2

2. Die Befehle

Das Herz eines jeden Computers ist die Befehlsliste. RTA kennt 96 Befehle. Die meisten Befehle führen natürlich Rechenoperationen aus. Außerdem gibt es Sprungbefehle-, Ein-/Ausgabebefehle, Pointerbefehle, Systembefehle sowie die Pseudobefehle.

2.1 Rechenbefehle

2.1.1 Die vier einfachen Befehle

```
        clr     a
```

Der einfachste Befehl ist der Lösch- oder Clear-Befehl **clr.** Der clear setzt den Wert einer Variablen auf Null.

```
        mov     a       b
```

Der häufigste Befehl ist der Move-Befehl **mov,** auch Transportbefehl genannt. Mit diesem Befehl wird ein Wert von einem Speicherplatz auf einen anderen kopiert, z. B. der Zahlenwert einer Variablen „b“ nach „a“. In RTA gilt, dass ein Operand, der *geschrieben* wird, immer an erster Stelle steht. Folglich operiert der Move von rechts nach links: „b“ wird gelesen, „a“ geschrieben. Bei der PDP-11 standen die Operanden andersherum.

```
        inc     a
        dec     a
```

Der Inkrementbefehl **inc** erhöht eine Zahl um 1. Der Dekrementbefehl **dec** vermindert sie um 1.

2.1.2 Grundrechenarten

```
        add     a       b               ; Addiere b zu a
        sub     a       c               ; Subtrahiere c von a
        mul     a       d               ; Multipliziere a mit d
        div     a       e               ; Teile a durch e
```

Nur mit „inc“ und „dec“ zu rechnen, wäre sehr umständlich. Für die Grundrechenoperationen gibt es darum **add, sub, mul** und **div.** Die Schreibweise „`add a 10`“ entspricht also der Schreibweise „`a=a+10`“ höherer Programmiersprachen.

2.1.3 Höhere Rechenarten

```
        power   a       3               ; Kubikzahl von a
        root    a       2               ; Quadratwurzel aus a
```

Mit den Befehlen **power** und **root** kann RTA Potenzen und Wurzeln berechnen. Bitte den Exponenten 2 insbesondere bei der Quadratwurzel nicht vergessen. Wurzeln sind in RTA (im Gegensatz zu anderen Programmiersprachen) nicht selbstverständlich Quadratwurzeln. Ohne Exponent würde er eine „nullte“ Wurzel rechnen.

2.1.4 Exponential- und Logarithmusfunktionen

```
        exp     a                       ; berechnet e hoch a
        exp10   a                       ; berechnet 10 hoch a
        exp2    a                       ; berechnet 2 hoch a
        expx    a       b               ; berechnet b hoch a
```

Im Gegensatz zu „klassischem Assembler“ unterstützt RTA auch Operationen höherer Mathematik. RTA kennt vier Befehle, mit denen Exponentialfunktionen berechnet werden können, und zwar mit den Basen e, 10 und 2, sowie mit einer frei wählbaren Basis *b.*

```
        log     a                       ; Basis e  (natürlich)
        log10   a                       ; Basis 10 (dekadisch)
        log2    a                       ; Basis 2  (dyadisch)
        logx    a       b               ; beliebige Basis b
```

Die Umkehrfunktionen der Exponentialfunktionen sind die Logarithmen. **log10** wandelt z. B. eine 1000 in eine 3 um.

2.1.5 Winkel- und Arkusfunktionen

```
        sin     a                       ; Sinus
        cos     a                       ; Cosinus
        tan     a                       ; Tangens
        cot     a                       ; Cotangens
        sec     a                       ; Secans
        csc     a                       ; Cosecans
```

Außer den 4 Haupt-Winkelfunktionen kennt RTA die beiden Winkelfunktionen Secans **sec** und Cosecans **csc**. Es gilt: sec α = (1/cos α) und csc α = (1/sin α).

```
asin    a    b              ; Arcussinus
acos    a    b              ; Arcuscosinus
atan    a    b              ; Arcustangens
acot    a    b              ; Arcuscotangens
asec    a    b              ; Arcussecans
acsc    a    b              ; Arcuscosekans
```

Die Arkusfunktionen sind die Umkehrfunktionen der Winkelfunktionen. Die RTA-Arkusfunktionen bieten mit FAI („full angle inversion“) eine praktische Erweiterung gegenüber ihren Vorbildern aus der Mathematik.

Exkurs FAI („full angle inversion“)

Gewöhnliche Arkusfunktionen liefern Werte lediglich aus einem Bereich von nur 180°. Für viele Berechnungen z. B. auf der Erdoberfläche, ist es aber erforderlich, einen über volle 360° umlaufenden Winkel aus Winkelfunktionswerten ermitteln zu können. Dabei kann oft sowohl die Funktion (z. B. der Sinus) als auch die Kofunktion (der Cosinus) bereitgestellt werden. RTA-Arkusbefehle verfügen nun über einen zusätzlichen zweiten Operanden *b,* auf dem der Befehl den Kofunktionswert entgegennehmen kann. Mit dessen Hilfe kann der Definitionsbereich auf –180° bis +180°, also volle 360° erweitert werden.

Die Regel lautet: Beim Arkuskosinus bewirken negative b-Werte die Negation des Winkels. Dadurch werden Werte 180°…0° zu Werten –180°…0°. Beim Arkussinus bewirken negative b-Werte eine Spiegelung. Positive Winkel 0°…90° werden zu 180°…90°, negative Winkel 0°…–90° zu –180°…–90°.

Wie man erkennen kann, kommt es nur auf das Vorzeichen des b-Wertes an. Umlaufende **atan**, **atan**, **asec** und **acsc** funktionieren analog. Bei der Tangensgruppe muss der b-Wert den Kosinus bzw. Sinus (und nicht Kotangens bzw. Tangens) enthalten. Das liegt in der 180°-Periodizität des Tangens begründet. Hier der Überblick, wie die b-Operanden aller Funktionen ermittelt werden können:

Befehl	b-Operand	Als b-Operand ebenfalls möglich
asin	cos	sec
acos	sin	csc
atan	cos	sec
acot	sin	csc
asec	csc	sin
acsc	sec	cos

Ohne b-Operand arbeiten die Befehle mathematisch-klassisch mit 180-Grad-Wertebereich.

2.1.6 Hyperbel- und Areafunktionen

```
        sinh    a                       ; Sinus hyperbolicus
        cosh    a                       ; Cosinus hyperbolicus
        tanh    a                       ; Tangens hyperbolicus
        coth    a                       ; Cotangens hyperbolicus
        sech    a                       ; Secans hyperbolicus
        csch    a                       ; Cosekans hyperbolicus
```

Auch für die Hyperbelfunktionen gibt es in RTA alle sechs Formen einschließlich Hyperbelsekans und-kosekans.

```
        asinh   a                       ; Area Sinus
        acosh   a                       ; Area Cosinus
        atanh   a                       ; Area Tangens
        acoth   a                       ; Area Cotangens
        asech   a                       ; Area Secans
        acsch   a                       ; Area Cosekans
```

Die Areafunktionen sind die Umkehrfunktionen der Hyperbelfunktionen: Wenn *x* = sinh *y* ist, so ist *y* = asinh *x*.

2.1.7 Logische Befehle

Logische Operationen arbeiten nur mit den beiden „Wahrheitswerten“ 0 (= ‚falsch‘) und 1 (=‚wahr‘). Alle Werte ungleich 0 gelten wie die 1 ebenfalls als ‚wahr‘.

```
        bin     a                       ; Logische Identität
```

Der Befehl **bin** gibt bei Eingabewert 0 wieder eine 0 zurück, ansonsten eine 1.

```
        not     a                       ; Logisch Nicht
```

Das Gegenstück zum bin-Befehl ist der Befehl **not**. Dieser macht aus einer 0 eine 1, ansonsten wird eine 0 zurückgegeben. Er ändert (oder negiert) also einen Wahrheitswert von ‚falsch‘ in ‚wahr‘ und von ‚wahr‘ in ‚falsch‘.

```
        and     a       b               ; Logisch Und
```

Der Befehl **and** gibt nur dann eine 1 zurück, wenn beide Eingangswerte, Wert *a* und Wert *b*, ‚Wahr‘ sind. Sofern nur ein Wert auf 0 steht, wird 0 zurückgegeben.

```
        or      a       b               ; Logisch Oder
```

Dessen Gegenstück ist der Befehl **or**, der immer dann eine 1 zurückgibt, wenn entweder Wert *a* oder Wert *b*, 1 sind. Sind beide Werte 0, so wird 0 zurückgegeben.

Beispiele:

```
        mov     a       -10
        bin     a               ; a ist nun 1
        mov     b       0
        bin     b               ; b bleibt 0

        mov     a       1
        not     a               ; not(1) = 0
        mov     b       0
        not     b               ; not(0) = 1
```

```
        mov     a       0
        and     a       0       ; 0 and 0 = 0
        mov     b       1
        and     b       0       ; 1 and 0 = 0
        mov     c       0
        and     c       1       ; 0 and 1 = 0
        mov     d       1
        and     d       1       ; 1 and 1 = 1

        mov     a       0
        or      a       0       ; 0 or 0 = 0
        mov     b       1
        or      b       0       ; 0 or 1 = 1
        mov     c       0
        or      c       1       ; 1 or 0 = 1
        mov     d       1
        or      d       1       ; 1 or 1 = 1
```

RTA ist ein einfacher Assembler. Darum gibt es keine Unterscheidung von „bitweisen“ und „logischen“ Und-/Oder-/Not-Befehlen etc. RTA arbeitet immer „logisch“.

Abschweifung: Weil es keine bitweisen Operationen gibt, gibt es auch keine bitweisen Verschiebefehle, wie in anderen Assemblerbefehlssätzen. Statt „shift a“ notiere man „`mul a 2`“.

2.1.8 Weitere Rechenbefehle

```
        neg     a                       ; Vorzeichenwechsel
```

Der **neg** ändert das Vorzeichen einer Zahl, macht also aus –1 eine +1 und aus einer +1 eine –1. Die 0 bleibt unverändert.

```
        abs     a                       ; Absolutbetrag
```

Der **abs** liefert den absoluten Betrag einer Zahl, d. h. er setzt das Vorzeichen der Zahl immer auf Plus. So werden alle negativen Zahlen positiv:

```
sgn     a                       ; Vorzeichen ermitteln
```

Der **sgn** liefert das Vorzeichen einer Zahl. Das Ergebnis ist bei negativen Zahlen –1, bei Null 0, bei positiven Zahlen +1.

Beispiele:

```
mov     a       -100
neg     a               ; neg(-100) ergibt 100
mov     b       0
neg     b               ; neg(0) ergibt 0
mov     c       1100
neg     c               ; neg(1100) ergibt -1100
```

```
mov     a       -100
abs     a               ; abs(-100) ergibt 100
mov     b       0
abs     b               ; abs(0) ergibt= 0
mov     c       1100
abs     c               ; abs(1100) ergibt 1100

mov     a       -100
sgn     a               ; sgn(-100) ergibt -1
mov     b       0
sgn     b               ; sgn(0) ergibt 0
mov     c       1100
sgn     c               ; sgn(1100) ergibt 1
```

```
round   a                       ; Runden
ceil    a                       ; Aufrunden
floor   a                       ; Abrunden
fix     a                       ; Zur 0 hin runden
frac    a                       ; Nachkommastellen von a
```

Die Befehle **round**, **ceil**, **floor** und **fix** runden auf verschiedene Art. Der Befehl **fix** ist zugleich der Vorkommastellen-Abtrenner. Er besitzt in **frac**, dem Nachkommastellen-Abtrenner sein Gegenstück. Die Beispiele erläutern die z. T. feinen Unterschiede zwischen den Befehlen:

```
mov     a       3.4
round   a               ; a = 3.0
mov     b       3.6
round   b               ; b = 4.0
mov     c       -3.4
round   c               ; c = -3.0
mov     d       -3.6
round   d               ; d = -4.0
```

```
        mov     a       3.4
        ceil    a                       ; a = 4.0
        mov     b       3.6
        ceil    b                       ; b = 4.0
        mov     c       -3.4
        ceil    c                       ; c = -3.0
        mov     d       -3.6
        ceil    d                       ; d = -3.0

        mov     a       3.4
        floor   a                       ; a = 3.0
        mov     b       3.6
        floor   b                       ; b = 3.0
        mov     c       -3.4
        floor   c                       ; c = -4.0
        mov     d       -3.6
        floor   d                       ; d = -4.0

        mov     a       3.4
        fix     a                       ; a = 3.0
        mov     b       3.6
        fix     b                       ; b = 3.0
        mov     c       -3.4
        fix     c                       ; c = -3.0
        mov     d       -3.6
        fix     d                       ; d = -3.0

        mov     a       3.4
        frac    a                       ; a = 0.4
        mov     b       3.6
        frac    b                       ; b = 0.6
        mov     c       -3.4
        frac    c                       ; c = 0.4
        mov     d       -3.6
        frac    d                       ; d = 0.6
```

```
        clip    a       b       c
```

Der **clip** ist ein Dreioperandenbefehl, der einen Wert *a* auf einen Wertebereich *b* … *c* bringt. Der Befehl setzt *a* auf *b*, wenn *a* < *b* ist und auf *c*, wenn *a* > *c* ist. Ansonsten bleibt *a* unverändert. Im Normalfall soll *b* < *c* sein. Dann ist b ein Minimalwert und *c* ein Maximalwert. Der Befehl funktioniert aber auch bei *b* > *c*; *a* wird in jedem Fall in den Bereich zwischen *b* und *c* „geclippt“. Wenn *b* = *c* ist, dann wird immer *b* (= *c*) zurückgegeben.

```
        cmod    a       b       c
```

Der ***cmod*** („clip and modulo“) ist ebenfalls ein Dreioperandenbefehl. Er funktioniert ähnlich wie der ***clip***, nur wird hier mit einer „Sägezahnfunktion“ gearbeitet. Wenn *a* an einer Wertebereichsgrenze den zulässigen Wertebereich verlässt, (z. B. größer als *c* wird), so wird am anderen Ende des Bereiches (z. B. bei *b*) wieder in den Bereich eingetreten. Beispiel:

```
        mov     a        99.8
        cmod    a        80      100             ; a =  99.8
        mov     b        99.9
        cmod    b        80      100             ; b =  99.9
        mov     c       100.0
```

```
        cmod    c       80      100             ; c = 100.0
        mov     d       100.1
        cmod    d       80      100             ; d =  80.1
        mov     e       100.2
        cmod    e       80      100             ; e =  80.2
```

Dieses Verfahren ist für die Überlaufbehandlung sehr praktisch. So wird z. B. bei umlaufenden Monaten über 12 wieder beim Monat 1 fortgesetzt. Nach der Uhrzeit 24 Uhr folgt 0 Uhr und auf Winkel 360° folgt der Winkel 0°.

```
        random  a               ; Zufallswert 0 ... 1 liefern
```

Schließlich liefert der Befehl **random** einen Zufallswert zwischen 0 und 1. Jeder Aufruf liefert einen anderen Wert. Beispiele:

```
        random  a               ; a = 0.584314 ...
        random  a               ; a = 0.489694 ...
        random  a               ; a = 0.128802 ...
```

2.2 Sprungbefehle

Befehle werden normalerweise immer aufeinanderfolgend abgearbeitet, d. h. so, wie sie zeilenweise im Programm stehen. Oft gibt es aber Code, der wiederholt oder aber überhaupt nicht durchlaufen werden soll. Hierfür sind Sprungbefehle erforderlich. Diese werden in Abhängigkeit von Bedingungen („bedingte Sprünge“) oder in jedem Fall („unbedingter Sprung“) ausgeführt. Ein Sprungbefehl führt immer zu einer Marke.

2.2.1 Bedingte Sprünge mit Vergleich

```
        cmpgt   a       b       m       ; Wenn a>b Sprung zu m
        cmpge   a       b       m       ; Wenn a≥b Sprung zu m
        cmplt   a       b       m       ; Wenn a<b Sprung zu m
        cmple   a       b       m       ; Wenn a≤b Sprung zu m
        cmpeq   a       b       m       ; Wenn a=b Sprung zu m
        cmpne   a       b       m       ; Wenn a≠b Sprung zu m
```

„Compare-Befehle“. Diese Befehle vergleichen zwei Werte und springen in Abhängigkeit vom Vergleichsergebnis zu einer Marke. **cmpgt** lies als „compare greather than“, ge = greather equal, lt = less than, le = less equal, eq = equal, ne = not equal.

2.2.2 Bedingte Sprünge mit Test

```
        tstgt   a       m                   ; Wenn a>0 Sprung zu m
        tstge   a       m                   ; Wenn a≥0 Sprung zu m
        tstlt   a       m                   ; Wenn a<0 Sprung zu m
        tstle   a       m                   ; Wenn a≤0 Sprung zu m
        tsteq   a       m                   ; Wenn a=0 Sprung zu m
        tstne   a       m                   ; Wenn a≠0 Sprung zu m
```

Oft wird mit Null verglichen. Hierfür gibt es die Reihe der „Testbefehle“. Die Testbefehle gleichen den Compare-Befehlen, der Vergleichswert „b“ entfällt aber. **tstgt** lies als „test greather than“, ge = greather equal, lt = less than, le = less equal, eq = equal, ne = not equal.

2.2.3 Der unbedingte Sprung

```
        jump    m                           ; Stets Sprung zu m
```

Wenn in jedem Fall gesprungen werden soll, so nutze man den unbedigten Sprungbefehl **jump.**

2.3 Ein- und Ausgabebefehle

Mit einem Programm soll nicht nur gerechnet werden. Ohne die Möglichkeit, Zahlen in das Programm ein- und ausgeben zu können ist die Rechenmaschine gleichsam gehör- und sprachlos. RTA kennt drei Möglichkeiten oder „Geräte“, mit denen Daten ein- und ausgegeben werden können: Dialoge, einen globalen Ausgabetext und Dateien.

2.3.1 Dialog-Ein-/Ausgabe

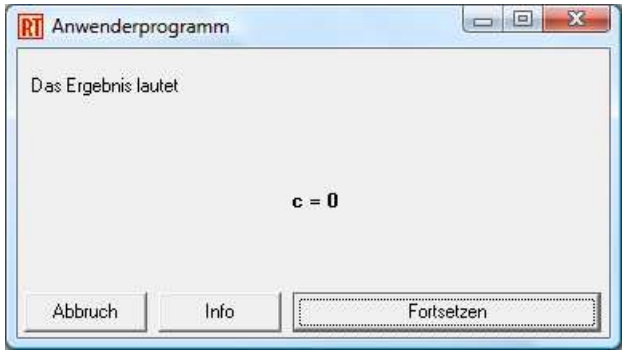
```
        input   a       Bitte~eine~Zahl~eingeben
```

Dialoge kommunizieren mit dem Anwender über Bildschirmein- und Ausgaben. Der hier angegebene Befehl öffnet auf dem Bildschirm das Fenster

Der Anwender wird aufgefordert, eine Zahl eingeben. Diese befindet sich nach Betätigen der Schaltfläche „Übernehmen“ auf der Variablen *a* abgespeichert.

```
output   c        Das~Ergebnis~lautet
```

Das Gegenstück des **input** bildet der Befehl output. Dieser gibt z. B. mit dem Fenster



eine Variable *c* auf dem Bildschirm aus.

```
pause    c        Beginn~des~2.~Teils~der~Berechnung
```

Schließlich gibt es den Befehl **pause,** mit dem man einen Mitteilungstext

an den Bediener übermitten kann.

In allen diesen Fällen hält das Programm an und wartet auf darauf, dass die Schaltfläche „Übernehmen“ bzw. „Fortsetzen“ gedrückt wird.

```
proof    c        Zwischenergebnis
```

Zwei weitere Befehle arbeiten ohne Programmstop. Der Befehl proof gibt wie **output** eine Zahl aus, ohne allerdings Programm zu unterbrechen. Hier wird z. B. eine Variable *c* ausgegeben und mit dem Text „Zwischenergebnis“ kommentiert.

```
info             Zwischenschritt~erreicht
```

Der **info** zeigt einen kurzen Text, hier „Zwischenschritt erreicht“, an, ebenfalls ohne dass das Programm unterbrochen wird. Per **proof** und **info** kann man günstig Testmitteilungen ausgeben.

2.3.2 Der globale Ausgabetext

Der globale Ausgabetext ist ein großer Textspeicher, der global im Hintergrund existiert. Im Prinzip kann man sich den globale Ausgabetext als eine Art Drucker vorstellen, auf dem man Text „ausgeben“ kann.

```
printn  a  b  c  ; Zahl a drucken
```

Der Befehl **printn** („print number“) gibt eine Zahl *a* als Textzeichenkette aus. Dabei werden *b* Vorkomma- und *c* Nachkommastellen erzeugt.

Null Vorkommastellen bewirken eine Ausgabe mit variabler Länge, dann werden genau die erforderlichen Vorkommastellen erzeugt. Null Nachkommastellen bewirken die Ausgabe einer Ganzzahl ohne Nachkommastellen.

Zu den angegebenen Stellen *b* und *c* kommt immer noch eine **Vorzeichenstelle** hinzu und, sofern Nachkommastellen angegeben sind, eine Stelle mit einem **Dezimalpunkt.**
Es gibt folgende 4 Möglichkeiten:

Vor-komma-stellen	Nach-komma-stellen	Zahlenformat	Beispiel mit *a*=6.2811, *v*=3, *n*=2	Stellen (insgesamt)
>1	>1	Zahl mit Kommastellen fester Länge	[•••6.28]	1+3+1+2
>1	0	Ganzzahl fester Länge	[•••6]	1+3
0	>1	Mit Kommastellen, variable Länge	[•6.28]	1+*v*+1+2
0	0	Ganzzahl variabler Länge	[•6]	1+*v*

Es bedeuten: • = Leerzeichen auf der Vorzeichenstelle. • = Leerzeichen statt Vorkommaziffer

Das Vorzeichen ist bei positiven Zahlen ein Leerzeichen, bei negativen Zahlen ein Minus; so werden alle Zahlen immer gleich lang ausgegeben.

Außerdem bietet der **printn** die Möglichkeit mit sog. „erweiterten Formaten“ Vorzeichen, Dezimalzeichen, Vornullen etc. gezielt einzustellen. Dies ist im Anhang erläutert.

```
prints  s  ; Zeichenkette s drucken
```

Der **prints** („print string“) gibt eine Zeichenkette aus. Dabei dient der Symbolname „s“ als Zeichenkette. Hier wird also genau ein einzelnes „s“ ausgegeben.

Der **prints** ist wichtig, um Zahlenausgaben zu gestalten.

Ohne **prints** würde die printn-Befehlsfolge

```
printn    -111.1    3         2
printn    -222.2    3         2
printn    -333.3    3         2
printn    -444.4    3         2
```

lediglich folgenden Text erzeugen:

```
-111.10-222.20-333.30-444.40
```

Es wird wirklich nur das ausgegeben, was der Befehl anweist. Soll tabelliert, in Zeilen gegliedert (und noch eine Überschrift hinzugefügt) werden, so programmiere man beispielsweise

```
prints    Rechenergebnis:\
printn    -111.1    3         2
prints    ~~~
printn    -222.2    3         2
prints    \
printn    -333.3    3         2
prints    ~~~
printn    -444.4    3         2
prints    \
```

Nun werden Leerzeichen und Zeilenumbruchszeichen gesondert hinzugefügt. Man erhält nun:

```
Rechenergebnis:
-111.10   -222.20
-333.30   -444.40
```

```
cls
```

Vor dem ersten „Ausdruck“ ist es sinnvoll, den globalen Ausgabetext zu löschen. Das geschieht mit dem Befehl **cls** („clear screen“).

```
save      ergebnis          typ
```

Mit dem **save** kann man den gesamten Ausgabetext in einer Datei „ergebnis.txt“ ablegen, *ergebnis* ist hier der Symbolname, der als Zeichenkette benutzt wird.

An den Dateinamen wird standardmäßig der Dateityp „.txt“ angehängt, so dass Dateiname „ergebnis.txt“ entsteht. Mit dem 2. Parameter, *typ,* kann ein anderer Dateityp, als „.txt“ gewählt werden. Ein gültiger Dateityp wird daran erkannt, dass sein erstes Zeichen ein Buchstabe ist.

Aus Betriebssystemgründen sollen im Dateinamen nur Buchstaben, Ziffern, sowie die Zeichen „_“, „(“„)“ und „$“ stehen. Buchstaben werden in Kleinbuchstaben umkodiert, Fremdzeichen in den Unterstrich.

2.3.3 Ein- und Ausgabe in Dateien

Variablen können auch in Dateien geschrieben und aus Dateien gelesen werden.

```
write   a       b       typ
read    a       b       typ
```

Der **write** schreibt den Wert der Variablen *a* in eine Datei „a.dat“.
Der **read** liest eine Datei „a.dat“ und ordnet den ermittelten Wert der Variablen *a* zu.

Für die Speicherung von Feldern (s. u.) gibt es den 2. Parameter *b*, der ein Längenparameter ist. Mit dessen Hilfe können mehrere aufeinander folgende Variablen übertragen werden. Dabei wird jede Variable in einer Zeile der Datei abgelegt. Es wird immer ein Wert mehr übertragen, als im Längenparameter angegeben ist. Im Normalfall – wenn also b fehlt – wird Länge 0 angenommen. Damit ergibt sich der Transfer einer Einzelvariablen automatisch als Standardfall. Ist aber ein Längenparameter *b*>0 angegeben, so werden *b*+1 Variablen in ihrer Reihenfolge in der Symboltabelle übertragen.

Mit dem 3. Parameter *typ* kann (ähnlich, wie beim save-Befehl) ein anderer Dateityp, als „.dat“ gewählt werden. Ein gültiger Dateityp wird daran erkannt, dass sein erstes Zeichen ein Buchstabe ist. Auch hier gilt: Aus Betriebssystemgründen dürfen im Dateinamen nur Buchstaben, Ziffern, „_“, „(“, „)“ und „$“ stehen. Alle Buchstaben werden in Kleinbuchstaben umkodiert, Fremdzeichen, auch Umlaute in den Unterstrich.

Anmerkung: Unterschiedliche Groß- und Kleinschreibung in Symbolnamen ist zwar möglich, aber ein wenig gefährlich. Dann muss nämlich darauf geachtet werden, dass es zu keinen Verwechslungen kommen kann. Eine Variable „F“ kann durchaus als Datei gespeichert werden. Diese würde dann aber den Namen „f.dat“ erhalten. Nur darf dann natürlich nicht gleichzeitig eine andere Variable „f“ als Datei gespeichert werden.

2.4 Pointerbefehle

Normalerweise wird ein Operand durch seinen Symbolnamen angesprochen. Diese Zugriffsart heißt auch Direktzugriff. Darüberhinaus gibt es den Indirektzugriff oder Pointerzugriff, bei dem ein Operand über die Symboladresse, als „Pointer“ angesprochen wird.

```
adrof   p       v
```

Der Befehl **adrof** bereitet den Indirektzugriff vor. Er trägt die ***Adresse*** eines Symbols mit dem ***Symbolnamen*** v auf dem ***Symbolwert*** von p ein. Das Symbol *p* wird damit zu einem Pointer auf *v*.

```
        put     p       q       v       ; Schreiben per Pointer
        get     v       p       q       ; Lesen per Pointer
```

Nun kann mit den Befehlen **put** und **get** indirekt geschrieben und gelesen werden.

Die Variable „p“ ist der Pointer *p*. Die Variable „q“ heißt Offset *q.* Der Offset q sei in erster Näherung immer Null. Man notiere ihn zunächst mit dem Leersymbol „.“.

Der Befehl „`put p q v`“ überträgt den ***Wert*** des Symbols mit dem ***Namen*** „v“ auf den ***Wert*** des Symbols mit der ***Adresse*** „p+q“. Man sagt: Er schreibt v *indirekt* nach p.

Der Befehl „`get v p q`“ überträgt den ***Wert*** des Symbols mit der ***Adresse*** „p+q“ auf den ***Wert*** des Symbols mit dem ***Namen*** „a“: Er liest p *indirekt* nach v.

Der Pointerzugriff ist etwas für fortgeschrittene Programmierer und wird ausführlich im Kapitel „Pointer und Felder“ behandelt.

2.5 Systembefehle

```
        init
```

Der **init** bereitet die Maschine auf ein Programm vor. Der Befehl wird aber beim Assemblieren automatisch als erster Befehl in den Code eingefügt, man ihn nicht weiter zu beachten.

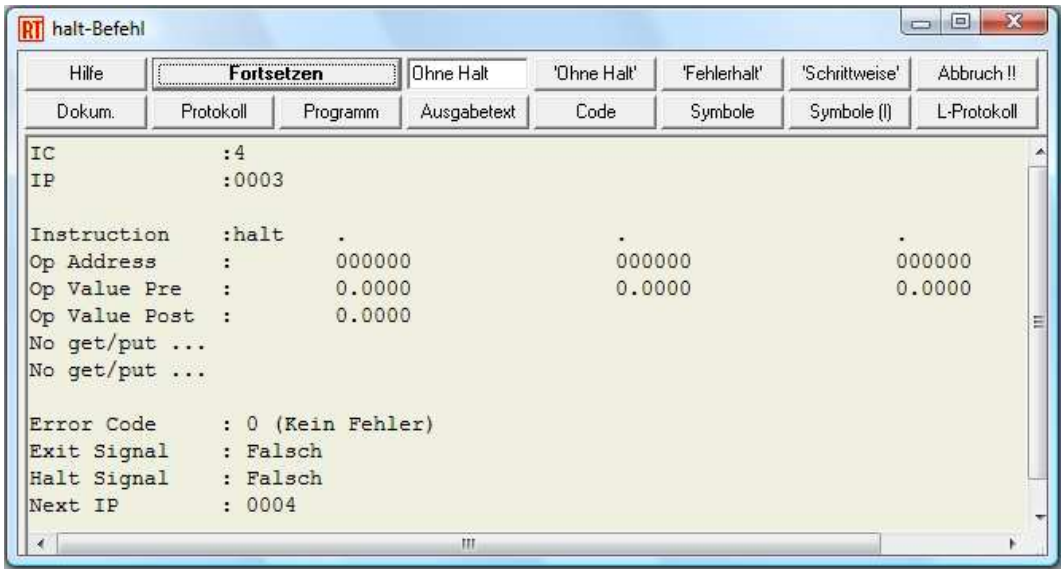
```
        nop
```

Der **nop** – „no operation“ – macht überhaupt nichts (außer dass er wenig Rechenzeit verbraucht). Eigentlich gibt es den nop-Befehl nur, weil er ein Klassiker ist, den jedem Assembler-Codesatz enthält.

```
        halt
```

Der **halt** ist ebenfalls ein Klassiker. Er unterbricht die Programmabarbeitung und hält die Maschine an.

Es öffnet sich das „Prozessorfenster“:



Halt-Befehle waren früher wichtig, als es noch keine Debugger gab und man, um einen Fehler zu finden den Prozessor „zu Fuß“ anhalten musste. Du hast da anstatt eines Breakpoints einfach einen Halt in den Code reingeschrieben und wenn er da reinrauschte, blieb der Rechner einfach — stehen. Da war dann wirklich Ruhe auf der Maschine. Dann konntest du den Hauptspeicher aufmachen und nachgucken. Es ist für jeden Assembler-Befehlssatz unbedingte Ehrenpflicht, über einen Halt-Befehl zu verfügen. Dessenungeachtet gilt aber:

In einem fertigen Programmen hat ein HALT-Befehl nichts zu suchen!

```
mode a
```

Der **mode** erlaubt es, einen von 3 Testmodi einzustellen:

• *a* = 0: Modus ‚Ohne Halt‘: Das Programm läuft durch und hält nie an (außer beim Halt-Befehl).
• *a* = 1: Modus ‚Fehlerhalt‘: Das Programm hält in Fehlerfällen an.
• *a* = 2: Modus ‚Schrittweise‘: Das Programm hält nach jedem Befehl an. Das ist zum Austesten eines Programmes sehr hilfreich. So kann man jeden einzelnen Befehl kontrollieren. Aber auch hier gilt: In einem ausgetestetes Programm hat ein „`mode 2`“ nichts zu suchen. Man sollte dann immer mit Modus 0 arbeiten.

```
err a m
```

Eine andere Fehlerverfolgungsmöglichkeit bietet der Fehlerbefehl **err.** Der err-Befehl erlaubt zwei Reaktionen. Er überträgt den aktuellen Fehlercode auf eine Variable *a* und springt in Fehlerfällen zu einer Marke *m*. Der Fehlercode ist Null, wenn der vorherige Befehl fehlerfrei abgearbeitet worden ist, ansonsten ungleich Null.

So lassen sich Fehlerbehandlungen organisieren, z. B.:

```
          div       x         y
          err       e         Fehlerroutine
          ...
Fehlerroutine:
          output    e         Fehler!~y~war~wohl~0.~Fehlercode:
```

Wenn der Fehlercode nicht benötigt wird, gebe man als 1. Operand das Leersymbol „.“ an. Soll nicht gesprungen werden, so gebe man das Leersymbol als 2. Operand an.

Anmerkung: Bis Ausgabe 4.1.269 wurde die Funktionalität dieses Befehls durch 2 Befehle (errcode und errjump) realisiert. Diese Befehle werden ab 4.1.470 nicht mehr dokumentiert. Bitte nicht mehr benutzen.

```
          exit
```

Der letzte Systembefehl ist sehr wichtig: Der exit-Befehl beendet die Programmabarbeitung.
Die Abarbeitung eines Programmes sollte immer mit dem exit-Befehl enden, denn sonst gibt es nur noch den Absturz.

2.6 Pseudobefehle

Pseudobefehle sind Befehle, die nur bei der Aufbereitung des Programmes („zur Assemblerzeit“, s. u.) bedeutsam sind und quasi vor dem Programmstart helfen, das Programm zu organisieren. Sie erzeugen keine Befehlsanweisungen im Code, laufen infolgedessen nicht „auf der Maschine“. Sie sind in gewissem Sinne „unvollständige“ Befehle – eben Pseudobefehle.

```
          _name     Programmname
```

Mit dem Pseudobefehl **_name** kann dem Programm ein Programmname „Programmname“ gegeben werden.

```
          _var      a
```

Mit dem **_var** wird der Speicherplatz einer Variablen, z. B. „a“ in der Symboltabelle bereitgestellt. Man sagt auch, „die Variable *a* wird deklariert.“

Nun ist dies nicht unbedingt erforderlich, denn RT ist eine sog. selbstdeklarierende Programmiersprache. Neue, unbekannte Symbole werden beim Assemblieren automatisch in die Symboltabelle eingetragen. Das ist bei Fortran, Basic und vielen Skriptsprachen ebenfalls der Fall.

Selbstdeklaration spart Schreibarbeit, hat aber einen großen Nachteil. Man braucht nur einmal statt „A0“ etwa „Ao“ zu tippen und schon entsteht ein neues Symbol Ao. Dies wird sofort mit Null initialisert und ist voll verwendbar. Das ist ein wenig gefährlich, denn nun rechnet er, statt mit A0, plötzlich mit Null weiter. Schon hat man einen Fehler im Programm.

Der Ausweg heißt: Symbole per _var-Befehl deklarieren. Die Vertipper sammeln sich so am Ende der Symboltabelle an. Dort ist das unbekannte „Ao“ leicht aufzuspüren.

Symboldeklarationen machen Programme auch leichter lesbar. Man stellt alle Symbole vor, kann sie in Gruppen gliedern und mit Kommentar erläutern. Ein kleineres Programmen ist ohne Deklaration schnell einmal zum Laufen gebracht. Bei längeren Codes wird hingegen Deklaration empfohlen.

```
        _dim    F       100
```

Der **_dim** ähnelt dem _var-Pseudobefehl. Der **_dim** erzeugt aber nicht nur eine einzelne Variable, sonder ein Feld mit mehreren Elementen. Dieser Befehl erzeugt ein Feld F mit der Länge, hier z. B. 100. Weiteres hierzu siehe das Kapitel „Pointer und Felder“.

Im Gegensatz zu den Variablen besteht bei Feldern Deklarationspflicht. Andernfalls wüsste das Programm ja nicht, wie lang das Feld sein soll. Zu beachten ist auch, dass die Feldlänge wirklich eine Zahl sein muss, also z. B. „100“ oder „256“. Eine Variable, z. B. „k“ würde als 0 aufgefasst und damit ein Feld der Länge Null erzeugen.

```
        _lab    m
```

Der **_lab** definiert Marken. Er erzeugt ein Symbol *m* und trägt auf diesem die aktuell vom Assembler erreichte Codeadresse ein. Auf diese Art entsteht ein Zielpunkt im Code, der z. B. von Sprungbefehlen angesprungen werden kann.

Für diesen Befehl gibt es noch eine andere Schreibweise. Statt „`_lab m`“ lässt sich auch die Schreibweise „`m:`“ benutzen. Genau das ist die übliche Schreibweise einer Markendefinition.

Assemblerprogrammierer aufgepasst: In RTA sind Markendefintionen eigenständige Befehlszeilen. Damit sind Schreibweisen wie „`m: add a b`“ (Marke und Befehl mit Operanden in einer Zeile) nicht möglich. In anderen Assemblersprachen ist das durchaus üblich.

```
        _config 1
```

Der Befehl **_config** ist dafür vorgesehen, der virtuellen Maschine einen Konfigurationsparameter, hier z. B. „1“, mitzuteilen. Der Konfigurationsparameter muss immer eine Zahl sein, Variablen, z. B. „a“ würden als 0 inerpretiert.

```
        _end
```

Der allerletzte Befehl ist der Pseudobefehl **_end.** Der _end zeigt an, dass eine RTA-Programmdatei zu Ende ist.

Was ist der Unterschied zwischen dem **_end** und dem **exit**? — Der **exit** ist der letzte Befehl, der im Programmlauf tatsächlich „auf der Maschine“ abgearbeitet wird. Mit dem **exit** hört das Programm zur Laufzeit auf, „zu laufen“. Die Steuerung wird vom Prozessor an „das Betriebssystem“ zurückgegeben.

Der **_end** hingegen kennzeichnet die letzte Zeile in einer Programmdatei.

Ein Programm mit Verzweigungen kann mehrere exit-Befehle haben, z. B. ein reguläres Programmende und ein Fehlerende. In der Praxis steht der **exit** oft auf der vorletzten Programmzeile. Ein Programm hat aber wirklich immer nur eine letzte Zeile: Den **_end**.

Zum Schluss kommen wir zu der wichtigen Frage, ob man einen **_end** kommentieren darf. Heutzutage würde so etwas gehen und die hängen da dann auch noch 20 weitere Kommentarzeilen dran. Aber so *richtig stylish* ist so etwas nicht.

Wenn da dein Assembler früher das abschließende „d“ in dem **_end** erkannt hat, war wirklich Schluss. Er hat dann noch einen halben Meter leeren Lochstreifen rausgelassen und ist dann stehengeblieben.

Du bist dann zu dem Lochstreifenstanzer rüber, hast den Streifen abgemacht und mit Bleistift den Namen von dem Objektmodul draufgeschrieben. Zusammengerollt, Büroklammer dran, damit er sich nicht wieder aufrollt und rein in die Pappkiste mit der Aufschrift „Software“ auf dem Deckel. Feierabend.

Da kam dann auch nicht einmal ein Punkt dahinter:

```
_end
```