei mit Makros

Die auszuführenden Kommandos sind bei jedem zu compilierenden Modul gleich. Durch die Verwendung von impliziten Regeln ist es möglich, diese Wiederholungen einzusparen. Eine implizite Regel ist schon die in make vordefinierte Regel, eine C-Datei in eine O-Datei zu überführen.

Die implizite Regel, die für die C-Programmierung interessant ist, ist die Regel:

$(CC) -c $(CPPFLAGS) $(CFLAGS)

die alle C-Dateien in O-Dateien überführt unter der Verwendung der Variablen CC, CPPFLAGS und CFLAGS. Diese Variablen können vom Anwender auf die entsprechenden Werte gesetzt werden, um den Namen des C-Compilers und notwendige Kommandoschalter für den C-Compiler zu übergeben. Die implizite Regel vereinfacht die Beispiel-Make-Datei weiter, indem die auszuführenden Kommandos zur Compilation der C-Dateien wegfallen und eine reine Abhängigkeitsliste (**Beispiel 8.3**) stehen bleibt.

```
# Beispielmakedatei mit der Verwendung von Makros und impliziten Regeln
#  Sourcedateien: main.c mmi.c
#  Includedateien: calc.h mmi.h
Modul1 = main
Modul2 =  mmi
Target = prj.x
Linker =  lnk
CC = cc
$(Target): $(Modul1).o $(Modul2).o
          $(Linker) $(Modul1).o $(Modul2).o -o $(Target)
$(Modul1).o: calc.h mmi.h
$(Modul2).o: mmi.h
```

Beispiel 8.3: Make-Datei unter Verwendung von impliziten Regeln