ANWENDER-DOKUMENTATION	Anleitung für den Programmierer	MOS
11/87	Wartungsprogramm make	MUTOS 1700

Programmtechnische Beschreibung Teil 2

Anleitung für den Programmierer

Wartungsprogramm make

AC A 7100/7150

VEB Robotron-Projekt Dresden

C 8014-0003-1 M 3030

Ausgabe: 11/87

Die Ausarbeitung dieser Dokumentation erfolgte durch ein Kollektiv des
VEB Robotron-Elektronik Dresden Stammbetrieb des VEB Kombinat Robotron.

Im Interesse einer ständigen Weiterentwicklung werden alle Leser gebeten, Hinweise zur Verbesserung der Dokumentation dem Herausgeber mitzuteilen.

Herausgeber:

VEB Robotron-Projekt Dresden
Leningrader Str. 9
Dresden 8010

Kurzreferat

"make(1)" Größere Programmprojekte bestehen gewöhnlich aus mehreren Teilprogrammen, die ihrerseits in verschiedenen Files stehen können. Werden änderungen an solchen Teilprogrammen durchgeführt, kann man leicht die übersicht darüber verlieren, welche Files infolge der änderungen erneut übersetzt und eventuell noch bearbeitet werden müssen. Das Programm make mit seinem gleichnamigen Kommando liefert einen unkomplizierten Mechanismus für die Wartung von Programmprojekten. Mit Hilfe von make läßt sich nach einer oder mehreren änderungen an den Teilprogrammen die neueste Version des Programmprojektes insgesamt oder von Teilen, welche hier allgemein Ziele heißen, herstellen. Damit make die für ein bestimmtes Programmprojekt benötigten Arbeitsschritte kennt, müssen diese vom Bearbeiter ein einziges Mal spezifiziert werden. Diese Spezifikationen werden in einem, dem bestimmten Programmprojekt zugeordneten File, dem Beschreibungs-File, untergebracht. Zu diesen Spezifikationen gehören Kommandos und Beziehungen zwischen den das Programmprojekt bildenden Files. Im allgemeinen heißt das Beschreibungs-File Makefile oder makefile.
Die grundlegende Funktion von make besteht aus folgenden Schritten: Zunächst ist der Name des gewünschten Zieles zu finden, dann ist zu sichern, daß alle Files, von denen das Ziel abhängt, existieren und auf dem neusten Stand sind, und schließlich ist das Ziel zu generieren. Dies aber nur dann, wenn es seit seiner letzten Generierung modifiziert wurde oder noch nicht existiert. Diese grundlegende Funktion wird auch auf die benötigten Zwischenziele angewendet. Wie aus dieser Charakterisierung hervorgeht, werden von make nur die tatsächlich benötigten Arbeitsschritte aktiviert.

Inhaltsverzeichnis Seite

1. Einleitung

Bei der Entwicklung von größeren Programmen ist es üblich, diese aus kleineren Programmteilen aufzubauen. Diese Programmteile können sehr verschiedene Behandlungen erfordern. So kann man sich die jeweilige Verarbeitung der einzelnen Programmteile durch verschiedene Programmgeneratoren oder Compiler vorstellen, wobei vielleicht gemeinsame Definitionen und Deklarationen aus bestimmten Files verwendet werden sollen und für die Programmteile spezifische Kommandos mit entsprechenden Optionen angewendet werden. Die sich ergebenden Files sind dann eventuell mit verschiedenen Bibliotheken mit speziellen Optionen zu laden oder sind weiter zu übersetzen. Bei änderungen in den Programmteilen kann man leicht die übersicht darüber verlieren, welche Programmteile z.B. erneut zu übersetzen sind, welche Teile weiter verwendet werden können usw. Wird z.B. vergessen, einen Programmteil neu zu compilieren, der in seinem Quellcode geändert wurde, so arbeitet man mit dem vorhergehenden alten Objektprogramm weiter und hat u. U. keine Dokumentation des alten Quellprogramms. Derartige Fehler zu finden, ist sehr zeitaufwendig. Um solchen technologischen Fehlern zu entgehen, könnten nach jeder änderung alle zum Programmprojekt gehörenden Teile erneut übersetzt und geladen werden. Diese Verfahrensweise ist unökonomisch.
Das Wartungsprogramm make automatisiert viele der für die Programmentwicklung und -wartung erforderlichen Aktivitäten. Die für ein bestimmtes Programmprojekt benötigten Kommandos mit ihren Parametern und die zwischen den Quell-Files und den sich ergebenden Files bestehenden Beziehungen werden in einem speziellen File, dem Beschreibungs-File, notiert. Das Beschreibungs-File wird allgemein Makefile oder makefile genannt. Man kann dem Beschreibungs-File auch jeden anderen Namen geben, muß diesen aber dann beim Kommando mit angeben. Mit dem einfachen Kommando

```
make
```

wird das Programm make gestartet. Dies sorgt dafür, daß das Programmprojekt, falls es seit seiner Erstellung modifiziert wurde, auf den neuesten Stand gebracht wird. Der Nutzer muß sich nicht mehr um die einzelnen benötigten Kommandos sorgen. Es ist in vielen Fällen, bei denen ein Kommando mit vielen Parametern einzugeben ist, einfacher, den Aufwand für das besondere Beschreibungs-File einmal zu treiben. Danach kann dann das kurze Kommando make öfters verwendet werden. Unter Verwendung von make gestaltet sich der Programmentwicklungszyklus wie folgt:

```
Denken - Editieren - make - Testen . . .
```

Das Werkzeug make rationalisiert sowohl die Entwicklung als auch die Wartung von Programmprojekten.

2. Grundsätzliche Arbeitsweise

Die grundlegende Funktion von make ist, die neueste Version eines Files, das Ziel-File, herzustellen. Für diesen Zweck muß gewährleistet werden, daß alle Files, von denen das Ziel-File abhängt, d.h. diejenigen, die für die Herstellung des Ziel-Files als "Quell"-Files vorausgesetzt werden, vorhanden und auf dem neuesten Stand sind. Um diese Frage entscheiden zu können, wertet make das Datum und die Uhrzeit der Erstellung bzw. der letzten Veränderung der einzelnen Files aus.
Zur Veranschaulichung soll folgendes Beispiel dienen. Ein Programm prog sei herzustellen, indem drei Files, die in der Programmiersprache C geschrieben sind, x.c, y.c und z.c, zu compilieren und anschließend mit der Bibliothek lS zu laden sind. Nach den in MUTOS 1700 üblichen Regeln gibt der C-Compiler für die oben genannten Files die Files x.o, y.o und z.o aus. Angenommen, die Files x.c und y.c haben einige Deklarationen gemeinsam, die im File defs stehen, das File z.c habe sie aber nicht. Dies bedeutet, daß sowohl im File x.c als auch im File y.c die Zeile

```
#include "defs"
```

steht. Der folgende Text beschreibt die Beziehungen und Operationen, mit denen das Programm make das Programm prog herstellen kann oder dessen neueste Version, falls in den vier Quell-Files x.c, y.c, z.c und defs geändert wurde.

```
prog :  x.o  y.o  z.o
          cc  x.o  y.o  z.o   -lS  -o  prog

x.o  y.o :   defs
```

Wenn diese Informationen in dem Beschreibungs-File mit dem Namen makefile gespeichert werden, löst das Kommando

```
make
```

die gewünschten Operationen aus, um prog zu schaffen.
Das Programm make benutzt folgende Informationsquellen: ein vom Nutzer geliefertes Beschreibungs-File (oben makefile genannt), die Namen der Files sowie Datum und Uhrzeit ihrer Erstellung bzw. letzten Veränderung. Diese Informationen werden durch das File-System von MUTOS 1700 geführt. Außerdem verwendet make noch einige interne Regeln, um fehlende Angaben im Beschreibungs-File sinnvoll ergänzen zu können. In unserem Beispiel oben bedeutet die erste Zeile, daß prog von drei .o-Files abhängt. Wenn von diesen drei Files die neuesten Versionen vorhanden sind, beschreibt die zweite Zeile, wie sie zusammen mit der Bibliothek zu laden sind, damit prog hergestellt wird. Die dritte Zeile besagt, daß die Files x.o und y.o vom File defs abhängen. Aus dem File-System von MUTOS 1700 und den Namen x, y und z erkennt make, daß es drei gleichartig benannte .c-Files gibt, die mit den .o-Files im Zusammenhang stehen. Mit dieser "Erkenntnis" und den internen Regeln ist make bekannt, wie aus den Quell-Files die Objekt-Files zu erzeugen sind (indem nämlich ein Kommando "cc -c" zu aktivieren ist).

Das folgende ausführliche Beschreibungs-File ist zu dem vorigen äquivalent. Es nutzt nicht die Intelligenz von make aus.

```
prog :  x.o  y.o  z.o
          cc  x.o  y.o  z.o  -lS  -o  prog

x.o :  x.c  defs
          cc  -c  x.c
y.o :  y.c  defs
          cc  -c  y.c
z.o :  z.c
          cc  -c  z.c
```

Wenn keins der im Beispiel angegebenen Quell- oder Objekt-Files verändert wurde, seitdem prog das letzte Mal hergestellt wurde, dann sind alle Files aktuell, d.h. sie sind auf dem neuesten Stand, und das Kommando

```
make
```

würde diese Tatsache feststellen und anhalten. Wenn jedoch z.B. das File defs editiert wurde, werden mit Hilfe von make die Files x.c und y.c erneut compiliert (aber z.c nicht) und dann würde prog erneut aus den aktuellen neuen .o-Files hergestellt werden. Wenn in einem anderen Fall nur das File y.c verändert wurde, dann muß nur dieses erneut compiliert und prog anschließend geladen werden.
Es besteht die Möglichkeit, hinter dem Kommando make auf derselben Zeile den Namen des Zieles, das hergestellt werden soll, anzugeben. Wird kein Ziel angegeben, so wird das im Beschreibungs-File zuerst genannte Ziel hergestellt, in unserem Beispiel also prog. Wenn man, wieder auf das vorige Beispiel bezogen, das Kommando

```
make x.o
```

angibt, so wird x.o erneut compiliert, falls x.c oder defs modifiziert wurden.
Es wird nochmals betont, daß von make nur dann Ziele hergestellt werden, wenn sie noch nicht vorhanden sind oder im anderen Fall, wenn Veränderungen an den Files erfolgten, von denen das Ziel abhängt. Zum Vergleich werden bei vorhandenen Files der Zeitpunkt der Erstellung oder der letzten änderung, und bei noch nicht vorhandenen Files die gerade aktuelle Zeit benutzt. Es ist häufig nützlich, Regeln mit mnemotechnischen Namen, die Kommandos enthalten, zu bilden, die aber kein File mit dem angegebenen Namen herstellen. Vielmehr wird damit die Fähigkeit von make in vorteilhafter Weise genutzt, Files zu generieren und Substitutionen von Makros auszuführen. So kann z.B. eine Eintragung "save" vorgesehen werden, um eine bestimmte Menge von Files zu kopieren, oder eine Eintragung "clearup" kann verwendet werden, um nicht weiter benötigte Files zu löschen. In einigen anderen Fällen kann ein File mit der Länge Null gewartet werden, um lediglich festzustellen, ob nach einem bestimmten Zeitpunkt eine Aktion ausgeführt wurde. Dies ist besonders bei der Wartung von Archiven und Listen hilfreich, die sich nicht am Ort befinden.
make besitzt einen einfachen Ersetzungstext-Mechanismus, die sogenannten Makros, für das Substituieren von Zeilen. Die Makros können zur Beschreibung von Abhängigkeiten zwischen den Files oder für die Spezifizierung von Kommandos verwendet werden. Die Makros werden definiert, indem bei Kommandoargumenten oder in den Zeilen des Beschreibungs-Files ein Gleichheitszeichen angegeben wird. Makros werden aufgerufen, indem dem Namen des Makros das Zeichen Dollar "$" vorangestellt wird. Dabei müssen Namen, die länger als ein Zeichen sind, eingeklammert werden. Beispiele für richtige Makroaufrufe sind:

```
$Si $(CFLAGS) $2 $(xy) $Z $(Z)
```

Die beiden letzten Aufrufe sind identisch. Mit $$ wird ein Zeichen $ angegeben. Allen diesen Makros können "Werte", wie nachfolgend noch gezeigt wird, während der Kommandoeingabe zugewiesen werden. Die folgenden vier speziellen Makros verändern Werte während der Kommandoausführung: $*, $@, $? und $<. Sie werden später erklärt. Einige Fragmente, die als Anwendungsbeispiele dienen sollen und sich auf unser vorangegangenes Beispiel beziehen, sind:

```
OBJECTS = x.o y.o z.o
LIBES = -lS
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  -o prog
  . . .
```

Mit dem Kommando

```
make
```

werden wie zuvor die drei Objekt-Files zusammen mit der Bibliothek lS geladen. Mit dem Kommando

```
make  "LIBES= -ll -lS"
```

würden sie mit den zwei Bibliotheken, der Lex-Bibliothek ("-ll") und der Standardbibliothek ("-lS"), geladen, da die Makrodefinitionen in der Kommandozeile die Makrodefinitionen des Beschreibungs-Files unwirksam machen. Gleichzeitig wird hier gezeigt, daß es notwendig ist, die Argumente für das Kommando make, die durch Leerzeichen getrennt werden, insgesamt in Anführungsstriche zu setzen. Die folgenden Abschnitte behandeln die Form des Beschreibungs-Files, die Kommandozeile und diskutieren Optionen und interne Regeln ausführlich.

3. Beschreibungs-File und Substitutionen

Ein Beschreibungs-File enthält drei Arten von Informationen: Makrodefinitionen, Abhängigkeitsinformationen und auszuführende Kommandos. Es gibt auch eine Kommentarfestlegung. Alle Zeichen nach einem Nummernzeichen "#" werden einschließlich des "#" selbst bis zum Zeilenende ignoriert. Leerzeilen werden ebenfalls übergangen. Wenn eine Zeile, die nicht Kommentar ist, zu lang ist, kann sie unter Verwendung der Zeichen inverser Schrägstrich (Reverse, solidus) "\" und Neue Zeile (Newline) "NL" auf der nächsten Zeile fortgesetzt werden. In solch einem Fall werden die Zeichen "\" und "NL" sowie eventuell folgende Leer- und Tabulatorzeichen durch ein einziges Leerzeichen ersetzt.
Eine Makrodefinition ist eine Zeile, die ein Gleichheitszeichen enthält, dem kein Doppelpunkt oder Tabulatorzeichen voransteht. Dem Namen (eine Kette von Buchstaben und/oder Ziffern), der links vom Gleichheitszeichen steht (dazwischen stehende Leerzeichen und Tabulatorzeichen sind bedeutungslos) wird die Zeichenkette, die dem Gleichheitszeichen folgt (führende Gleichheitszeichen und Tabulatorzeichen sind ohne Bedeutung), zugewiesen. Richtige Makrodefinitionen sind:

```
2 = xyz
abc = -ll -ly -lS
LIBES =
```

Bei der letzten Definition wird dem Makro LIBES die Null-Kette zugewiesen. Ein Makro, welcher nirgends explizit definiert wurde, hat als Wert die Null-Kette. Makrodefinitionen können auch in der make-Kommandozeile stehen (siehe unten).
In den anderen Zeilen des Beschreibungs-Files stehen Informationen über die Ziel-Files. Die allgemeine Form der Kommandozeile ist:

```
Ziel1 [Ziel2...] :[:] [Abhängigkeit1...] [; Kommandos] [#...]    [(Tabulatorzeichen) Kommandos] [#...]
 . . .
```

Die Angaben in Klammern können weggelassen werden. Die Ziele und Abhängigkeiten werden durch Ketten von Buchstaben, Ziffern, Punkten und Schrägstrichen dargestellt. (Die Sonderzeichen von Shell wie "*" und "?" werden expandiert.) Ein Kommando ist irgendeine Zeichenkette, die nicht das Nummernzeichen, außer in Anführungsstrichen, oder Neue Zeile "NL" enthält. Kommandos können entweder nach einem Semikolon auf den Zeilen der Abhängigkeiten oder auf einer mit einem Tabulatorzeichen "HT" beginnenden Zeile, die unmittelbar der Abhängigkeitszeile folgt, stehen.
Eine Abhängigkeitszeile kann einen oder zwei Doppelpunkte ":" enthalten. Der Name eines Zieles kann auf mehr als einer Abhängigkeitszeile auftreten, aber alle diese Zeilen müssen den Doppelpunkt gleichartig, d.h. einfach oder doppelt verwenden. Damit hat es folgende Bewandtnis.

1. Im Normalfall, wenn ein einfacher Doppelpunkt notiert wird, darf höchstens eine dieser Abhängigkeitszeilen eine Folge von Kommandos enthalten. Wenn das Ziel veraltet ist, infolge irgendwelcher Abhängigkeiten, die auf irgendwelchen dieser Zeilen stehen, und eine Folge von Kommandos spezifiziert wurde (vielleicht sogar nur eine leere, die einem Semikolon oder Tabulatorzeichen folgt), wird sie ausgeführt. Wenn dies nicht der Fall ist, wird die

standardmäßige Regel für diesen Fall aufgerufen.

2. Im Fall zweier Doppelpunkte kann jeder Abhängigkeitszeile eine Folge von Kommandos zugeordnet werden. Wenn ein Ziel veraltet ist, wegen irgendeines der Files auf einer so bestimmten Zeile, werden die zugehörigen Kommandos ausgeführt. Eine interne Regel kann eventuell ausgeführt werden. Diese ausführliche Form ist insbesondere bei der überarbeitung von Archiv-Files wertvoll.

Wenn ein Ziel hergestellt werden muß, wird die Folge der Kommandos abgearbeitet. Jede Kommandozeile wird protokollierend auf die Standardausgabe ausgegeben, falls für make nicht die Schweige-Option angegeben wurde (siehe 4. Die Benutzung...) oder die betreffende Kommandozeile mit dem Zeichen kommerzielles a (Commercial at) "@" beginnt. Nach der Ausgabe der Kommandozeile wird sie einzeln, nachdem die Makros substituiert wurden, an Shell übergeben. make beendet die Abarbeitung, wenn ein Kommando einen Fehler signalisiert. Die Fehler werden ignoriert, wenn auf der Kommandozeile für make die Option -i angegeben wurde. Die Fehler werden auch ignoriert, wenn der Zielname ".IGNORE" im Beschreibungs-File steht oder wenn die Kette von Kommandos im Beschreibungs-File mit einem Minus (Hyphen) "-" beginnt. (Einige Kommandos von MUTOS 1700 übergeben einen bedeutungslosen Status.) Weil jedes Kommando einzeln für sich an Shell übergeben wird, muß man bei bestimmten Kommandos aufpassen (z.B. bei cccddd und anderen speziellen Shell-Kommandos), die nur eine Bedeutung innerhalb eines Shell-Prozesses haben. Die Ergebnisse sind bei Ausführung der nächsten Zeile vergessen.
Vor der Ausführung jedes Kommandos werden bestimmte Makros automatisch eingestellt. (Der Abschnitt Beispiel und der Anhang werden zum Verständnis beitragen.) $@ wird auf den Namen des Files gesetzt, das hergestellt werden soll. $? stellt die Kette von Namen dar, die gefunden wurden, die jünger als das Ziel sind. Wenn das Kommando mittels einer impliziten Regel (siehe nachfolgend) generiert wurde, dann ist $< der Name des Files, welches die Aktion verursachte und $* ist der Präfix, den die aktuellen und die abhängigen File-Namen gemeinsam haben.
Wenn ein File herzustellen ist, es aber keine expliziten Kommandos oder relevanten internen Regeln gibt, dann werden die Kommandos, die mit dem Namen ".DEFAULT" verknüpft sind, verwendet. Wenn es solch einen Namen nicht gibt, wird von make eine Meldung ausgegeben und die Abarbeitung wird gestoppt.

4. Die Benutzung des Kommandos make

Das Kommando make kann vier Arten von Argumenten benutzen. [Bes sind Makrodefinitionen, Optionen, Beschreibungs-File-Namen und Ziel-File-Namen. Die Syntax für das Kommando, d.h. für die Kommandozeile, ist:

make [Optionen] [Makrodefinitionen] [Ziele]

Die folgende Zusammenstellung der Kommando-Optionen zeigt, wie die Argumente zu verstehen sind.
Zuerst werden alle Argumente, die Makrodefinitionen sind, (dies sind solche Argumente, die das Zeichen gleich (Equals sign) "=" enthalten) analysiert und die Zuweisung ausgeführt. Die in der Kommandozeile angegebenen Makros machen die im Beschreibungs-File stehenden gleichnamigen Makrodefinitionen ungültig.
Als nächstes werden dann die Optionen abgefragt. Die möglichen Optionen sind:

- -i Ignoriere den vom aufgerufenen Kommando übergebenen Fehlercode. Dieser Zustand besteht auch, wenn der Zielname ".IGNORE", dies ist ein Name eines scheinbaren Zieles, im Beschreibungs-File steht.

- -s Schweige-Zustand. Die auszuführenden Kommandos werden nicht ausgegeben. Dieser Zustand besteht auch, wenn der Zielname ".SILENT" im Beschreibungs-File steht. ".SILENT" ist der Name eines scheinbaren Zieles.

- -r Die internen Regeln sind nicht zu verwenden.

- -n Zustand "Nicht ausführen". Die Kommandos werden auf die Standardausgabe ausgegeben, aber nicht ausgeführt. Zeilen, die mit "@" beginnen, werden auch ausgegeben.

- -t Berühre (touch) die Ziel-Files. (Dies hat zur Folge, daß die Angabe über den Zeitpunkt der Erstellung bzw. der letzten Modifikation durch die gerade aktuelle Zeit ersetzt wird.) Die Kommandos selbst werden nicht ausgeführt.

- -q Frage (question). Das Kommando make übergibt als Status Null, wenn das Ziel-File auf dem neusten Stand ist, und einen von Null verschiedenen Wert, wenn das Ziel-File veraltet ist.

- -p Gibt alle Makrodefinitionen und Zielbeschreibungen aus (print).

- -d Prüfzustand (debug). Es werden detaillierte Informationen über die untersuchten Files und deren Zeitpunkte ausgegeben.

- -f Beschreibungs-Filename. Das dieser Option folgende Argument wird als der Name des Beschreibungs-Files angesehen. Mit dem File-Namen - wird die Standardeingabe angegeben. Wenn keine Option -f angegeben wird, wird als Beschreibungs-File ein File mit dem Namen makefile oder Makefile aus der aktuellen Directory verwendet. Der Inhalt des Beschreibungs-Files kann interne Regeln von make außer Kraft setzen.

Abschließend wird von den übrigen Argumenten in der Kommandozeile angenommen,

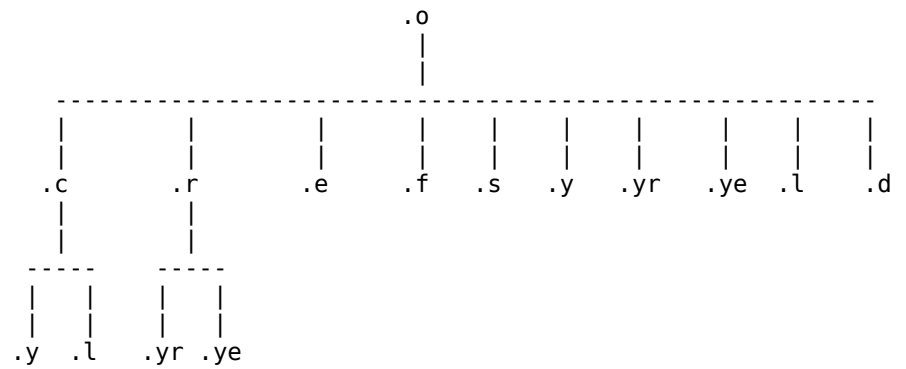
daß sie Namen von herzustellenden Zielen sind. Sie werden in der Reihenfolge von links nach rechts bearbeitet. Wenn es keine solchen Argumente gibt, wird der erste Name aus dem Beschreibungs-File, der nicht mit einem Punkt beginnt, als herzustellendes Ziel angesehen.

4.1. Implizite Regeln

Das Programm make benutzt eine Tabelle der in MUTOS 1700 üblichen Suffixe und eine Menge von Transformationsregeln, die die standardmäßigen Abhängigkeiten und die diesbezüglichen Kommandos bereitstellen. Im Anhang ist diese Tabelle beschrieben und die Mittel, um sie unwirksam zu machen. Die Liste der standard-mäßigen Suffixe ist:

.o	Objekt-File
.c	C-Quell-File
.e	Efl-Quell-File
.r	Ratfor-Quell-File
.f	Fortran-Quell-File
.s	Assembler-Quell-File
.y	Yacc-C-Quellgrammatik
.yr	Yacc-Ratfor-Quellgrammatik
.ye	Yacc-Efl-Quellgrammatik
.l	Lex-Quellgrammatik

Das folgende Diagramm veranschaulicht die standardmäßigen Transformationspfade. Wenn es zwei Pfade gibt, die ein Paar von Suffixen verbinden, dann wird der län-gere nur dann verwendet, wenn das Ziel-File vorhanden ist oder im Beschreibungs-File genannt wurde.



Wenn z.B. das File x.o benötigt wird und es in dem Beschreibungs-File oder der Directory nur ein File x.c gibt, wird x.o durch Compilation aus x.c hergestellt. Wenn es auch noch ein File x.l gäbe, würde die Grammatik x.l zuerst durch den Generator Lex bearbeitet und dessen Ergebnis anschließend compiliert werden. Wenn jedoch kein x.c sondern nur x.l angegeben ist, benutzt make die direkte Verbindung, wie sie in obigem Graphen angegeben ist.
Es besteht die Möglichkeit, die Namen von einigen der standardmäßig verwendeten Compiler oder die Option-Argumente, mit denen sie aufgerufen werden, zu ändern. Um solche änderungen ausführen zu können, werden hier die von make verwendeten Makronamen angegeben. Die Compiler-Namen werden, wenn sie als Kommandos ange-geben werden, klein geschrieben und verbergen sich in den Makros AS, CC, RC, EC, YACC, YACCR, YACCE und LEX. Das Kommando

```
make CC=newcc
```

bewirkt, daß statt des üblichen Kommandos cc für den C-Compiler, das Kommando "newcc" verwendet wird. Die Makros CFLAGS, RFLAGS, EFLAGS, YFLAGS und LFLAGS können gesetzt werden, um damit zu veranlassen, daß für die Kommandos ihre wahlweisen Optionen angebbar sind. So bewirkt

```
make "CFLAGS=-O"
```

daß der optimierende C-Compiler verwendet wird.

4.2. Beispiel

Als Beispiel für die Benutzung von make wollen wir das Beschreibungs-File vorstellen, das verwendet wird, um das Programm make selbst zu warten. Der Code für make besteht aus einer Reihe von C-Quell-Files und einer Yacc-Grammatik. Das Beschreibungs-File sieht wie folgt aus:

```
# Beschreibungs-File für das Make-Kommando

P = lpr
FILES = Makefile version.c defs main.c doname.c
        misc.c files.c dosys.c
        gram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o
         dosys.o gram.o
LIBES= -lS
LINT = lint -p
CFLAGS = -O

make:  $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
        size make

$(OBJECTS):  defs
gram.o: lex.c

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print:  $(FILES)      # print zuletzt Geaendertes
        pr $? | $P
        touch print

test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap
```

```
        lint :  dosys.c doname.c files.c main.c
                misc.c version.c gram.c
                $(LINT) dosys.c doname.c files.c
                main.c misc.c version.c gram.c
                rm gram.c

        arch:
                ar uv /sys/source/s2/make.a $(FILES)
```

make gibt gewöhnlich jedes Kommando aus, bevor es ausgeführt wird. Die folgende Ausgabe entsteht, wenn man sich in der Directory befindet, die nur die Quell-Files und das Beschreibungs-File enthält, und das einfache Kommando

```
make
```

eintippt.

```
 cc  -c version.c
 cc  -c main.c
 cc  -c doname.c
 cc  -c misc.c
 cc  -c files.c
 cc  -c dosys.c
 yacc  gram.y
 mv y.tab.c gram.c
 cc  -c gram.c
 cc  version.o main.o doname.o misc.o files.o
        dosys.o gram.o -lS -o make
 13188+3348+3044 = 19580b = 046174b
```

Obwohl keins der Quell-Files und keine der Grammatiken mit Namen bei den Kommandos direkt im Beschreibungs-File erwähnt werden, findet sie make, indem es seine Suffix-Regeln verwendet und die benötigten Kommandos ausgibt. Die Zahlen in der obigen Ausgabe ergeben sich durch das Kommando "size make", während die protokollarische Ausgabe des Kommandos size selbst durch das Zeichen "@" unterdrückt wird.
Die übrigen Eintragungen im Beschreibungs-File sind hilfreiche Folgen für die Wartung. Die Eintragung "print" sorgt dafür, daß nur die Files ausgegeben werden, die seit dem letzten Kommando "make print" geändert wurden. Ein File print mit der Länge Null wird gewartet, um die Zeit der Ausgabe im File-System von MUTOS 1700 zu führen. Das Makro in der Kommandozeile sammelt dann nur die Namen der Files auf, die verändert wurden, seitdem print berührt (touch) wurde. Die Ausgabe wird über den Spooler "lpr" realisiert. Man kann die Ausgabe auch an ein File "zap" dirigieren, indem man die Definition des Makros P verändert:

```
make print "P = lpr"
        oder
make print "P = cat >zap"
```

5. Hinweise und Warnungen

Erfahrungsgemäß entstehen die meisten Schwierigkeiten aus dem nicht gründlichen Durchdenken und Verstehen der von make zu behandelnden Abhängigkeiten. Wenn ein File x.c eine Zeile ``#include "defs"'' enthält, dann hängt eben das Objekt-File x.o von defs ab, und nicht das Quell-File x.c. (Wenn nämlich defs verändert wird, ist nicht das File x.c zu behandeln, sondern das File x.o muß erneut hergestellt werden.)
Um festzustellen, was make tun würde, ist die Option -n sehr hilfreich. Das Kommando

```
make -n
```

befiehlt diejenigen Kommandos auszugeben, die make ausführen lassen würde, ohne die Zeitpunkte der betroffenen Files zu verändern. Wenn es nach einer änderung eines Files absolut sicher ist, daß sich diese änderung auf bestimmte Files nicht inhaltlich auswirkt, sich aber andererseits eine völlig überflüssige Bearbeitung durch Kommandos ergibt, weil dies die Zeitauswertung von make veranlaßt, kann man (um die Zeit dieser unnötigen Bearbeitung zu sparen) die Option -t benutzen, um e im File-System von MUTOS 1700 eingetragenen Zeiten der betroffenen Files durch die aktuelle Zeit zu ersetzen. Das Kommando

```
make -ts
```

("touch silently") bewirkt, daß die relevanten Files "aktualisiert" werden. Es muß völlig klar sein, daß mit dieser Möglichkeit sehr sorgfältig umgegangen werden muß, da mit dieser Operation die ursprüngliche Absicht von make hintergangen wird und die betreffenden vorangegangenen zeitlichen Beziehungen zerstört werden.
Die Prüfoption -d (debugging) veranlaßt make, eine sehr ausführliche Beschreibung davon auszugeben, was es gerade macht, einschließlich der File-Zeiten. Die Ausgabe ist sehr wortreich und ist nur als letzter Ausweg zu empfehlen.

Anlage A Suffixe und Transformationsregeln

Damit dem Programm make bekannt ist, welche Suffixe von Interesse sind und wie ein File mit einem bestimmten Suffix in ein File mit einem anderen bestimmten Suffix zu transformieren ist, sind diese Informationen in einer internen Tabelle gespeichert. Diese Tabelle hat die Form eines Beschreibungs-Files. Wenn die Option -r benutzt wird, wird diese Tabelle nicht verwendet.
Die Liste der Suffixe hat den Namen ".SUFFIXES". Wenn make ein File sucht, dann wird es mit jedem der Suffixe aus der Liste gesucht. Sobald ein File gefunden wird und es eine Transformationsregel dafür gibt, arbeitet make so, wie es zuvor beschrieben wurde. Der Name einer Transformationsregel besteht aus der Verkettung zweier Suffixe. So ist z. B. der Name der Regel, mit der ein File mit dem Suffix .r in ein File mit dem Suffix .o transformiert wird, .r.o. . Wenn die Regel vorhanden ist und keine explizite Kommandofolge im Beschreibungs-File des Nutzers angegeben wurde, dann wird die Folge von Kommandos der Regel .r.o verwendet. Wenn ein Kommando unter Verwendung dieser durch die Suffixe bestimmten Regeln generiert wird, dann gibt der Makro $* den Wert des Stammes (alles außer dem Suffix) des Namens von dem herzustellenden File an. Der Makro $< gibt in diesem Fall den Namen der Abhängigkeit an, die die Aktion verursachte.
Die Reihenfolge der Suffixe in der Liste ist signifikant. Die Liste wird von links nach rechts gemustert. Der erste Name, der gebildet wird und für den sowohl ein File existiert als auch eine damit im Zusammenhang stehende Regel, wird verwendet. Der Nutzer kann an die Suffixliste weitere Suffixe anfügen, indem er in seinem Beschreibungs-File nach ".SUFFIXES" diese weiteren Suffixe angibt. Steht nach ".SUFFIXES" nichts, so wird die aktuelle Liste gelöscht. Letzteres ist notwendig, wenn die Reihenfolge der Suffixe geändert werden soll.

Nachfolgend werden einige der von make standardmäßig verwendeten Regeln angegeben:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as -
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
    $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
    $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
    $(AS) -o $@ $<
.y.o :
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.y.c :
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@
```