KLEINCOMPUTER

KC 85

M026 FORTH

KLEINCOMPUTER

KC 85

Beschreibung zu M 026 FORTH

veb mikroelektronik
›wilhelm pieck‹
mühlhausen

im veb kombinat mikroelektronik

Wir danken Herrn Dipl.-Ing. Klaus Katzmann für die Implementierung des FORTH-Systems und für die Erarbeitung der Beschreibung.

Gesamtherstellung: VEB Druckerei "Thomas Müntzer" Bad Langensalza

Gen.-Nr. Ri 11/89 V/12/6

veb mikroelektronik "wilhelm pieck" mühlhausen

Inhaltsverzeichnis

0. FORTH-Einführung

Was ist FORTH? Diese Frage kann unter den verschiedensten Gesichtspunkten beantwortet werden.

FORTH ist zunächst einmal eine Programmiersprache, die sich gegenüber einer Vielzahl anderer Programmiersprachen durch eine Reihe besonderer und unkonventioneller Eigenschaften auszeichnet.

FORTH ist erweiterbar. Es gibt einen standardisierten Grundwortschatz, der durch anwendereigene Wortdefinitionen erweitert werden kann, ja sogar erweitert werden muß, um überhaupt zu einem in FORTH geschriebenen Programm zu kommen. Die Programmentwicklung geht vonstatten, indem das Kernvokabular durch weitere Wortdefinitionen ergänzt wird, die dann ihrerseits wieder in neue Worte eingebaut werden können. Dies muß solange fortgesetzt werden, bis ein einziges FORTH-Wort die gegebene Aufgabenstellung realisiert. Hierdurch wird meistens ein hirarchisch gut gegliedertes Programm entstehen.

FORTH ist interaktiv. Der Benutzer des FORTH-System kann völlig ungezwungen im Dialog mit dem Computer neue Wortdefinitionen eingeben, testen, streichen, wieder neu eingeben usw. Fast alle Sprachelemente (Worte) dürfen direkt von der Tastatur aus aufgerufen werden.

FORTH ist maschinennah. Dem Anwender wird nicht soviel Komfort geboten, wie z.B. von BASIC oder FORTRAN. Er wird dafür aber durch eine besondere Flexibilität entschädigt, die es gestattet, Probleme, die sich in anderen Sprachen nur schwerfällig formulieren lassen, "im Spaziergang" zu lösen. Ohne Umwege lassen sich zeitkritische Aufgaben im Maschinencode formulieren.

FORTH ist eine Compilersprache. Dadurch ist eine hohe Verarbeitungsgeschwindigkeit gegeben. Der Compiler verarbeitet jedoch nicht ein komplettes Programm in einem Zug, sondern übersetzt dieses Wortweise, wodurch später jede Komponente eines Programms (jedes Wort) interaktiv aufrufbar bleibt. Die Programmtestung ist dadurch einfach und übersichtlich.

FORTH unterstützt wirkungsvoll die strukturierte Programmierung, indem es nur solche Programmstrukturen erlaubt, die damit im Einklang stehen. Einen GOTO-Befehl gibt es nicht, er ist auch nicht nötig.

FORTH ist nicht nur eine Programmiersprache, sondern ein komplettes Programmentwicklungs-Betriebssystem, das auf eigenen Füßen steht. Die Schnittstellen zur Peripherie sind einfach gehalten und beinhalten im wesentlichen die Ein- und Ausgabe von einzelnen Zeichen (Tastatur und Bildschirm) und von Datenblöcken (Externer Speicher). Dadurch ist FORTH leicht auf andere Computer übertragbar. Alle Komponenten des FORTH-Systems stehen dem Anwender zur freien Verfügung. Er kann damit machen, was er will, er soll und muß das auch, denn schließlich ist er ja für die Effektivität seiner Programme verantwortlich.

FORTH ist eine relativ junge Programmiersprache. Ihr Erfinder ist Charles H. Moore, der sie 1969 am National Radio Astronomy Observatory in Charlottesville (USA) auf einer IBM 1130 implementierte. Er wollte FORTH als Sprache der vierten Generation verstanden wissen. Aufgrund der Eigenschaften des genannten Computers mußte er jedoch von "FOURTH" ein Zeichen weglassen, so daß der Name "FORTH" entstand, der dadurch auch etwas doppeldeutig geworden ist (forth–vorwärts), was jedoch dem Charakter dieser Sprache vollkommen entspricht.

Die hier vorliegende Version ist aus dem Standart der FORTH Interest Group entstanden, dem fig-FORTH. Die FORTH Interest Group vertreibt auf nichtkommerzieller Basis FORTH-Listings und andere FORTH-Literatur für die verschiedensten Mikroprozessoren und Minirechner, so daß FORTH jedem Interessenten zugänglich wird. Zur optimalen Nutzung des KC85 wurden am fig-FORTH Veränderungen und Erweiterungen vorgenommen, die im wesentlichen die Arbeit mit dem Bildschirm (Grafikmöglichkeiten) und dem Externspeicher (Magnetband) betreffen.

1. Modulhandhabung

1.1. Welcher Modulsteckplatz

Der Modul ist zur Nutzung in einem Steckplatz des Systems zu kontaktieren. Der FORTH-Modul kann prinzipiell in jedem Modulsteckplatz betrieben werden. Dies gilt auch für Aufsätze mit weiteren Modulsteckplätzen. Hierbei ist jedoch zu beachten, daß die Modulprioritätskette geschlossen bleibt.
Zur Modulsteuerung im Computer verfügt der Modul über eine Modulprioritätsschaltung. Sind mehrere Module mit gleicher Anfangsadresse eingeschaltet, so sichert die Prioritätsschaltung, daß beim Zugriff des Prozessors nur der Speichermodul aktiviert wird, der sich auf der niedrigsten Modulsteckplatzadresse befindet. Soll im weiteren Verlauf der Modul mit der niedrigsten Priorität (höchste Modulsteckplatzadresse) erreicht werden, müssen alle in der Prioritätskette vorherliegenden Module mit gleicher Anfangsadresse ausgeschaltet sein (zur näheren Erläuterung finden Sie im Abschnitt 1.4. einige Beispiele).
Hieraus ist zu entnehmen, daß erst im Grundgerät der Steckplatz 08 (rechts), dann der Steckplatz 0C (links) und anschließend erst weitere Steckplätze von Erweiterungsaufsätzen in vorgesehener Reihenfolge zu belegen sind.
Soll der FORTH-Modul in einem KC 85/2 auf der Grundlage des im BASIC-Modul M006 befindlichen Betriebssystems genutzt werden, so sind der FORTH-Modul im Schacht 08 und der BASIC-Modul im Schacht 0C zu kontaktieren. Zuerst wird dann der FORTH-Modul entsprechend den folgenden Abschnitten zugewiesen. Danach ist der BASIC-Modul mit

```
JUMP C
```

zu starten.

1.2. Kontaktierung

ACHTUNG!
Das Stecken des Moduls sowie das Entfernen des Moduls aus dem Steckplatz darf nur im ausgeschalteten Zustand des Systems erfolgen!

Der Modul ist durch folgende Handgriffe zu stecken:

1. Den Computer ausschalten.
2. Die Kappe des Modulschachtes durch leichten Druck mit Daumen und Zeigefinger auf die Griffflächen abnehmen.
3. Den Modul bis zum fühlbaren Rasten einschieben (hervorstehender Rand des Moduls liegt unmittelbar an der Gerätewand an).

Zum Entfernen des Moduls aus dem System sind folgende Schritte notwendig:

1. Den Computer ausschalten.
2. Den linken und den rechten Zeigefinger unter den Modulkopf legen und mit den Daumen die seitlich am Modul befindlichen Hebel gleichzeitig nach unten drücken. Dabei rastet der Modul aus und wird etwa einen Zentimeter aus dem Gerät herausgeschoben. Nun den Modul aus dem Schacht entnehmen.
3. Die Kappe auf die Schachtöffnung stecken.

1.3. Zuweisung

Mit der Anweisung

```
SWITCH mm kk
```

wird der Modul auf dem Steckplatz mm zugewiesen. Dabei ist mm die Steckplatzadresse und kk das Steuerbyte.
Die Steckplatzadresse mm ist für jeden Steckplatz eines Gerätes in der dazugehörigen Beschreibung angegeben. Im Grundgerät besitzt der rechte Modulschacht die Adresse 08 und der linke die Adresse 0C. Der zweite Parameter der Anweisung SWITCH ist das zweistellige Steuerbyte kk.
Die erste Ziffer des Steuerbytes kk legt die Anfangsadresse für den Modul fest und die zweite den Betriebszustand.

Betriebszustände:

Für den FORTH-Modul unterscheiden wir zwei Betriebszustände.

1. INAKTIV - Diode leuchtet nicht. Der Modul ist für den Prozessor nicht verfügbar.
2. AKTIV - Diode leuchtet. Der Modul ist arbeitsbereit.

1.4. Adressierung

Der Speicherbereich des FORTH-Moduls ist auf die Anfangsadresse C000H zu legen. Soll der 8-KByte-Modulspeicherblock in das KC85/3-System eingebunden werden, so ist vorher der auf dieser Adresse befindliche BASIC-Interpreter INAKTIV zu schalten. Dies wird mit der Anweisung SWITCH durch den Parameter kk realisiert. Die erste Ziffer enthält die Tausenderstelle des angesprochenen Speicherplatzes (also C). Die zweite Ziffer des Steuerbytes kk gibt den Betriebszustand an. Wie bereits festgestellt, gilt hierbei:

Ziffer	Zustand
0	INAKTIV
1	AKTIV

Im Rahmen des modularen Gesamtaufbaus des Systems besitzt der BASIC-Interpreter des KC85/3 die "Steckplatzadresse" 02.
Somit lautet die Anweisung zum INAKTIV schalten des BASIC-Interpreters:

```
SWITCH 02 00 .
```

Steckt der FORTH-Modul nun z.B. im Schacht 08, so kann er durch die Anweisung:

```
SWITCH 08 C1
```

zugewiesen werden.
Wurde der FORTH-Interpreter durch Betätigung der RESET-Taste verlassen, so ist vor dem erneuten Start durch REFORTH oder FORTH der BASIC-Interpreter wieder INAKTIV zu schalten.

1.5 Modulstrukturbyte

Jeder Modul besitzt zur Kennung ein zugeordnetes Strukturbyte. Der FORTH-Modul besitzt das Strukturbyte FBH . Über die SWITCH-Anweisung wird das Strukturbyte wie folgt angefordert:

```
SWITCH 08              Eingabe Modulsteckplatz-Adresse.

08  FB  C1            Antwort des Computers
|   |   ||
|   |   |+------------ Betriebszustand (AKTIV)
|   |   +------------- Anfangsadresse (C000H)
|   |
|   +----------------- Strukturbyte
|
+--------------------- Modulsteckplatzadresse
```

2. Die Arbeitsweise von FORTH

2.1. Die Vorbereitung des Computers

Wurde der Modul wie im Kapitel 1 beschrieben aktiviert, erscheinen beim Auflisten des CAOS-Menüs zusätzlich die Anweisungen

```
%FORTH
%REFORTH
```

Nach Aufruf von FORTH meldet sich der Computer mit

```
KC-FORTH
```

Aufrufen können Sie FORTH auf zwei verschiedene Arten:

1) Sie bewegen den Cursor auf die Zeile, auf der FORTH steht und drücken dann ENTER oder

2) Sie schreiben in einer Zeile nach dem Promtzeichen "%" oder ">" das Wort FORTH und drücken nun die ENTER-Taste.

Hier und in weiteren Ausführungen des Buches bezeichnet das Wort "[ENTER]" die Betätigung der ENTER-Taste (rechts unten auf der Tastatur). Der danachfolgende vom Computer ausgegebene Text wird im Buch durch Unterstreichung gekennzeichnet.

```
KC-FORTH
```

FORTH ist nun zur Arbeit bereit.

2.2. Die interaktive Arbeitsweise von FORTH

Nachdem FORTH sich mit seiner Systemausschrift

```
KC-FORTH
```

gemeldet hat, geschieht zunächst nichts weiter. Das ist ganz normal, denn der Computer "weiß" ja noch gar nicht, was er machen soll. Genau genommen macht er aber doch etwas: er wartet darauf, daß Sie eine Taste auf der Tastatur betätigen. Wenn Sie dies tun, erscheint das entsprechende Zeichen sofort auf dem Bildschirm. Diesen Vorgang nennen wir Echo. Sollte das Echo einmal fehlen, gibt es zwei mögliche Gründe:

1) Der Computer bearbeitet gerade eine Aufgabe, die Sie ihm gestellt haben, Dann müssen Sie solange warten, bis er diese beendet hat.

2) Es liegt ein mehr oder weniger schwerwiegender Programmfehler vor. Sie müssen den FORTH-Modul nach betätigen der Taste RESET starten.

Der Computer nimmt nun mit Echo ein Zeichen nach dem anderen von Ihnen entgegen, bis Sie ihm mit der ENTER-Taste mitteilen, daß ihre Eingabe beendet ist. Außerdem bewirkt die Betätigung der ENTER-Taste die Ausführung der in der eingegebenen Zeile enthaltenen Worte. Danach meldet sich der Computer wieder, und zwar, wenn alle Anweisungen korrekt eingegeben waren mit

```
OK
```

anderenfalls mit irgendeiner Fehlermeldung. Nun wird wieder auf eine Eingabezeile von Ihnen gewartet.
Diese Arbeitsweise - Zeile eingeben, Zeile ausführen, Zeile eingeben usw. - heißt interaktiv. Das wollen wir gleich einmal ausprobieren.
FORTH ist nun also bereit, Ihre Wünsche zu erfüllen. Nun betätigen Sie einfach die ENTER-Taste. Sie erhalten die Antwort

```
OK
```

Das ist richtig so, denn Sie haben von FORTH nichts verlangt, was es auch prompt ausgeführt hat. Schreiben Sie nun:

```
CR (ENTER)
OK
```

FORTH schreibt das "OK" eine Zeile tiefer und zwar an den Zeilenanfang. Mit CR haben Sie ein erstes Element der Programmiersprache FORTH kennengelernt. Bei FORTH werden solche Sprachelemente einfach Worte genannt.
CR (carriage return - Wagenrücklauf) veranlaßt den Computer, den Cursor auf den Anfang der nächsten Zeile zu setzen. Sie können das Wort CR auch durch Leerzeichen getrennt mehrmals in eine Zeile schreiben, also folgendermaßen:

```
CR CR CR (ENTER)


OK
```

CR wurde nun dreimal hintereinander ausgeführt.
Ihre eingegebene Zeile darf nur FORTH-Worte und Zahlen enthalten. Folgendes passiert, wenn Sie davon abweichen:

```
CR CR OTTO CR CR CR (ENTER)

OTTO? MSG # 0
```

Die Ausführung der Zeile wurde nach OTTO abgebrochen. OTTO ist dem Computer offenbar nicht bekannt. Derartige Worte erhalten Sie mit einem Fragezeichen versehen postwendend zurück. Anschließend erfolgt noch die Ausgabe einer Nachricht, die auf die Art des aufgetretenen Fehlers schließen läßt. MSG ist das abgekürzte englische Wort message, was einer Nachricht oder Mitteilung entspricht. Jede FORTH-Nachricht hat eine Nummer, und jeder Nummer ist ein kurzer Text zugeordnet. Mit der oben erschienenen Mitteilung Nr.0 will der Computer nur seine Ratlosigkeit ausdrücken, da er mit dem ihm angebotenen Text nichts anfangen kann.
Im Gegensatz zu BASIC ist bei FORTH zwischen je zwei Sprachelementen (Worte oder Zahlen) unbedingt ein Leerzeichen zu setzen!
Möchten Sie den Bildschirm löschen, so können Sie dies wie gewohnt durch die Zweitbelegung der HOME-Taste realisieren.

2.3. Der Stack

Wir bieten dem Computer nun eine Zahl an:

```
 77 (ENTER) OK
```

Die Zahl ist offenbar geschluckt worden, aber wohin?
Wir bieten ihm noch eine Zahl an:

```
 23 (ENTER) OK
```

Diese Zahl ist nun auch im Computer. Wir befehlen jetzt dem Computer, uns die beiden Zahlen wieder auf dem Bildschirm darzustellen:

```
 . (ENTER) 23 OK
 . (ENTER) 77 OK
```

Der Punkt sieht sehr unscheinbar aus. Er ist aber ein vollwertiges FORTH-Wort und hat die Aufgabe, eine Zahl aus dem Zahlenspeicher herauszuholen und auf dem Bildschirm darzustellen.
Wir müssen uns nun noch den besonderen Eigenschaften des Zahlenspeichers widmen. Sie haben sicher bemerkt, daß sich zwei Zahlen gleichzeitig im Zahlenspeicher befunden haben. Wir mußten dem Computer aber nicht mitteilen, welche der beiden Zahlen auf dem Bildschirm zu erscheinen hatte. Er hat uns einfach die letzte Zahl zuerst wieder ausgegeben.
Unser Zahlenspeicher kann mit einem Bücherstapel verglichen werden. Wenn Sie die gestapelten Bücher ins Regal stellen wollen, werden Sie sicher das oberste Buch herunternehmen, weil es am bequemsten erreichbar ist.
Ein Zahlenspeicher, der in der beschriebenen Weise organisiert ist, heißt sehr anschaulich Stapelspeicher, in der vorwiegend englischsprachigen Computerwelt hat sich die Bezeichnung "Stack" eingebürgert, was zu deutsch auch Stapel heißt. Das Vorhandensein eines Stack zur Speicherung von Zahlen ist keine Eigenheit von FORTH. Es ist aber FORTH-typisch, daß sie selbst darüber entscheiden dürfen (und auch müssen), welche Zahlen sich wann auf dem Stack zu befinden haben und wie sie zu verarbeiten sind. Von Ihnen wird bei FORTH verlangt, dafür zu sorgen, daß sich stets die richtigen Zahlen in der richtigen Reihenfolge auf dem Stack befinden. Für den Computer bedeutet das eine erhebliche Arbeitserleichterung und damit eine höhere Rechengeschwindigkeit.

2.4. Zeichen und Zahlen - Wir arbeiten mit dem Stack

Der Stack bietet eine elegante Möglichkeit zur Übergabe von Parametern. Die, meisten FORTH-Worte übernehmen Parameter vom Stack bzw. übergeben dort welche. Auch der Punkt übernimmt einen Parameter vom Stack, nähmlich die Zahl, welche auszugeben ist. Im Computer und damit auch im Stack liegen alle Zahlen als Binärzahl vor. Das ist für den Computer sehr vorteilhaft, denn mit Binärzahlen kann er besonders bequem und schnell rechnen. Mehr darüber im nächsten Kapitel.
Auf dem Bildschirm lassen sich Binärzahlen nicht unmittelbar darstellen . Sie müssen erst in eine Folge von Zeichen umgewandelt werden. Dies wird von einem Rechenprogramm besorgt, das vom Punkt aufgerufen wird. Ein Zeichencode ist eine Zuordnungsvorschrift zwischen Code-Zahlen und Zeichen.

Im Anhang A finden Sie eine Übersicht über den in unserem Computer verwendeten Zeichencode. Beispielsweise ist dem Symbol "*" die Zahl 42 zugeordnet. Alle Zeichensymbole lassen sich nunmehr über ihren Code auf dem Bildschirm darstellen. Wenn FORTH ein Sternchen auf den Bildschirm schreiben soll, muß zunächst die Zahl 42 auf dem Stack vorliegen. Dann kann das Wort EMIT aufgerufen werden, das von dort eine Zahl holt und dann das entsprechende Zeichen im Bildschirm auf der Cursorposition darstellt.
Sehen sie selbst, wie einfach das ist:

```
 42 EMIT (ENTER) *OK
```

Die 42 ist danach vom Stack verschwunden.
Auch Ziffernsymbole werden in dieser Weise codiert:

```
 49 EMIT (ENTER) 1 OK
 50 EMIT (ENTER) 2 OK
```

Das genaue Gegenstück zu EMIT ist das Wort KEY. KEY wartet, bis Sie eine Taste drücken und hinterläßt dann den Tastencode auf dem Stack. Im folgenden Beispiel müssen Sie deshalb nach ENTER noch eine Taste drücken, KEY wartet darauf. Der Punkt gibt den Tastencode wieder aus.

```
 KEY . (ENTER) 65 OK
```

Welche Taste wurde hier gedrückt? Mit EMIT (s.o.) können Sie das schnell herausfinden.
Nun wollen wir ein Zeichen von der Tastatur gleich zum Bildschirm schicken:

```
 KEY EMIT (ENTER) A OK
```

Unsere Beispiele zeigen, daß Parameterübernahme und -übergabe auf dem Stack sehr einfach sein können. Dennoch haben sie auch ihre Tücken, insbesondere dann, wenn mehrere Parameter für verschiedene Worte gleichzeitig auf dem Stack vorliegen.

2.5. Der Textinterpreter

Der Textinterpreter ist ein kleines Stückchen Programm innerhalb von FORTH. Sie sind der Chef, und der Textinterpreter arbeitet wie eine zuverlässige Sekretärin. Stellen Sie sich das bitte folgendermaßen vor:
Wenn Sie eine Zeile eintippen, schreiben Sie einen kurzen und knappen Text über die Aufgaben, die der Textinterpreter für Sie zu erledigen hat. Mit der ENTER-Taste vollziehen Sie gewissermaßen eine Unterschrift, daß Sie das, was Sie eingetippt haben, auch wirklich wollen. Damit übergeben Sie Ihren Text an den Textinterpreter und können nun nachträglich nichts mehr ändern. Sie haben ja „unterschrieben".
Der Textinterpreter ist "froh", daß es nun wieder Arbeit gibt und sucht sich den Anfang vom Text. Von dort an teilt er sich den Text in handliche Portionen ein. Dabei orientiert er sich an den Leerzeichen im Text. Die einzelnen Portionen enthalten deshalb immer nur eine Zahl oder ein Wort.

Der Textinterpreter ist in Ihrem Computer der Chef aller vorhandenen FORTH-Worte. Er hat ein dickes Buch, in dem jedes Wort vermerkt ist, das Wörterbuch. Sie können sich das Wörterbuch selber mal ansehen. Schreiben Sie dazu:

VLIST (ENTER)
TASK DUMP EDITOR PLOT CSAVE VERIFY ... usw.

Erschrecken Sie bitte nicht gleich, den größten Teil dieser Worte werden wir vorläufig nicht benötigen, ein kleiner Teil dieses Wortschatzes ist bereits für viele Anwendungen ausreichend.
Aber zurück zum Textinterpreter:
In der einen Hand hält er eine Portion Ihres Textes (z.B. ein beliebiges FORTH-Wort), mit der anderen Hand blättert er das Wörterbuch von hinten her durch und sucht die entsprechende Eintragung. Hat er sie gefunden, macht er eine kurze Pause und läßt das von Ihnen geschriebene Wort abarbeiten. Anschließend wird alles mit der nächsten Textportion wiederholt. Wird die entsprechende Eintragung vom Textinterpreter nicht gefunden, nimmt er an, daß es sich um eine Zahl handelt und übergibt die Textportion an das Wort NUMBER. Dessen Aufgabe ist das genaue Gegenteil zum Wort. (Punkt). NUMBER erzeugt aus einer Folge von Ziffernsymbolen eine Binärzahl und bringt sie auf den Stack. Sollte dies mißlingen (z.B. beim Wort OTTO), erhalten Sie dieses Wort mit einem Fragezeichen wieder zurück, und der Textinterpreter wartet auf ihre nächste Eingabe.

2.6. Der Textinterpreter kann noch mehr

Bis jetzt haben wir den Textinterpreter alle Anweisungen sofort ausführen lassen. Für jedes auszuführende Wort und jede eingegebene Zahl hatte er einen zeitraubenden Suchlauf im Wörterbuch durchzuführen. Nun wollen wir ihm und uns die Arbeit erleichtern. Es ist nämlich möglich, daß wir unsere Anweisungen zusammenfassen, ihnen einen Namen geben und nahtlos an das bisherige Wörterbuch anfügen. Damit erreicht man, daß der Textinterpreter bei den nächsten Suchläufen unser neu definiertes Wort genauso findet, wie alle anderen Worte, die zum FORTH-Kern gehören. Das ist faktisch eine Erweiterung der Programmiersprache. Wir befehlen dem Interpreter, unsere nächsten Eingaben nicht sofort auszuführen, sondern sie nur unter einem von uns ausgedachten Namen zu notieren. Zu diesem Zweck gibt es ein spezielles Wort, den Doppelpunkt. Nun muß es ja noch eine Möglichkeit geben, den Textinterpreter wieder davon abzubringen, alle unsere Eingaben ins Wörterbuch einzutragen. Schließlich wollen wir ja ein neu definiertes Wort auch einmal wieder aufrufen und ausprobieren.
Auch das ist (wie Sie sicherlich ahnen) kein Problem. Wenn unsere Wortdefinition beendet ist, schreiben wir einfach ein Semikolon, und schon können wir uns wieder direkt mit dem Textinterpreter unterhalten. Aber genug der Theorie, jetzt kommt das Experiment.

```
 : OTTO (ENTER)
```

Unser neues Wort heißt OTTO. Nun hat der Textinterpreter den Namen OTTO bereits ins Wörterbuch eingetragen. Wir müssen "nur" noch hinschreiben, was bei Aufruf von OTTO zu tun ist, z.B.:

```
 ." HALLO OTTO ! " CR (ENTER)
```

Der Text zwischen ." (Punkt Anführungszeichen) und dem nächsten Anführungszeichen wird bei Abarbeitung unverändert ausgegeben. Es ist zu beachten, daß zwischen dem ." und dem Text ein Leerzeichen einzufügen ist, welches nicht mit ausgegeben wird.
Das Wort OTTO soll also den kurzen Text "HALLO OTTO" ausgeben. Nun müssen wir dem Textinterpreter noch mitteilen, daß unsere Wortdefinition OTTO beendet ist:

```
; (ENTER) OK
```

Jetzt erhalten wir auch das vertraute OK. Schauen wir einmal mit VLIST nach, ob OTTO auch wirklich im Wörterbuch steht. OTTO müßte als erstes Wort erscheinen, weil das Wörterbuch stets von hinten nach vorn durchsucht wird.
VLIST können Sie durch Drücken einer beliebigen Taste abbrechen.

```
VLIST (ENTER)
OTTO TASK DUMP EDITOR PLOT CSAVE ... usw.
```

Nun können wir OTTO auch aufrufen:

```
OTTO (ENTER) HALLO OTTO !
     OK
```

Da sich OTTO in nichts von anderen FORTH-Worte unterscheidet, kann es natürlich genauso wie diese in neue Wortdefinitionen eingebaut werden. Diesmal schreiben wir die gesamte Definition in eine Zeile:

```
: 3OTTO CR OTTO OTTO OTTO ;  (ENTER) OK
```

3OTTO wird gleich ausprobiert:

```
3OTTO (ENTER)
HALLO OTTO !
HALLO OTTO !
HALLO OTTO !
OK
```

Herzlichen Glückwunsch! Soeben haben Sie Ihr erstes FORTH-Programm geschrieben. Ein FORTH-Programm entsteht durch die Definition neuer Worte, die einfach an das Stammvokabular angefügt werden. Dies wird solange fortgesetzt, bis ein einziges Wort die Aufgabenstellung realisiert.

2.7. Vom Problem zum Programm

Unser Computer soll ein Häuschen auf den Bildschirm zeichnen. Dies soll unsere Aufgabe sein, die wir mit einem FORTH-Programm lösen wollen. Das Häuschen soll nur aus geraden Linien bestehen, die mit dem Wort PLOT leicht auf den Bildschirm gezeichnet werden können. Mit dem Wort INKP legen wir die Farbe für die Grafiken fest. Um einen guten Kontrast zum dunklen Hintergrund zu erreichen, legen wir mit

```
7 INKP
```

die Farbe weiß fest. Wir werden später noch ausführlicher Informationen zu diesem Wort erhalten.

Probieren wir doch gleich einmal aus, wie PLOT arbeitet:

```
 1 100 100 100 50 PLOT (ENTER) OK
```

PLOT hat uns nun eine Linie gezeichnet. Die 5 Parameter, die PLOT übernimmt, entscheiden über die Lage der Linie auf dem Bildschirm, deren Länge und darüber, ob die Linie gezeichnet oder gelösccht werden soll. Doch zunächst noch etwas Grundsätzliches zur Grafik:
Unser Bildschirm ist in 256 (vertikal) mal 320 (horizontal) einzelne Bildpunkte unterteilt. Der Nullpunkt befindet sich links unten.
Die Y-Koordinaten nehmen wir in vertikaler und die X-Koordinaten in horizontaler Richtung an. Damit ergibt sich für y ein Wertebereich von 0 bis 255 und für x von 0 bis 319.
Als ersten Parameter für PLOT haben wir eine 1 übergeben. Diese entscheidet zwischen Zeichnen und Löschen. Zum Löschen müssen wir eine 0 übergeben, zum Zeichnen eine Zahl ungleich 0.
Der zweite und dritte Parameter kennzeichnen die X- und die Y-Koordinate des ersten Punktes der Linie. Die Koordinaten des Endpunktes der Linie ergeben sich aus der Summe des zweiten und vierten (x) und aus der Summe des dritten und fünften Parameters (y). Schauen wir uns nun die von PLOT wieder übergebenen Parameter an:

```
 . . . (ENTER) 150  200  1 OK
```

Wir können die Koordinaten des Endpunktes (200,150) und unsere Zahl, die der Entscheidung zwischen Zeichnen und Löschen diente, wiedererkennen. In Zukunft wollen wir solche Zahlen, die Entscheidungszwecken dienen, Flags nennen.
Die von PLOT wieder Übergebenen Parameter können wir für einen weiteren Aufruf von PLOT verwenden. Zwei Linien, die einen spitzen Winkel bilden, erhalten wir auf folgende Art und Weise:

```
 1 100 100 100 10 PLOT -100 10 PLOT DROP DROP DROP (ENTER) OK
```

Der dreimalige Aufruf von DROP dient einfach dazu, die nun überflüssigen Übergabeparameter von PLOT wieder vom Stack zu entfernen. Mit dem nun erworbenen Wissen über das Wort PLOT (und auch das Wort DROP) sind wir jetzt in der Lage, unsere selbst gestellte Aufgabe zu lösen. Wir zerlegen zunächst unsere Aufgabe in sinnvolle Teilaufgaben:

- Dach zeichnen
- Wand zeichnen
- Tür zeichnen
- Fenster zeichnen

Wenn wir unsere Teilaufgaben weiter durchdenken, kommen wir zu folgender Struktur für alle vier Teilaufgaben:

- einige Linien zeichnen
- drei Zahlen vom Stack entfernen

Das Zeichnen besorgt uns das Wort PLOT, das Säubern des Stacks könnte das neu zu definierende Wort 3DR0P erledigen.
Um die Darstellung der Dialoge mit dem Computer übersichtlicher zu gestalten, verzichten wir von nun an auf die Darstellung der ENTER-Taste.

Beginnen wir also mit der Definition des Wortes 3DROP.

```
 : 3DROP  DROP DROP DROP ; OK
```

3DROP kann sofort getestet werden:

```
 1 2 3 4 5 3DROP . . 2 1 OK
```

3DROP funktioniert, so daß wir jetzt zum Zeichnen des Daches übergehen:

```
 : DACH  1 110 95 50 95 PLOT 50 -95 PLOT 3DROP ; OK
```

Bitte probieren Sie DACH gleich aus!
Wenn Sie vorher den Bildschirm löschen wollen, drücken Sie einfach die Tasten SHIFT und HOME (erst SHIFT und dann zusätzlich HOME).
Nun zur Wand und zur Tür:

```
 :  WAND  1 115 50 90 0 PLOT 0 55 PLOT -90 0 PLOT 0 -55 PLOT 3DROP
 ; OK

 : TUER  1 165 50 30 0 PLOT 0 50 PLOT -30 0 PLOT 0 -50 PLOT 3DROP
 ; OK
```

Testen Sie jetzt bitte auch WAND und TUER!
Beim Fenster ist es günstig, wenn die Position auf dem Bildschirm nicht schon innerhalb des Wortes festgelegt wird. Dann können wir FENSTER mehrfach verwenden, indem wir vor dessen Aufruf die Position und unser Flag (Zeichnen/Löschen) noch schnell auf den Stack schaffen.

```
 : FENSTER  30 0 PLOT 0 30 PLOT -30 0 PLOT 0 -30 PLOT 3DROP ; OK
```

Wenn Sie jetzt FENSTER testen, müssen Sie natürlich die noch fehlenden Parameter vorher auf den Stack legen.

```
 1 125 70 FENSTER OK
```

Wir haben nun festgestellt, daß alle Teilprogramme funktionieren. Damit gibt es nun kein Hindernis mehr, diese zum Wort HAUS zu kombinieren.

```
 : HAUS DACH WAND TUER 1 125 70 FENSTER 1 145 110 FENSTER ; OK
```

Und schon ist unser Programm fertig. Doch Halt! Das Löschen des Bildschirms könnte das Wort HAUS eigentlich gleich mit erledigen. Es müßte ja nur vor dem Zeichnen das Steuerzeichen CLS (clear screen) ausführen. CLS wird mit der Zahl 12 codiert.

```
 FORGET HAUS OK
```

Nun ist das HAUS wieder aus dem Vokabular gestrichen. Beachten Sie bitte: Mit FORGET wird nicht nur das gewählte Wort gestrichen, sondern auch alle Worte, die nach dem zu "vergessenden" Wort definiert wurden. In unserem Fall hatten wir Glück, weil HAUS das zuletzt definierte Wort war. Läßt sich ein fehlerhaftes Wort nicht mit FORGET löschen, so führen Sie vorher die Anweisung SMUDGE aus. Mehr über SMUDGE erfahren Sie im Punkt 6.2.1.

Hier also die verbesserte Definition von HAUS:

```
: HAUS  12 EMIT DACH WAND TUER 1 125 70 FENSTER 1 145 110 FENSTER
; OK
```

Damit ist unser Exkurs in die FORTH-Grundlagen beendet. Es empfiehlt sich, unser Beispiel noch einmal nachzuvollziehen und es dabei ein wenig zu verändern. Sie könnten z.B. noch eine Garage "anbauen".
In den nächsten Kapiteln werden wir uns jeweils einem speziellen Problemkreis zuwenden.

3. Zahlen, Variablen und Konstanten

3.1. Wir rechnen mit dem Stack

"Computer" heißt ja eigentlich Rechner, wir aber haben die Rechenfähigkeiten unseres FORTH-Computers noch nicht direkt genutzt. In diesem Kapitel werden wir uns ausführlich mit diesen Fähigkeiten beschäftigen, die wir auch unter dem Begriff ARITHMETIK zusammenfassen können.
Die FORTH-Arithmetik ist auf die Verarbeitung ganzer Zahlen unter Verwendung der vier Grundrechenarten beschränkt. Das ist natürlich auf den ersten Blick eine erhebliche Einschränkung, wenn man die komfortablen Rechenmöglichkeiten im Auge hat, die solche Programmiersprachen wie FORTRAN oder BASIC bieten.
In diesem Kapitel werden wir aber sehen, daß die FORTH-Arithmetik für die meisten Aufgaben ausreichend ist, wenn die entsprechenden Programme wohlüberlegt entworfen werden.
Der große Vorteil der ganzzahligen FORTH-Arithmetik ist die relativ hohe Rechengeschwindigkeit, die etwa 3 bis 10 mal höher liegt, als die der entsprechenden Gleitkomma-Arithmetik, wie sie in anderen Programmiersprachen zu finden ist.
Um zwei Zahlen miteinander arithmetisch zu verknüpfen, benötigt ein Computer vier Informationen:

1. Wo ist Operand Nr.1 zu finden?

2. Wo ist Operand Nr.2 zu finden?

3. Welche Rechenoperation ist auszuführen?

4. Wohin soll das Ergebnis geschafft werden?

Bei FORTH gibt es eine klare Festlegung zu den Fragen 1, 2 und 4:

- Die beiden Operanden werden auf dem Stack erwartet.
- Das Ergebnis wird auch wieder auf den Stack geschafft.

Damit muß FORTH nur noch "wissen", welche Rechenoperation auszuführen ist.
Das geschieht einfach dadurch, daß ein Wort, das die entsprechende Rechenoperation ausführt, aufgerufen wird. Es wird die Operanden als Parameter vom Stack übernehmen, die Rechenoperation ausführen und zum Schluß das Ergebnis wieder auf den Stack schaffen. Es ist sehr naheliegend, daß die entsprechenden mathematischen Symbole gleich als Wortnamen festgelegt worden sind. Hier also die "Standard"-Operatoren für die vier Grundrechenarten. (Später kommen wir dann zu den "Luxus"-Operatoren.)

Operator	Stack		
	vorher	nachher	
+	n1 n2	n1 + n2	Summe
-	n1 n2	n1 - n2	Differenz
*	n1 n2	n1 * n2	Produkt
/	n1 n2	n1 / n2	Quotient

Hinterlegen wir zur Demonstration zwei Zahlen (einfach durch Eingabe und Betätigung der ENTER-Taste) auf den Stack:

```
 77 23 OK
```

Nun addieren wir beide:

```
 + OK
```

Der Computer hat "im Stillen" addiert, und das Ergebnis steht auf dem Stack. Mit Hilfe des Ausgabewortes . erhalten wir folgende Ergebnisanzeige:

```
 . 100 OK
```

Schreiben wir alles in eine Zeile, ergibt sich folgendes Bild:

```
 77 23 + . 100 OK
```

Das Operationszeichen befindet sich nicht zwischen den Operanden, sondern dahinter. Diese Schreibweise heißt Umgekehrte Polnische Notation (UPN) und geht auf die Arbeiten des polnischen Mathematikers Lukasiewicz zurück.
Die UPN hat einen Nachteil: Wir müssen uns von der gewohnten Schreibweise, die auch Algebraische Notation heißt, trennen. Der Vorteil besteht darin, daß sich der Computer nicht mehr um Klammern und Klammerregeln kümmern muß. Er muß auch nicht wissen, daß Punktrechnung vor Strichrechnung geht, die entsprechenden "Gedanken" kann er sich sparen. Das bringt einen Gewinn an Rechengeschwindigkeit.
Wir steuern die Verarbeitung der arithmetischen Ausdrücke durch die Reihenfolge, in der wir die Operanden und Operatoren schreiben. Dazu folgendes Beispiel:
Wir wollen (4 + 5) * 9 rechnen.
In "FORTH" geschrieben, sieht das so aus:

```
 4 5 + 9 * . 81 OK
```

Nun wollen wir 4 + 5 * 9 rechnen:

```
 4 5 9 * + . 49 OK
```

Im ersten Fall wurde zuerst die Addition 4+5 und dann die Multiplikation 9*9 ausgeführt. Im zweiten Fall wurde zuerst die Multiplikation 5*9 ausgeführt und dann die Addition 4+45.
Wenn man neben die Aufgabe die aktuelle Stackbelegung schreibt, wird das noch deutlicher:

1. Aufgabe	Stack	2. Aufgabe	Stack
4	4	4	4
5	4 5	5	4 5
+	9	9	4 5 9
-	9 9	*	4 45
*	81	+	49
.	leer	.	leer

Bei der Darstellung der Stackbelegungen einigen wir uns darauf, daß die auf dem Stack zuoberst stehende Zahl in unserer Darstellung rechts steht. Den Ort, an dem diese Zahl steht, wollen wir in Zukunft TOS (top of Stack) nennen.

Unsere Beispiele zeigen, daß Übergabeparameter eines Wortes später Übernahmeparameter eines anderen Wortes werden, ohne dabei vom "Fleck" (vom Stack) bewegt worden zu sein. Daraus folgt eine gute Eignung der FORTH-Arithmetik für Kettenrechnungen. Weiterhin erlaubt die freie Nutzung des Stack weitgehenden Verzicht auf Variablen zur Speicherung von Zwischenergebnissen.

3.2. Manipulationen auf dem Stack

Einen Nachteil hat die Verwendung des Stack zur Zahlenspeicherung. Wir können unmittelbar immer nur die Zahlen verarbeiten, die sich im Stack an oberster Stelle befinden. Um hier Abhilfe zu schaffen, gibt es eine Reihe von Worten, die etwas Bewegung in den Stack bringen:

Wort	Wirkung	
DUP	(n --> n n)	dupliziert die Zahl im TOS
-DUP	(n --> n ?)	dupliziert die Zahl im TOS nur, wenn sie ungleich 0 ist
DROP	(n1 n2 --> n1)	entfernt eine Zahl vom Stack
SWAP	(n1 n2 --> n2 n1)	vertauscht die beiden obersten Zahlen im Stack
ROT	(n1 n2 n3 --> n2 n3 n1)	rotiert drei Zahlen. Die bisher unterste der drei Zahlen liegt dann auf dem TOS
2DUP	(n1 n2 --> n1 n2 n1 n2)	dupliziert die beiden obersten Zahlen

An dieser Stelle müssen Sie in ein Geheimnis eingeweiht werden. FORTH benutzt noch einen zweiten Stack, der jedoch hauptsächlich zur internen Verwendung durch das FORTH-System vorgesehen ist. Innerhalb von Wortdefinitionen können Sie diesen zweiten Stack ebenfalls verwenden, aber nur mit größter Vorsicht! Er muß am Ende der Definition wieder genauso aussehen, wie vorher. Da der zweite Stack vom FORTH-System zur Speicherung von Rückkehradressen benutzt wird, heißt er RETURN-Stack. Zur Unterscheidung nennen wir den bisher kennengelernten Stack DATEN-Stack. Hier nun die Manipulationswörter für den RETURN-Stack, die nur innerhalb von Wortdefinitionen angewendet werden dürfen.

>R	(n -->)	bringt eine Zahl aus dem TOS zum Return-Stack
R>	(--> n)	holt eine Zahl vom Return-Stack zum TOS
R	(--> n)	kopiert eine Zahl vom Return-Stack zum TOS. Die Zahl bleibt aber auf dem Return-Stack unverändert, im Gegensatz zu R>.

Als "Faustregel" für die Manipulationen mit dem RETURN-Stack gilt, daß innerhalb einer Wortdefinition für jedes >R auch ein R> vorhanden sein muß. Das gleiche gilt auch innerhalb einer DO...LOOP-Konstruktion.

3.3. Variablen und Konstanten

Der Stack ist zwar eine feine Sache zur Speicherung von Zahlen, zuweilen wird aber auch ein FORTH-Programmierer den Wunsch verspüren, Zahlen an einem Ort aufzubewahren, der sicherer ist. Manchmal kann es auch zweckmäßig sein, einem FORTH-Wort einige Übernahmeparameter nicht auf dem Stack bereitzustellen. Wir haben deshalb die einfache Möglichkeit, Zahlen an jeder beliebigen Stelle des Speichers abzulegen. Wir müssen dabei nur beachten, daß wir keine Speicherstellen benutzen, die FORTH benötigt. (FORTH befindet sich ja ebenfalls im Speicher.)
Nehmen Sie bitte als gegeben hin, daß die Speicherstelle mit der Nummer 0 zu beliebigen Zwecken benutzt werden kann. In Zukunft wollen wir die Nummer einer Speicherstelle ADRESSE nennen. In Ihrem Computer gibt es maximal 65536 solcher Adressen. Speichern wir also eine Zahl auf der Adresse 0:

```
 33 0 ! OK
```

Das Wort Speichern (um nicht immer "Ausrufezeichen" sagen zu müssen, nennen wir es "store") verlangt zwei Parameter:

1. die zu speichernde Zahl
2. die Speicheradresse

Und so können wir die auf der Adresse 0 gespeicherte Zahl ganz einfach wieder zum Stack befördern:

```
 0 @ . 33 OK
```

Das Wort @ (fetch) ersetzt die Adresse, die sich im TOS befindet durch deren Inhalt.
Es wäre von einem FORTH-Programmierer reichlich viel verlangt, wenn er sich immer selbst um freie Speicherplätze kümmern müßte. Gute Programmierer sind von Natur aus bequem, und dies wird von FORTH selbstverständlich unterstützt. Soll sich doch das FORTH-System um die Verwaltung der Speicherplätze kümmern! Wir müssen ihm dazu nur mitteilen, welchen Namen wir der zur Verfügung gestellten Speicheradresse geben wollen und welcher Inhalt zunächst gespeichert werden soll. Den Rest erledigt das Definitionswort VARIABLE.

```
 33 VARIABLE ZAHL OK
```

Damit haben wir eine Speicheradresse mit dem Namen ZAHL definiert. Auch ZAHL ist ein vollwertiges FORTH-Wort, wie Sie mit VLIST leicht feststellen können. Was macht nun ZAHL, wenn wir es wieder aufrufen? Es bringt uns seine Adresse zum Stack. Deshalb können wir nun auch wieder mit den Worten ! und @ arbeiten, wie im oberen Beispiel:

```
 ZAHL @ . 33 OK
 77 ZAHL ! OK
 ZAHL @ . 77 OK
```

Es gibt auch ein Wort, mit dem wir den Inhalt einer Variablen um einen bstimmten Wert erhöhen können. Dadurch müssen wir nicht mehr schreiben

```
 ZAHL @ 3 + ZAHL ! OK
```

sondern können dies mit +! Vereinfachen:

```
 3 ZAHL +! OK
```

Die Kombination @ . ist zu einem weiteren Wort zusammengefaßt. Sie können nämlich auch schreiben:

```
 ZAHL ? 80 OK
```

Das Fragezeichen ist doch sehr einleuchtend: Es beantwortet die Frage nach dem Inhalt der Adresse.

Eine weitere Art, in FORTH Zahlen zu speichern, ist die Verwendung von Konstanten. Im Gegensatz zu FORTH-Variablen bringen FORTH-Konstanten immer ihren Wert zum Stack, wenn sie aufgerufen werden. Konstanten sind sinnvoll, wenn ein und dieselbe Zahl mehrfach verwendet werden soll. Sie werden formal in der gleichen Weise definiert wie Variablen.

```
 10 CONSTANT TARIF OK
 TARIF . 10 OK
```

Auch Verkehrspolizisten können sich mit einem FORTH-Computer die Arbeit erleichtern.

```
 : STEMPEL  TARIF * . ." MARK " CR ;  OK
```

Probieren sie selbst!

```
 0 STEMPEL 0 MARK
 OK
 1 STEMPEL 10 MARK
 OK
 2 STEMPEL 20 MARK
 OK
```

Weil wir gerade beim Straßenverkehr sind, behandeln wir noch ein komplizierteres Beispiel. Wir nehmen für einen bekannten Kleinwagen ein bestimmtes Tankvolumen und einen Durchschnittsverbrauch als konstant an. Den Tankinhalt behandeln wir als Variable. Unserem Computer wollen wir mitteilen, wie weit wir gefahren sind. Er soll uns mitteilen, wie weit wir mit dem restlichen Treibstoff noch fahren können und wie viel wir bisher verbraucht haben. Da wir nicht mit Bruchteilen von Litern arbeiten können, nehmen wir einfach Milliliter. Für den Durchschnittsverbrauch ist die Größe Liter pro 100 km ebenfalls untauglich. Wir nehmen statt dessen Milliliter pro 1 km. Jetzt können wir unsere Konstanten definieren:

```
 26000 CONSTANT TANK OK
 85 CONSTANT VERBRAUCH OK
```

Der Tankinhalt soll eine Variable sein:

```
 0 VARIABLE TANKINHALT OK
```

Nach dem Tanken soll der Tank voll sein:

```
 : BETANKT TANK TANKINHALT ! ; OK
```

```
 GETANKT TANKINHALT ? 26000 OK
```

Die Anzahl der gefahrenen Kilometer multiplizieren wir mit Durchschnittsverbrauch, geben dem Produkt ein negatives Vorzeichen und addieren es zum Tankinhalt:

```
 : GEFAHREN  VERBRAUCH * MINUS TANKINHALT +! ; OK
 100 GEFAHREN TANKINHALT ? 17500 OK
```

Die Reichweite ergibt sich aus dem Quotienten Tankinhalt/Durchschnittsverbrauch:

```
 : REICHWEITE  TANKINHALT @ VERBRAUCH / . ." KM       "; OK
 REICHWEITE 205 KM       OK
```

Die verbrauchte Benzinmenge ergibt sich aus der Differenz zwischen dem vollen und dem momentanen Tankinhalt:

```
 : VERBRAUCHT  TANK TANKINHALT @ - . " ML       " ; OK
 VERBRAUCHT 8500 ML       OK
```

Jetzt schauen wir uns noch einmal einen vollständigen Benzin-Dialog an:

```
 GETANKT OK
 200 GEFAHREN OK
 100 GEFAHREN OK
 VERBRAUCHT 25500 ML       OK
 REICHWEITE 5 KM       OK
```

Ein FORTH-Computer läßt sich von irgendwelchen Nebensächlichkeiten, wie zum Beispiel einem leeren Tank nicht beirren:

```
 50 GEFAHREN OK
 VERBRAUCHT 29750 ML OK
 REICHWEITE -44 KM OK
```

3.4. Noch mehr über Zahlen und Zahlensysteme

Bevor wir weitere Arithmetikoperatoren kennenlernen, müssen wir uns noch damit beschäftigen, wie im Computer Zahlen gespeichert werden. Im 2. Kapitel wurde schon erwähnt, daß die Zahlen auf dem Stack in binärer Form vorliegen. Das trifft auch für die Zahlen zu, die in Variablen und Konstanten gespeichert sind.
Binär heißt, daß die Zwei die Zahlenbasis ist. Alle Ziffernstellen unserer Zahlen im Computer sind entweder 0 oder 1 und können damit bequem im Speicher abgelegt werden. Der verfügbare Zahlenbereich hängt nun davon ab, wieviele Binärstellen vorgesehen sind.
Eine Binärstelle, die ja entweder 0 oder 1 ist, nennen wir ein Bit. Eine Speicherzelle unseres Computers ist nun 8 Bit breit, das heißt, daß wir über eine Speicheradresse S Bit parallel erreichen. Diese 8 Bit werden als ein Byte bezeichnet. Mit einem Byte kann man jedoch nur 2 hoch 8, das sind 256, verschiedene Zahlen darstellen. Das ist natürlich viel zu wenig.
Deshalb werden 2 Byte bei der computerinternen Darstellung der Zahlen zu einer (bei FORTH) sogenannten Zelle zusammengefaßt. Damit sind 2 hoch 16 (gleich 65536) verschiedene Zahlen darstellbar. Die Worte ! und @ operieren immer mit Zellen, d.h. mit zwei aufeinanderfolgenden Speicheradressen.

Dabei wird in der Adresse mit der niedrigeren Nummer auch der niederwertige Teil einer 16-Bit-Zahl gespeichert.
Schauen wir uns dies einmal an, indem wir die Worte C@ und C! anwenden. Diese Worte arbeiten ähnlich wie @ und !, jedoch nur mit einem Byte statt mit einer Zelle:

```
 513 VARIABLE X OK
 X C@ . 1 OK
 X 1 + C@ . 2 OK
```

Das höherwertige Byte gibt an, wie oft in unserer Zahl die 256 enthalten ist. Im niederwertigen Byte steht der verbleibende Rest. Rechnen wir alles zusammen, erhalten wir:

```
 2 * 256 + 1 = 513
```

Für manche Anwendungen wird der Zahlenbereich von 0 bis 65535 zu klein sein. Hier bietet FORTH einen Ausweg. Man schließt einfach zwei Zellen zu einer 32-Bit-Zahl zusammen. Damit läßt sich der Bereich von 0 bis 4294967295 (2 hoch 32 verschiedene Zahlen) darstellen. Wenn sie in diesem Zahlenbereich beispielsweise mit Pfennigen rechnen, können Sie Beträge von ca. -21,47 Millionen bis 21,47 Millionen Mark aufsummieren. Wir bezeichnen 32-Bit-Zahlen als doppeltgenaue Zahlen.
Im Speicher des Computers wird der höherwertige Teil einer 32-Bit-Zahl in den beiden Bytes mit der niedrigeren Adresse gespeichert. Nehmen wir beispielsweise die Zahl 66049, die beginnend ab Adresse 10000 gespeichert sein soll:

```
     10003:  2  höherwertiges Byte der niederwertigen Zelle
     10002:  1  niederwertiges Byte der niederwertigen Zelle
     10001:  0  höherwertiges Byte der höherwertigen Zelle
     10000:  1  niederwertiges Byte der höherwertigen Zelle
```

Analog zu den Worten @ und ! gibt es für doppeltgenaue Zahlen die Worte 2@ und 2!.
Und so sieht die Zahl 66049 in der 32-Bit-Darstellung aus. Aus Gründen der Übersichtlichkeit wird die Dualzahl in Vierergruppen und mit den dazugehörigen Speicheradressen entsprechend unserem Beispiel dargestellt:

```
            0000 0000  0000 0001  0000 0010  0000 0001
 Adressen:    10001      10000      10003      10002
```

Wenn Sie prüfen wollen, ob dies wirklich der Zahl 66049 entspricht, addieren sie einfach die Zweierpotenzen, in deren Position eine 1 steht. Beachten Sie, daß die erste Position von rechts die Nummer 0 hat.

```
2 hoch  0 =          1             1
2 hoch  1 =          2             0
2 hoch  2 =          4             0
2 hoch  3 =          8             0
2 hoch  4 =         16             0
2 hoch  5 =         32             0
2 hoch  6 =         64             0
2 hoch  7 =        128             0
2 hoch  8 =        256             0
2 hoch  9 =        512             1
2 hoch 10 =       1024             0
2 hoch 11 =       2048             0
2 hoch 12 =       4096             0
2 hoch 13 =       8192             0
2 hoch 14 =      16384             0
2 hoch 15 =      32768             0
2 hoch 16 =      65536             1
2 hoch 17 =     131072             0
2 hoch 18 =     262144             0
2 hoch 19 =     524288             0
2 hoch 20 =    1048576             0
2 hoch 21 =    2097152             0
2 hoch 22 =    4194304             0
2 hoch 23 =    8388608             0
2 hoch 24 =   16777216             0
2 hoch 25 =   33554432             0
2 hoch 26 =   67108864             0
2 hoch 27 =  134217728             0
2 hoch 28 =  268435456             0
2 hoch 29 =  536870912             0
2 hoch 30 = 1073741824             0
2 hoch 31 = 2147483648             0
```

Addieren wir also die Positionen, in denen eine 1 steht:

```
 65536 + 512 + 1 = 66049
```

Die Vierergruppen unserer Zahl können zu je einer Ziffer zusammengefaßt werden. Das sieht zunächst einfacher aus, als es ist. Mit vier Binärstellen lassen ja sich 16 verschiedene Zahlen darstellen, die Ziffern von 0 bis 9 reichen für diesen Zweck nicht aus. Wir benötigen für die Zahlen 10 bis 15 noch 6 zusätzliche Ziffernsymbole. Es ist üblich, hier die Buchstaben von A bis F zu verwenden.

Binärzahl	Ziffer	
0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	8	
1001	9	
1010	A	(10)
1011	B	(11)
1100	C	(12)
1101	D	(13)
1110	E	(14)
1111	F	(15)

Da wir jetzt 16 verschiedene Ziffern haben, nennen wir Zahlen, die damit dargestellt werden, Hexadezimalzahlen. Schreiben wir nun die Vierergruppen der Zahl 66049 als Hexadezimalzahl auf, erhalten wir folgende Darstellung:

```
66049 = 00010201H
```

Um Hexadezimalzahlen auch dann von Dezimalzahlen unterscheiden zu können, wenn sie keine der Ziffern A bis F enthalten, wird einfach ein "H" angehängt.
Hexadezimale Zahlen haben neben den Oktalzahlen (hier ist die Zahlenbasis 8) in der Computertechnik eine große Bedeutung erlangt, weil deren Umwandlung in Binärzahlen besonders einfach ist.
Mit einem FORTH-Computer können Sie ein beliebiges Zahlensystem vereinbaren. Wenn Sie noch nichts vereinbart haben, werden alle Zahlen als Dezimalzahlen behandelt. Das Umschalten zwischen Hexadezimalzahlen und Dezimalzahlen besorgen die beiden Worte HEX und DECIMAL. Das vereinbarte Zahlensystem ist jedoch nur dann von Bedeutung, wenn Zahlen ein- oder ausgegeben werden sollen. Befindet sich eine Zahl einmal im Computer, sei es auf dem Stack oder in einer Variablen, ist das bei der Eingabe verwendete Zahlensystem ohne weitere Bedeutung. Sie dürfen sogar bei der Ein- und Ausgabe verschiedene Zahlensysteme benutzen und können so recht einfach Konvertierungen durchführen, wie das folgende Beispiel zeigt.

```
66049. OK
```

Nun befindet sich die Zahl 66049 auf dem Stack. Wenn die Ziffernzeichenkette wie in unserem Beispiel einen Punkt enthält, nimmt FORTH diese Zahl als 32-Bit-Zahl an. Der Punkt kann dabei auch an anderer beliebiger Stelle innerhalb und auch vor der Zahl stehen.

```
HEX OK
```

Jetzt wurde die Arbeit mit dem hexadezimalen Zahlensystem vereinbart.

```
D. 10201 OK
```

D. ist der Ausgabeoperator für doppelt genaue Zahlen. Vereinbaren wir wieder das Dezimalsystem

```
DECIMAL OK
```

Nun sollte es uns nicht mehr schwerfallen, einen Ausgabeoperator .HEX zu definieren, der Zahlen vom Stack hexadezimal ausgibt und dann wieder ins Dezimalsystem zurückkehrt.

```
: .HEX  HEX . DECIMAL ; OK
```

Damit können wir in einfacher Weise die Zahl 10000 hexadezimal konvertieren:

```
10000 .HEX 2710 OK
```

Beachten Sie bitte, daß der Computer Hexadezimalzahlen nicht besonders kennzeichnet. Sie müssen stets darüber im Bilde sein, welches Ihr aktuelles Zahlensystem ist.

Bis jetzt wissen Sie aber nur die halbe Wahrheit über Zahlensysteme. FORTH unterstützt nämlich nicht nur die Arbeit im Dezimal- und Hexadezimalsystem, sondern die Arbeit mit (in vernünftigen Grenzen) jeder Zahlenbasis. Zu diesem Zweck enthält FORTH eine Systemvariable, in die die gewünschte Zahlenbasis einfach eingetragen werden kann. Logischerweise heißt diese Variable BASE.

```
66049. 2 BASE ! CR D. DECIMAL
10000001000000001 OK
```

Anschließend definieren wir uns noch ein Wort, das Zahlen vom Stack binär ausgibt, ohne die aktuell vereinbarte Zahlenbasis zu verändern.

```
: .BINAER     ( ist der Name des Wortes                )
  BASE @      ( die aktuelle Zahlenbasis auf den Stack )
  SWAP        ( vertauschen mit der auszugebenden Zahl )
  2 BASE !    ( BASE auf zwei setzen                   )
  .           ( binäre Ausgabe der gewünschten Zahl    )
  BASE !      ( alte Zahlenbasis wieder einstellen     )
  ; OK

10000  .BINAER  10011100010000  OK
256 .BINAER 100000000 OK
HEX OK
F .BINAER 1111 OK
100 .BINAER 100000000 OK
```

Nun noch etwas zum Nachdenken:

```
DECIMAL  BASE ? 10 OK
HEX      BASE ? 10 OK
2 BASE ! BASE ? 10 OK
DECIMAL OK
8 BASE ! BASE ? 10 OK
```

Definieren Sie nun ein Wort, welches die aktuelle Zahlenbasis dezimal ausgibt, ohne sie zu verändern.

Wie werden negative Zahlen im Computer dargestellt?

Um dieses Problem zu lösen, wird der verfügbare Zahlenbereich (sowohl bei einfach als auch bei doppelt genauen Zahlen) in zwei Hälften geteilt. In der unteren Hälfte befinden sich die positiven, in der oberen Hälfte die negativen Zahlen. Das klingt zwar unlogisch, hat aber den Vorteil, daß sich bei der Darstellung positiver Zahlen nichts ändert, und auch die Null wird durch lauter binäre Nullen dargestellt. Das höchstwertige Bit, Bit 15 oder Bit 31 "arbeitet" als Vorzeichen. 0 entspricht einer positiven, 1 einer negativen Zahl.
Untersuchen wir das einmal genauer.
Dazu bietet sich der Ausgabeoperator U. an. U. gibt eine Zahl vom Stack ohne Berücksichtigung des Vorzeichens aus, während der einfache Punkt auch negative Zahlen berücksichtigt.
Zählen wir einmal von zwei beginnend rückwärts und schauen uns die Zahl einmal mit . und einmal mit U. an:

```
  2 DUP . U. 2 2 OK
  1 DUP . U. 1 1 OK
  0 DUP . U. 0 0 OK
 -1 DUP . U. -1 65535 OK
 -2 DUP . U. -2 65534 OK
```

Und noch einmal hexadezimal

```
 HEX OK
  2 DUP . U. 2 2 OK
  1 DUP . U. 1 1 OK
  0 DUP . U. 0 0 OK
 -1 DUP . U. -1 FFFF OK
 -2 DUP . U. -2 FFFE OK
```

Unser Computer zählt also so ähnlich wie der Kilometerzähler in einem Auto. Stellen Sie sich vor, der Zähler steht auf 0, und das Auto wird einen Kilometer rückwärts gefahren. Der Zähler zeigt nun 99999 an, die größte darstellbare Zahl innerhalb des Bereichs von 0 bis 99999. Wir nennen diese computerinterne Darstellung vorzeichenbehafteter Zahlen Zweierkomplement-Darstellung. Das Vorzeichen einer Zahl wird umgekehrt, indem alle Bits einer Zahl negiert (d.h. ins Gegenteil verkehrt) werden und dann noch 1 addiert wird.
Zu einer sehr anschaulichen Zahlenbereichsdarstellung kommt man, wenn man sich den Zahlenbereich von 0 bis 65535 wie ein Band vorstellt. Anfang und Ende des Bandes werden zusammengeklebt. Betrachtet man nun die Klebestelle, so schaut man auf den Bereich um die Null. Unterhalb der Null sind die negativen und oberhalb die positiven Zahlen. Zählt man vorwärts, entspricht das einer Aufwärtsbewegung auf dem vorderen Teil des Bandes. Das Rückwärtszählen entspricht einer Abwärtsbewegung.
Der Null gegenüber befindet sich ein Zahlensprung, und zwar von 32767 auf -32768. Sie können das selbst überprüfen, indem Sie zu 32767 eine Eins addieren.

```
32767 1+ . -32768 OK
```

Sinngemäß gilt dies auch für doppelt genaue Zahlen.

Schauen wir uns noch einmal die Zahlen auf dem Band an, wobei Sie sich das obere und das untere Ende zusammengeklebt vorstellen müssen:

Darstellung ohne Vorzeichen	mit	binär
65535	-1	1111 1111 1111 1111
65534	-2	1111 1111 1111 1110
65533	-3	1111 1111 1111 1101
.	.	.
.	.	.
.	.	.
32770	-32766	1000 0000 0000 0010
32769	-32767	1000 0000 0000 0001
32768	-37768	1000 0000 0000 0000
32767	32767	0111 1111 1111 1111
32766	32766	0111 1111 1111 1110
.	.	.
.	.	.
.	.	.
2	2	0000 0000 0000 0010
1	1	0000 0000 0000 0001
0	0	0000 0000 0000 0000
65535	-1	1111 1111 1111 1111
65534	-2	1111 1111 1111 1110
.	.	.
.	.	.
.	.	.

3.5. Weitere Arithmetikoperatoren

Da Sie nun über die interne Zahlendarstellung im FORTH-System ausreichend informiert sind, werden wir die restlichen Arithmetikoperatoren behandeln. Nehmen Sie dies aber bitte nicht zu wörtlich, FORTH ist ja eine beliebig erweiterbare Sprache, und wenn dieser Rest nun nicht all ihren Ansprüchen genügt, können Sie selbstverständlich Ihre eigenen Operatoren hinzudefinieren.

Achtung ! In den weiteren Ausführungen entfällt die Unterstreichung der Ausgaben!

3.5.1. Vorzeichenbehandlung, Minimum und Maximum

Die Vorzeichenbehandlung umfaßt das Bilden von Absolutwerten und den Vorzeichenwechsel. Da dies sowohl bei 16- und 32-Bit-Zahlen möglich ist, gibt es hierfür vier FORTH-Worte.

```
ABS           bildet den Absolutwert einer 16-Bit-Zahl
( n --> u )   u steht hier für "unsigned", also vorzeichenlos

DABS          bildet den Absolutwert einer 32-Bit-Zahl
( d --> ud )

MINUS         wechselt das Vorzeichen einer 16-Bit-Zahl
( n --> -n )

DMINUS        wechselt das Vorzeichen einer 32-Bit-Zahl
( d --> -d )
```

Die Wirkung dieser vier Worte ließe sich sehr schön mit dem Wort .BINAER aus dem vorigen Kapitel erkennen, hätten wir dort nicht zur Ausgabe den Punkt verwendet, sondern das Wort U., welches eine 16-Bit-Zahl ohne Berücksichtigung des Vorzeichens ausgibt. Deshalb definieren wir nun das Wort U.BINAER.

```
: U.BINAER  BASE @ 2 BASE ! SWAP U. BASE ! ; OK
```

U. hat allerdings noch eine hier ausnahmsweise einmal störende Eigenschaft, nämlich die Unterdrückung führender Nullen. Falls U.BINAER also nicht 16 Ziffern ausgibt, müssen wir uns die Zahl nach links mit Nullen aufgefüllt denken.

```
 1 U.BINAER 1 OK
-1 U.BINAER 1111111111111111 OK
-1 ABS U.BINAER 1 OK
1 MINUS U.BINAER 1111111111111111 OK
16 U.BINAER 10000 OK
16 MINUS U.BINAER 1111111111110000 OK
```

Mit U.BINAER haben wir versucht, uns die "echte" 16-Bit-Darstellung der Zahlen im Speicher "anzuschauen". Vollziehen Sie nun bitte unser Beispiel noch einmal nach, allerdings unter Verwendung von .BINAER, . und U.! Sie erkennen sicher die Vielfalt der Ausgabemöglichkeiten. Es ist ganz allein Ihre Sache, wie die Zahlen auf dem Stack aufgefaßt und ausgegeben werden.
Um nun auch die Wirkung von DMINUS und DABS auf 32-Bit-Zahlen in gleicher Weise zu erkennen, müssen wir uns ein Wort UD.BINAER definieren. Dies ist jedoch nicht ganz so einfach wie bei U.BINAER, weil es (noch) kein Wort gibt, das doppelt genaue Zahlen vorzeichenlos ausgibt.

Die nun folgende Definition von UD.BINAER müssen Sie jetzt noch nicht verstehen, tippen Sie sie einfach ein und probieren Sie sie dann aus:

```
: UD.BINAER CR BASE @ 2 BASE ! ROT ROT <# 32 0 DO # LOOP #>
TYPE CR BASE ! ; OK
```

Nun können wir mit doppeltgenauen Zahlen experimentieren:

```
-1. DABS UD.BINAER
00000000000000000000000000000001
OK
-1. UD.BINAER
11111111111111111111111111111111
OK
1. DMINUS UD.BINAER
11111111111111111111111111111111
OK
256. UD.BINAER
00000000000000000000000100000000
OK
256. DMINUS UD.BINAER
11111111111111111111111100000000
OK
```

Die beiden folgenden Worte dienen der Feststellung von Minimum oder Maximum zweier 16-Bit-Zahlen.

MIN (n1 n2 --> n3)	hinterläßt n3 als die kleinere der beiden Zahlen n1 und n2
MAX (n1 n2 --> n3)	hinterläßt n3 als die größere der beiden Zahlen n1 und n2

Ein Anwendungsfall für MIN und MAX ist das Verhindern von Bereichsüberschreitungen. Bei der Bildschirmgrafik muß zum Beispiel gesichert sein, daß die Koordinaten immer im zulässigen Bereich von 0 bis 319 bzw. 255 (für x bzw. y) liegen. Folgendes Wort kann ein Koordinatenpaar so behandeln, daß bei einer Bereichsüberschreitung der Maximal- oder Minimalwert den ursprünglichen Wert ersetzt:

```
: <XY> 0 MAX 255 MIN SWAP 0 MAX 319 MIN SWAP ; OK
50 60 <XY> . . 60 50 OK
50 360 <XY> . . 255 50 OK
50 -10 <XY> . . 0 50 OK
-10 20 <XY> . . 20 0 OK
```

3.5.2. Addition und Subtraktion

Drei weitere Worte erweitern bzw. vereinfachen die Rechenmöglichkeiten bei der Addition.

D+ (d1 d2 --> d3)	addiert zwei doppeltgenaue Zahlen zu einem doppeltgenauen Ergebnis
1+ (n --> n+1)	addiert 1 zur Zahl im TOS

```
 2+                addiert 2 zur Zahl im TOS
 ( n --> n+2 )
```

Sollten Sie einmal doppeltgenaue Zahlen subtrahieren wollen, können Sie sich ein Wort D- leicht selbst schreiben:

```
 : D- DMINUS D+ ; OK
```

3.5.3. Multiplikation und Division

Die nun folgenden FORTH-Worte steigern die Leistungsfähigkeit der FORTH-Arithmetik beträchtlich. Das bezieht sich besonders auf die Operationen, bei denen als Operanden und als Ergebnis einfach- und doppeltgenaue Zahlen gemischt auftreten. Hinzu kommen noch die beiden Worte U* und U/, die in der vorliegenden FORTH-Version eigentlich die einzigen Worte sind, die wirklich multiplizieren und dividieren können. Von allen anderen multiplizierenden und dividierenden Worten werden U* und U/ letztendlich aufgerufen.

```
 U*                multipliziert zwei einfach genaue vorzeichen-
 ( u u --> du )    lose Zahlen zu einer vorzeichenlosen doppelt-
                   genauen Zahl

 U/                dividiert eine vorzeichenlose doppeltgenaue
                   Zahl durch eine vorzeichenlose einfachgenaue
                   Zahl zu einer einfachgenauen Ergebnis u2 und
                   dem Teilerrest u1
 ( ud u --> u1 u2 )

 MOD               führt eine Division aus und übergibt nur den
 ( n1 n2 --> n3 )  Teilerrest n3 (Modulo-Division)

 /MOD              Division mit Rest, n4 ist der Quotient und n3
                   der Rest
 ( n1 n2 --> n3 n4 )

 */MOD             zunächst Multiplikation von n1 und n2 zu einem
                   doppeltgenauen Zwischenergebnis und
                   anschließend dessen Division durch n3 zu einem
                   einfachgenauen Quotienten n5 und dem Teilerrest
                   n4
 ( n1 n2 n3 --> n4 n5 )

 */                wie */MOD, nur ohne Übergabe des Teilerrestes
 ( n1 n2 n3 --> n5 )

 M*                Multiplikation zweier einfachgenauer Zahlen zu
 ( n1 n2 --> d )   einer doppeltgenauen Zahl, alles mit Vorzeichen

 M/                Division einer doppeltgenauen Zahl durch eine
                   einfachgenaue Zahl zu einem einfachgenauen
                   Quotienten n2 und dem Teilerrest n1, mit
                   Vorzeichen
 ( d n --> n1 n2 )
```

```
 M/MOD              vorzeichenlose Rechenoperation:
                    Division der doppeltgenauen Zahl ud1 durch die
                    einfachgenaue Zahl u2 zum doppeltgenauen
                    Quotienten ud4 und dem einfach genauen
                    Teilerrest u3
 ( ud1 u2 --> u3 ud4 )
```

Um den Sinn der gemischt genauen Operatoren zu erkennen, schauen wir uns folgendes Beispiel an:
Wir möchten ein Wort definieren, das uns eine Zahl auf dem Stack mit einem bestimmten Prozentsatz multipliziert und dann das Ergebnis ausgibt, also etwa folgendermaßen:

```
 250 80 .% 200
 OK
```

.% soll nun auf dem Stack zwei Zahlen vorfinden, beide multiplizieren und dann das Zwischenergebnis durch 100 teilen und ausgeben. Die Definition ist recht einfach.

```
 : .% * 100 / . ; OK
```

Testen wir nun .% :

```
  250 80 .% 200 OK
  400 80 .% 320 OK
  500 80 .% -255 OK
  500 10 .% 50 OK
 1000 50 .% -155 OK
```

Die Ergebnisse unseres Tests können uns nur teilweise zufriedenstellen. Das "OK" unseres Rechners hat also nicht sehr viel mit der Korrektheit der Rechenergebnisse zu tun. Es liegt hier kein Syntaxfehler, also z. B. ein falsches Wort oder ähnliches, vor. Irgendwo muß ein logischer Fehler in dem von uns definierten Wort stecken. Um der Fehlerursache auf die Spur zu kommen, schauen wir uns einmal die Zwischenergebnisse nach der Multiplikation an:

```
  250 * 80 = 20000
  400 * 80 = 32000
  500 * 80 = 40000
  500 * 10 =  5000
 1000 * 50 = 50000
```

Unser Rechner "verrechnet" sich regelmäßig dann, wenn das Zwischenergebnis größer als 32767 wird, weil hier unser Zahlenbereich zu Ende ist. Ein Ausweg ist die Erzeugung eines doppelt genauen Zwischenergebnisses. Damit ist das Problem der Zahlenbereichsüberschreitung gelöst, denn bei der Multiplikation zweier 16-Bit-Zahlen kann kein Ergebnis entstehen, das sich mit 32 Bit nicht darstellen läßt. Definieren wir also .% unter Verwendung der gemischt genauen Operatoren M* und M/ neu:

```
 FORGET .% OK
 : .% M* 100 M/ SWAP DROP . ; OK
```

Die Sequenz SWAP DROP ist notwendig, um den Divisionsrest vom Stack zu entfernen. Nun testen wir unser neues .%:

```
 500 80 .% 400 OK
 30000 103 .% 30900 OK
```

Jetzt funktioniert es also richtig. Eine Bereichsüberschreitung im Ergebnis müssen wir aber trotzdem verhindern.
Unsere Definition können wir unter Verwendung von */ noch erheblich vereinfachen. Dieses Wort scheint speziell für unser Problem geschaffen:

```
 FORGET .% OK
 : .% 100 */ . ; OK
```

Einfacher geht's doch wirklich nicht mehr! Testen wir also wieder:

```
 500 200 .% 1000 OK
 273 1 .% 2 OK
```

Das Wort rechnet korrekt wie das vorige. Da wir uns aber mit dem Erreichten nicht zufrieden geben wollen, stellen wir fest, daß .% nicht ordentlich runden kann. Unser letztes Ergebnis hätte eigentlich 3 sein müssen. Das kommt daher, weil wir den Divisionsrest nicht beachtet haben. Ist dieser Rest größer als 50, sollte zum Ergebnis noch eine 1 addiert werden. Das Wort */MOD übergibt den Divisionsrest zusätzlich zum Quotienten.

```
 FORGET .% OK
 : .% 100 */MOD SWAP 50 / + . ; OK
 251 1 .% 3 OK
 250 1 .% 3 OK
 249 1 .% 2 OK
 . 46 .? MSG #1
```

Dieser Test stellt uns nun zufrieden. Mit der letzten Zeile wurde noch das "gutmütige" Stack-Verhalten überprüft. Wir wollen ja sicher sein, daß .% nicht unkontrolliert Zahlen auf dem Stack hinterläßt. Schauen wir uns die Funktionsweise anhand des Stackdiagramms an:

Aktivität	Stack
	251 1
100	251 1 100
*/MOD	51 2
SWAP	2 51
50	2 51 50
/	2 1
+	3
.	leer

*/ und */MOD sind hervorragend geeignet, Multiplikationen und Divisionen mit Konstanten durchzuführen, die nicht ganzzahlig sind.

Wenn wir zum Beispiel die Zahl Pi durch den Bruch 355/113 annähern, ist die Berechnung eines Kreisumfanges aus dem Durchmesser schnell erledigt.

```
 : UMFANG  355 113 */ ; OK
 1000 UMFANG . 3141 OK
```

Dieses Verfahren ist natürlich auch auf andere Konstanten anwendbar. Bei Benutzung der entsprechenden optimalen Näherung ergeben sich Abweichungen, die sich gerade noch in der letzten Stelle eines 8-stelligen Taschenrechners bemerkbar machen würden.

3.6. Noch einmal Aus- und Eingabe von Zahlen

Sie haben die computerinterne Darstellung von 16- und 32-Bit-Zahlen kennengelernt. Sie wissen, daß 32-Bit-Zahlen bei der Eingabe durch einen Punkt gekennzeichnet werden und daß die Ausgabe von 16-Bit-Zahlen sowohl unter Berücksichtigung des Vorzeichens als auch vorzeichenlos erfolgen kann.
Wie erkennt nun ein FORTH-Programm eine eingegebene 32-Bit-Zahl? Man könnte zum Beispiel die Position des TOS vor und nach der Zahleneingabe auswerten und beim "Wandern" um 2 Bytes auf eine 16-Bit-Zahl und beim "Wandern" um 4 Bytes auf eine 32-Bit-Zahl schließen. Die Position des TOS bekommen wir auf einfache Weise, nämlich durch das Wort SP@. SP@ legt uns nun selbst wieder eine Zahl auf dem Stack ab und verursacht damit auch eine Verschiebung des TOS. Deshalb muß man wissen, daß die durch SP@ auf dem Stack abgelegte TOS-Position die vor dem Aufruf von SP@ ist. Schauen wir uns nun einmal an, wie sich die TOS-Position bei der Zahleneingabe ändert:

```
 SP@ . 224 OK
```

Unser TOS befindet sich jetzt in der Speicheradresse 224. Dies könnte auch irgendeine andere Adresse sein, abhängig davon, wo sich gerade unser Stack befindet. Nun geben wir eine Zahl ein:

```
 0 OK
```

und schauen, wo sich nun der TOS befindet:

```
 SP@ . 222 OK
```

Hoppla, die Spitze des Stacks wandert ja nach unten! Das soll uns aber nicht weiter stören, ist es doch Sache des Computers, den Stack zu verwalten. Wenn wir die Null wieder ausgeben, müßte der TOS wieder in der alten Position sein:

```
 . 0 OK
 SP@ . 224 OK
```

Geben wir eine 32-Bit-Zahl ein, wandert der TOS um 4 Bytes nach unten.

```
 555555. OK
 SP@ . 220 OK
 D. 555555 OK
 SP@ . 224 OK
```

Es wäre aber doch etwas kompliziert, die TOS-Position auszuwerten, um auf die Art der eingegebenen Zahl schließen zu wollen.
Im Abschnitt 2.5 wurde erläutert, daß das Wort NUMBER die eingegebenen Zeichenketten bei Bedarf in Binärzahlen umwandelt. Aufgabe von NUMBER ist es auch, den Punkt als Kennzeichen von 32-Bit-Zahlen zu erkennen. NUMBER tut sogar noch etwas mehr, es übergibt uns die Position des Punktes innerhalb der Zahl. Wir müssen dazu nur die Systemvariable DPL abfragen. DPL steht für "decimal point left" und gibt an, wieviele Positionen der Punkt vom rechten "Ende" der Zahl entfernt war. Bei 16-Bit-Zahlen, in denen Number keinen Punkt gefunden hat, wird in DPL eine -1 übergeben.

```
 255 OK
 DPL ? -1 OK
 255. OK
 DPL ? 0 OK
 .255 OK
 DPL ? 3 OK
 22.5 OK
 DPL ? 1 OK
```

In der Variablen DPL wird die Position des Punktes nur gehalten, solange keine weitere Zahleneingabe erfolgt. Es empfiehlt sich daher, DPL immer unmittelbar nach der Eingabe von Zahlen auszuwerten.
Die letzte Eigenschaft von FORTH, die Sie in diesem Abschnitt kennenlernen sollen, ist die Fähigkeit zur formatierten Ausgabe von Zahlen. Zuweilen möchte man Zahlen innerhalb von Tabellen dargestellt haben. Dies wird durch eine rechtsbündige Darstellung, wie sie mit den beiden Worten .R und D.R erreichbar ist, recht übersichtlich.

.R (n1 n2 -->)	rechtsbündige Ausgabe einer 16-Bit-Zahl Die Zahl n1 wird dabei rechtsbündig in ein Feld der Länge n2 eingetragen.
D.R (d n -->)	rechtsbündige Ausgabe einer 32-Bit-Zahl Die Zahl d wird rechtsbündig in ein Feld der Länge n eingetragen.

Im nächsten Abschnitt werden wir dann die formatierte Zahlenausgabe in einem einfachen Beispiel anwenden.

3.7. Der Computer hilft beim Kartenspiel

Welcher Doppelkopfspieler kennt nicht die Schwierigkeiten, an einem Doppelkopfabend zu vorgerückter Stunde die Übersicht über den Spielstand zu behalten, um zu einem korrekten Spielergebnis zu kommen. Bei einer Doppelkopfrunde könnte durchaus ein FORTH-Programmierer zugegen sein, der mit Hilfe des Computers und des hier folgenden kleinen Programms die Buchführung übernimmt.
Das Herzstück unseres Programms sind die Spielerkonten, die wir jetzt als Variable definieren.

```
 0 VARIABLE #ELKE OK
 0 VARIABLE #SUSI OK
 0 VARIABLE #HANS OK
 0 VARIABLE #OTTO OK
```

Unser Programm soll die Punktekonten stets ausbalanciert halten. Das heißt, daß die Summe aller Konten immer gleich Null sein soll. Dadurch bleibt der Spielstand übersichtlich. Hier wird das erreicht, indem der Spielwert zunächst von allen vier Konten subtrahiert wird und dann dem Spieler, dessen Name eingegeben wurde in vierfacher Höhe gutgeschrieben wird.
Wir benötigen deshalb zunächst ein Wort, das einen Spielwert von allen Spielerkonten subtrahiert. Es soll den auf dem Stack vorliegenden Spielwert nicht zerstören.

```
 : ALLE- DUP MINUS
         DUP #ELKE +!
         DUP #SUSI +!
         DUP #HANS +!
             #OTTO +! ; OK
```

Nun definieren wir vier Worte mit den Namen der Spieler. Jedes Wort ruft erst ALLE- auf und addiert dann seinem eigenen Konto den vierfachen Spielwert. Dieses Verfahren funktioniert allerdings nur bei Solo-Spielen richtig. Bei normalen Spielen mit zwei Gegenspielern würde den Spielerkonten immer der doppelte Spielwert gutgeschrieben bzw. abgezogen werden.
Das gleichen wir aus, indem vor der Ausgabe des Spielstandes der Inhalt des Punktekontos halbiert wird. Um nun wiederum bei den Solo-Spielen korrekt zu verfahren, verdoppeln wir den Spielwert bedarfsweise durch das Wort SOLO.

```
 : SOLO 2 * ; OK
```

Und jetzt die Definition der oben erwähnten vier Worte:

```
 : ELKE  ALLE- DUP 4 * #ELKE +! ; OK
 : SUSI  ALLE- DUP 4 * #SUSI +! ; OK
 : HANS  ALLE- DUP 4 * #HANS +! ; OK
 : OTTO  ALLE- DUP 4 * #OTTO +! ; OK
```

Auch diese vier Worte zerstören durch das Duplizieren den auf dem Stack liegenden Spielwert nicht.
Abschließend benötigen wir noch ein Wort, das den Spielstand in übersichtlicher Form ausgibt. Wir sollten dabei auch daran denken, den Stack in Ordnung zu bringen. Dort befindet sich ja nach wie vor unser Spielwert. Man könnte dies mit dem Wort DROP erledigen. Das hätte aber den Nachteil, daß wir den Spielstand nur dann ohne Fehlermeldung ausgeben könnten, wenn sich vorher mindestens eine Zahl auf dem Stack befunden hat. Zweckmäßiger ist hier die Anwendung eines kleinen Tricks. Mit dem Wort SP! können wir den Stack ohne großen Aufwand völlig leeren. SP! wird übrigens auch nach Fehlern stets vom FORTH-System aufgerufen, deshalb ist dann der Datenstack auch immer leer.

Die Ausgabe der Zahlenwerte erfolgt rechtsbündig in einem 10 Zeichen breiten Feld mit Hilfe des Wortes .R .

```
 : ST SP! CR 15 SPACES
      ." ELKE  " #ELKE @ 2 / 10 .R
          CR 15 SPACES
      ." SUSI  " #SUSI @ 2 / 10 .R
          CR 15 SPACES
      ." HANS  " #HANS @ 2 / 10 .R
          CR 15 SPACES
      ." OTTO  " #OTTO @ 2 / 10 .R
          CR ; OK
```

Und jetzt kann das Spiel beginnen. Lassen wir uns erstmal den Spielstand zeigen:

```
 ST
                ELKE           0
                SUSI           0
                HANS           0
                OTTO           0
 OK
```

Elke und Susi gewinnen ein Spiel mit dem Spielwert 10:

```
 10 ELKE SUSI ST
                ELKE          10
                SUSI          10
                HANS         -10
                OTTO         -10
 OK
```

Hans gewinnt ein Solo-Spiel mit dem Spielwert 30:

```
 30 SOLO HANS ST
                ELKE         -20
                SUSI         -20
                HANS          80
                OTTO         -40
 OK
```

Lassen wir jetzt den Pechvogel Otto noch ein Solo-Spiel verlieren und verabschieden uns dann von der Doppelkopfrunde.

```
 -60 SOLO OTTO ST
                ELKE          40
                SUSI          40
                HANS         140
                OTTO        -220
 OK
```

4. Der Editor und die externe Textspeicherung

FORTH ist nicht nur einfach als Programmiersprache zu verstehen, sondern als komplettes in sich abgeschlossenes Konzept zur Programmentwicklung. FORTH hat Betriebssystemeigenschaften, und aus dieser Sicht ist es selbstverständlich, daß FORTH-Programme auch durch das FORTH-System selbst auf einem externen Speichermedium abgespeichert werden können.
Zum Speichern von Programmen gibt es zwei grundsätzliche Möglichkeiten:

1) Speichern des Textes, der dem Textinterpreter angeboten werden soll
2) Speichern des Programms, das der Textinterpreter aus den Wortdefinitionen erzeugt hat.

Wir werden uns der ersten Möglichkeit bedienen, da diese im FORTH-System bereits enthalten ist, während man sich bei der zweiten Möglichkeit erst entsprechende (Software-)Werkzeuge schaffen muß. Ihnen wird die Textspeicherung eine große Arbeitserleichterung bringen. Das FORGET einer weiter zurückliegenden Wortdefinition wird Ihnen nicht mehr schwerfallen, da Sie nicht mehr alle nachfolgenden Definitionen neu eintippen müssen.

4.1. Der Textpuffer

Alle Texte, die der Textinterpreter verarbeitet (auch Ihre Tastatureingaben), entnimmt er einem Textpuffer. Es gibt so viele Textpuffer wie die Kapazität des zur Verfügung stehenden Speichermediums es zuläßt.
Die Tastatureingaben stehen dem Textinterpreter nach Betätigung der ENTER-Taste im Textpuffer mit der Nummer 0 zur Verfügung. Dieser Puffer heißt terminal-input-buffer. Er hat eine Länge von 80 Zeichen. Wenn Sie dem Textinterpreter mehr anbieten, werden die überzähligen Zeichen ignoriert. Die Lage im Speicher, d.h. dessen Startadresse können Sie mit der Systemvariablen TIB einstellen. Der terminal-input-buffer nimmt unter den übrigen Textpuffern insofern eine Sonderstellung ein, als er eine von den übrigen Textpuffern abweichende Länge und Lage im Speicher hat. Dem Textinterpreter selbst ist es vollkommen egal, ob er seine Aufgaben nun aus dem Terminal-Input-Buffer oder aus einem der übrigen Puffer bezieht.
Bei diskettenorientierten FORTH-Versionen wird die gesamte Diskette mit Textpuffern belegt. Falls der Textinterpreter auf einen dieser Textpuffer zugreifen will, organisiert das FORTH-System die Abbildung dieses einen Puffers in einen kleinen Speicherbereich des Computers. Bei Bedarf, z.B. nach Änderungen wird dann der Inhalt dieses Speicherbereichs wieder in den zuständigen Textpuffer auf der Diskette zurückübertragen, um Platz für einen neuen Textpuffer zu schaffen.
Diese Arbeitsweise ist mit einem Magnetbandgerät anstelle des Diskettenlaufwerks undenkbar, da sehr häufig solche Vorgänge, ähnlich dem SAVE und LOAD, erforderlich wären. Die Laufwerksmechanik würde schnell verschleißen.
Die hier vorliegende FORTH-Version wurde speziell für die Arbeit mit dem Magnetband als externes Speichermedium optimiert. Alle Textpuffer werden dem FORTH-System ständig im Speicher des Computers zur Verfügung gestellt.

Das hat den Nachteil, daß deren Anzahl recht begrenzt ist, abhängig von der Speicherausstattung des Computers. Bei Bedarf kann ein zusammenhängender Bereich von Textpuffern als Datei auf das Magnetband ausgegeben werden bzw. von dort auch wieder gelesen werden.
Sie können selbst festlegen, welcher Speicherbereich für den obengenannten Zweck verwendet werden soll. Anfangs- und Endadresse dieses Bereichs legen Sie mit den beiden Systemvariablen FIRST und LIMIT fest. Hier liegt aus programmtechnischer Sicht einer der wenigen Unterschiede zwischen KC-FORTH und diskettenorientierten FORTH-Versionen, dort sind FIRST und LIMIT als Konstanten vereinbart. Wenn Sie also die Verwendung von FIRST und LIMIT vermeiden, sind Ihre Programme auf andere FORTH-Systeme übertragbar.

Wo befinden sich nun die Textpuffer? Fragen wir doch den Computer:

```
 HEX OK
 FIRST ? 2FF0 OK
 LIMIT ? 4000 OK
```

Weiterhin kann uns der Computer "erzählen", wie viele Textpuffer es gibt:

```
 #BUFF . 8 OK
```

Sollten sie mit einem Speichererweiterungsmodul z.B. einen Speicherbereich bis z.B. 8000H zur Verfügung haben, können Sie LIMIT auf 8000H stellen.

```
 8000 LIMIT ! OK
```

Die Anzahl der Textpuffer erhöht sich dadurch beträchtlich.

```
 #BUFF DECIMAL . 41 OK
```

Jeder Textpuffer benötigt 514 Bytes im Speicher. Davon werden 512 Bytes für die Textspeicherung verwendet. Im Anschluß daran stehen in zwei Bytes binäre Nullen, die dem Textinterpreter das Ende des Textpuffers anzeigen.

4.2. Der Screen

Eine bestimmte Anzahl von Textpuffern wird zu einem Screen zusammengefaßt. Unter einem Screen versteht man bei FORTH eine Textmenge, die sich bequem auf einem Bildschirm darstellen läßt. Ein Screen enthält 16 Zeilen zu je 32 Zeichen. Da das insgesamt bereits 512 Zeichen sind, enthält also ein Screen genau einen Textpuffer. Üblich sind bei anderen FORTH-Systemen auch zwei, vier oder acht Textpuffer. Auch die Zeilen in den Screens sind meistens 64 Zeichen lang. Um zu erreichen, daß Programme zwischen unterschiedlichen FORTH-Systemen übertragbar sind, gibt es drei Systemkonstanten, die die Verhältnisse von Puffergröße, Screengröße und Zeilenlänge exakt beschreiben.

```
 C/L   . 32 OK  (Zeichen pro Zeile)
 B/BUF . 512 OK (Bytes pro Textpuffer)
 B/SCR . 1 OK   (Textpuffer pro Screen)
```

Die Anzahl der Screens kann ebenfalls durch die Systemvariablen FIRST und LIMIT verändert werden. Es ist zu beachten, daß nicht mit dem nichtexistierenden Screen 0 gearbeitet wird. Ein solches Unterfangen führt zum Systemabsturz.
Der FORTH-Programmierer legt seine Programmtexte in den Screens ab. Anschließend teilt er dem Textinterpreter mit, daß dieser von nun an seine Aufgaben dem entsprechenden Screens zu entnehmen hat. Ist der Textinterpreter am Ende des Screens angelangt, meldet er sich mit seinem vertrauten "OK" wieder zurück (oder auch nicht, wenn er unterwegs einen Fehler gefunden hat).

4.3. Ein Text wird gespeichert

Wie schaffen wir Texte auf einen Screen? Schauen wir uns dazu den Screen 3 an:

```
 3 LIST

 SCR  #  3

  0
  1
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 OK
```

Der Screen 3 ist noch leer, er könnte aber auch ein zünftiges Durcheinander von irgendwelchen Zeichen enthalten. Deshalb wollen wir diesen Screen zunächst mit ordentlichen Leerzeichen füllen, aber wie? Wie finden wir heraus, wo der Screen 3 im Speicher zu finden ist? Natürlich gibt es ein FORTH-Wort hierfür. Es heißt (LINE), und wir übergeben ihm die Zeilen- und die Screen-Nummer und erhalten dafür die Adresse der gesuchten Zeile und deren Länge. Wir benötigen also nun die Adresse der Zeile 0 im Screen 3:

```
 0 3 (LINE) . . 32 13300 OK
```

Mit dem Wort BLANKS füllen wir einen Speicherbereich mit Leerzeichen. Wir übergeben BLANKS die Anfangsadresse und die Länge dieses Bereiches. Bei uns ist die Länge 16 mal 32 Zeichen, also 512 Bytes.

```
 0 3 (LINE) DROP 512 BLANKS OK
```

Der Screen 3 enthält jetzt nur Leerzeichen. Mit 3 LIST können Sie Sich wie oben davon überzeugen. Übrigens, wenn sie während des Screen-Listings eine beliebige Taste drücken, beendet der Computer sofort das Ausschreiben weiterer Zeilen.

Das Wort BLANKS stammt vom Wort FILL ab. Mit FILL können wir einen Speicherbereich mit beliebigen Zeichen füllen. FILL benötigt zusätzlich als dritten Parameter noch den Zeichencode.
Füllen wir doch einfach die Zeile 1 mit Ausrufezeichen. Der Zeichencode hierfür ist 33.

```
 1 3 (LINE) 33 FILL OK
 3 LIST

 SCR  #  3

 0
 1 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 2
 3         ( hier wurde eine Taste gedrückt )
 OK
```

Als nächste Übung wollen wir von der Tastatur aus ein Zeichen eingeben und dieses dann in der Zeile 0 in die Mitte als 16. Zeichen plazieren. Das Wort KEY kennen Sie ja schon aus dem 2. Kapitel.

```
 KEY 0 3 (LINE) DROP 16 + C! OK
 3 LIST

 SCR  #  3

  0               A
 1 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 2
 OK
```

Mit dem Wort KEY ist es allerdings sehr beschwerlich, eine Zeile einzugeben. Günstiger und bequemer kann es das Wort EXPECT. EXPECT erwartet eine Speicheradresse und eine maximale Zeichenzahl. Es nimmt eine Zeichenkette aus dem Terminal-Input-Buffer entgegen und speichert diese dann beginnend ab der übergebenen Adresse bis zur vorgegebenen Maximalzahl. Die Eingabe wird mit der ENTER-Taste abgeschlossen. An das Ende der Zeichenkette wird ein Byte mit einer 0 angehängt. Das Wort EXPECT paßt übrigens sehr gut mit dem Wort (LINE) zusammen, wenn wir genau eine Zeile eingeben wollen.
Für eine bessere Übersichtlichkeit wird in den folgenden Ausführungen die Betätigung der ENTER-Taste wie in den ersten Kapiteln teilweise durch (ENTER) hervorgehoben.

```
 2 3 (LINE) OK
 EXPECT (ENTER) DIESER TEXT IST SINNLOS  (ENTER)

 3 LIST

 SCR  #  3

  0                 A

  1 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  2 DIESER TEXT IST SINNLOS
  3
 OK
```

Die Systemvariable SCR enthält immer die Nummer des gerade bearbeiteten Screens. SCR wird auch von LIST beeinflußt, schauen wir einmal nach:

```
 SCR ? 3 OK
```

Nun können wir uns ein Wort schaffen, mit dem es möglich ist, einen Text in eine Zeile des aktuellen Screens zu schreiben. Wir nennen es einfach P, in Anlehnung an das englische Wort "put".

```
    : P SCR @ (LINE) EXPECT ; OK
```

Sie erkennen sicher schon, daß P als Übergabeparameter lediglich die Zeilennummer benötigt.

```
 3 P  (ENTER)  DIESER TEXT IST AUCH OHNE SINN  OK

 3 LIST

 SCR  #  3

  0                A
  1 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  2 DIESER TEXT IST SINNLOS
  3 DIESER TEXT IST AUCH OHNE SINN
  4
  5
 OK
```

Das Wort CLEAR soll uns den gesamten Screen mit Leerzeichen füllen.

```
 : CLEAR 0 SCR @ (LINE) DROP 512 BLANKS ; OK
 CLEAR OK
 3 LIST

   SCR  #  3

    0
    1
    2
    3
    4
   OK
```

Damit ist der Screen 3 wieder frei für sinnvollere Texte. CLEAR funktioniert allerdings in dieser Form nur bei FORTH-Systemen, bei denen die Länge des Textpuffers gleich der des Screens ist. Das heißt, daß B/SCR gleich 1 sein muß.

Wenn wir die beiden Wortdefinitionen P und CLEAR in einen Screen schreiben, können wir sie uns mit Hilfe des Magnetbandgerätes speichern und für später aufheben. Die Ausgabe mehrerer zusammenhängender Screens auf Magnetband besorgt das Wort CSAVE. Es erwartet auf dem Stack die Nummern des ersten und des letzten auszugebenden Screens. Außerdem müssen wir noch einen Dateinamen angeben, unter dem unser Text auf Band gespeichert werden soll. Das Wort VERIFY ruft, um die Aufzeichnung zu überprüfen, das CAOS-Systemprogramm VERIFY auf. Mit CLOAD können wir unsere Screens wieder vom Magnetband in den Speicher laden. CLOAD erwartet die gleichen Parameter wie CSAVE. CLOAD bricht das Laden ab, wenn der letzte angegebene Screen erreicht ist, auch wenn die Datei länger sein sollte.
Schließlich können wir noch eine Anzahl Screens vom Dateianfang beginnend ausblenden (sie werden dann nicht mitgeladen), wenn wir

deren Anzahl in die Systemvariable OFFSET eintragen.
Die Definitionen von P und CLEAR wollen wir in den Screen 3 eintragen. Dabei verwenden wir natürlich unser Wort P. Beachten Sie, daß sie nach Eingabe von P die ENTER-Taste betätigen müssen. Außerdem müssen Sie mindestens 32 Zeichen eingeben (also am besten die gesamte Zeile mit Leerzeichen auffüllen) damit das Ende-Byte mit der Null als 33.Zeichen in die nächste Zeile gelangt. Sonst würde später der Textinterpreter irritiert, weil er ja die Null als Screenende interpretiert.

```
 3 SCR ! OK
```

Damit ist der Screen 3 aktueller Screen geworden. Mit 3 LIST hätten wir das auch erreicht.

```
 0 P : CLEAR 0 SCR @ (LINE) DROP 512     OK
 1 P   BLANKS ;                          OK
 2 P : P SCR @ (LINE) EXPECT ;           OK
 3 LIST

 SCR  #  3

  0 : CLEAR 0 SCR @ (LINE) DROP 512
  1   BLANKS ;
  2 : P SCR @ (LINE) EXPECT ;
  3
  4
 OK
```

Nun "schaffen" wir den Screen 3 auf das Magnetband. Als Dateinamen wählen wir "PROGRAMM". Tippen Sie dazu die folgende Zeile ein und starten Sie unmittelbar vor dem Betätigen der ENTER-Taste Ihr auf Aufnahme geschaltetes Magnetbandgerät.

```
 3 3 CSAVE PROGRAMM OK
```

Wenn das OK erschienen ist, stoppen Sie das Magnetbandgerät wieder. Sie können jetzt zurückspulen und mit VERIFY Ihre Aufzeichnung überprüfen. Die Bildschirmausschriften und auch die sonstigen Reaktionen von VERIFY entsprechen denen, die in der Bedienungsanleitung des Computers angegeben sind. Es arbeitet ja das gleiche Programm. Wenn Ihre Aufzeichnung in Ordnung ist, wollen wir den Text vom Band wieder in den Speicher laden, allerdings in einen anderen Screen. Spulen Sie dazu das Band wieder zurück.Dem Wort CLOAD übergeben wir wieder die Nummer des ersten und des letzten Screens, die wir mit dem Text vom Band füllen wollen. Die Angabe des letzten Screens ist nur dann von Bedeutung, wenn die Datei mehr Screens belegen würde, als angegeben worden sind.

```
 4 8 CLOAD
```

Sobald Sie die ENTER-Taste betätigt haben, wird der Bildschirm gelöscht, und das FORTH-System beginnt, die Datei einzulesen. Nach dem Einlesen des ersten Blockes wird im Normalfall der Dateiname auf dem Bildschirm erscheinen. Sollten statt dessen nur Sterne ausgegeben werden, heißt dies, daß der erste Block mit dem Dateinamen noch nicht gefunden wurde. Überprüfen Sie in diesem Fall, ob das Band richtig positioniert ist. Anschließend an den Dateinamen werden die Nummern der gelesenen Blöcke ausgeschrieben. Hinter der Blocknummer steht ein Kennzeichen, ob der Block in den

Speicher übernommen wurde. Dabei können 3 Fälle auftreten:

1. > -- Der Block wurde korrekt übernommen.

2. ? -- Der Block ist fehlerhaft und wurde deshalb nicht übernommen.

3. * -- Der Block steht am Dateianfang und wurde wegen OFFSET ungleich Null nicht übernommen oder ein vorangegangener Block ist fehlerhaft.

Wenn beim Einlesen einer Datei ein Fehler auftritt, spulen Sie das Bandgerät vor den fehlerhaften Block zurück und schalten wieder auf Wiedergabe. Das können Sie im Bedarfsfall wiederholen. Nach dem dritten erfolglosen Versuch wird der fehlerhafte Block übernommen, und Sie sollten dann den entsprechenden Screen korrigieren. Wenn das Lesen unserer kleinen Datei ordnungsgemäß verlaufen ist, müßten jetzt beide Wortdefinitionen auch im Screen 4 stehen.

```
 4 LIST

 SCR  #  4

  0 : CLEAR 0 SCR @ (LINE) DROP 512
  1   BLANKS ;
  2 : P SCR @ (LINE) EXPECT ;
  3
  4
 OK
```

4.4. Der Textinterpreter verarbeitet den gespeicherten Text

Wir haben unseren Text nicht um seiner selbst willen gespeichert. Wir wollen ihn jetzt wieder verarbeiten.
Eine Form der Verarbeitung besteht darin, für den Textinterpreter vorübergehend einen Screen anstelle der Tastatur als Eingabequelle zu definieren. Der Textinterpreter betrachtet dann den gesamten Screen als eine riesenlange Eingabezeile und meldet sich schließlich am Screenende mit seinem OK wieder.
Unsere beiden Worte P und CLEAR dürfen wir nun getrost aus dem Wörterbuch streichen, da sie uns der Textinterpreter gleich wieder neu erstellen kann.

```
 FORGET CLEAR OK
```

Das Wort LOAD schaltet die Eingabe auf einen Screen um. Die Nummer des gewünschten Screen wird selbstverständlich auf dem Stack übergeben.

```
 4 LOAD OK
```

Jetzt stehen P und CLEAR wieder im Wörterbuch. Überprüfen Sie es mit VLIST. Es sei an dieser Stelle nochmals betont, daß sich der Textinterpreter genauso verhält, als würde man den Screeninhalt von der Tastatur aus eintippen. Eine Ausnahme bestätigt hierbei allerdings die Regel: Sollte beim Laden eines Screens ein Fehler auftreten, hinterläßt der Textinterpreter die genaue Textposition, bei der der Fehler auftrat. Er legt dazu die Inhalte der beiden

Systemvariablen BLK und IN auf dem ansonsten geleerten Stack ab. In BLK steht die Nummer des Textpuffers (bei uns wegen B/SCR=1 auch gleich die Nummer des Screens) und in IN die genaue Textposition innerhalb dieses Textpuffers. Somit ist beim Laden mehrerer Screens eine schnellere Fehlerlokalisation möglich.
LOAD kann auch verschachtelt angewendet werden, d.h. innerhalb eines Screens kann LOAD wieder aufgerufen werden. Zu diesem Zweck werden BLK und IN auf dem Returnstack zwischengespeichert. Die Verschachtelungstiefe ist lediglich durch die Größe des Returnstacks eingeschränkt.

4.5. Der Editor

Ein Editor ist ein Programm, mit dessen Hilfe man Texte erstellen und manipulieren kann. So gesehen, können unsere beiden Worte P und CLEAR bereits als Vorstufe eines einfachen Editors angesehen werden, wenn auch der Komfort, den sie bilden, äußerst bescheiden ist.
Bei diskettenorientierten FORTH-Systemen verfährt man meistens so, daß auf einigen Screens auf der Diskette der Programmtext für einen einfachen Screen-Editor bereitgehalten wird. So ein Editor kann allerdings erst dann arbeiten, wenn er wie im vorigen Abschnitt beschrieben, mit LOAD in ein ausführbares Computerprogramm übersetzt wurde. Diesen Übersetzungsvorgang nennen wir compilieren. Der Textinterpreter arbeitet dann als Compiler. Da der Editor bei der Programmentwicklung sehr oft gebraucht wird, ist er bei KC-FORTH ständig im Wörterbuch enthalten. Allerdings ist er dort etwas versteckt, er ist nämlich nicht im FORTH-Vokabular enthalten, sondern nur mit diesem verbunden. Die Editor-Worte sind erst dann aufrufbar, wenn das Vokabular EDITOR aktiviert ist. Das können Sie ganz einfach durch Eingabe des Vokabularnamens tun:

```
EDITOR OK
```

Schauen Sie nun mit VLIST nach, welche neuen Worte jetzt zur Verfügung stehen. Der Textinterpreter führt seine Suchläufe so durch, daß er zuerst im gerade aktivierten Vokabular sucht, im sogenannten Context-Vokabular, anschließend in dem Vokabular, innerhalb dessen das Context-Vokabular definiert wurde u.s.w. bis zum Startpunkt des FORTH-Vokabulars. Durch Eingabe von

```
FORTH OK
```

erreichen Sie, daß der Editor bei Suchdurchläufen wieder übergangen wird. Nun ist FORTH wieder Context-Vokabular. Mit VLIST erkennen Sie, daß nur das Wort EDITOR selbst im FORTH-Vokabular enthalten ist. Das muß auch so sein, denn sonst könnten wir es ja vom FORTH-Vokabular aus nicht aufrufen.

4.6. Das Editor-Vokabular im einzelnen

Der Editor benutzt einen speziellen Speicher, in dem üblicherweise eine Zeile gespeichert wird. Dieser Speicher hat einen konstanten Abstand zum Ende vom Wörterbuch, er verschiebt sich also, wenn neue Worte hinzudefiniert werden. Das Wort PAD legt uns die Anfangsadresse dieses Speicherbereichs auf den Stack. In diesem Zusammenhang muß auch das Wort HERE erwähnt werden, das die erste freie Adresse nach dem Ende des Wörterbuchs zur Verfügung stellt. Egal, wo sich das Ende des Wörterbuchs nun konkret befindet, die

Differenz der beiden Adressen von PAS und HERE wird immer gleich sein.

```
 PAD HERE - . 68 OK
```

Doch jetzt wollen wir die Möglichkeiten des Editors etwas näher kennenlernen.
Die Worte TEXT, LINE und -MQVE üben lediglich Hilfsfunktionen aus. Sie sind die Basis der nun folgenden Worte, mit denen Sie etwas "spielen" sollten, um deren Bebrauch zu erlernen.
Für die folgenden Worte müssen wir immer eine Zahl n auf dem Stack zur Verfügung stellen.
Es gilt also für alle diese Worte: (n -->)

H	kopiert die Zeile n aus dem aktuellen Screen in den Zwischenspeicher PAD
E	überschreibt die Zeile n im aktuellen Screen mit Leerzeichen
S	einfügen einer Leerzeile in die Position n, alle folgenden Zeilen rücken nach unten
D	löschen der Zeile n, alle folgenden Zeilen rücken nach oben, die gelöschte Zeile steht danach in PAD zur Verfügung
T	ausschreiben der Zeile n
R	Transport der in PAD liegenden Zeile in die Position n des aktuellen Screens
P	eintragen des nachfolgend eingegebenen Textes in die Zeile n des aktuellen Screens
CLEAR	Screen n wird mit Leerzeichen vollgeschrieben

Außerdem existiert noch das Wort MOVE.

```
 MOVE    ( n1 n2 --> )
       * kopieren des Inhaltes von Screen n1 in den Screen n2
```

Mit Hilfe dieser Worte ist es uns möglich, Manipulationen mit Zeilen und kompletten Screens auszuführen. Die folgenden Beispiele sollen das darstellen.

```
 EDITOR 5 CLEAR OK
 5 LIST

 SCR  #  5

  0
  1
  2
OK
0 P QUAELE NIE EIN TIER ZUM SCHERZ OK
1 P DENN ES FUEHLT WIE DU OK
2 P DEN SCHMERZ ! OK
5 P SPIELE NIE MIT DEM GEWEHR OK
6 P DENN ES KOENNT' GELADEN SEIN ! OK
5 LIST
```

```
 SCR  #  5

  0 QUAELE NIE EIN TIER ZUM SCHERZ
  1 DENN ES FUEHLT WIE DU
  2 DEN SCHMERZ !
  3
  4
  5 SPIELE NIE MIT DEM GEWEHR
  6 DENN ES KOENNT' GELADEN SEIN !
  7
  8
 OK
 0 H 5 S 5 LIST

 SCR  #  5

  0 QUAELE NIE EIN TIER ZUM SCHERZ
  1 DENN ES FUEHLT WIE DU
  2 DEN SCHMERZ !
  3
  4
  5
  6 SPIELE NIE MIT DEM GEWEHR
  7 DENN ES KOENNT' GELADEN SEIN !
  8
 OK
 5 R 5 LIST

 SCR  #  5

  0 QUAELE NIE EIN TIER ZUM SCHERZ
  1 DENN ES FUEHLT WIE DU
  2 DEN SCHMERZ !
  3
  4
  5 QUAELE NIE EIN TIER ZUM SCHERZ
  6 SPIELE NIE MIT DEM GEWEHR
  7 DENN ES KOENNT' GELADEN SEIN !
 OK
 6 D 0 R 5 LIST

 SCR  #  5

  0 SPIELE NIE MIT DEM GEWEHR
  1 DENN ES FUEHLT WIE DU
  2 DEN SCHMERZ !
  3
  4
  5 QUAELE NIE EIN TIER ZUM SCHERZ
  6 DENN ES KOENNT' GELADEN SEIN !
 OK
```

Zur Eingabe eines kompletten Screens und natürlich auch zur Änderung beliebiger Zeichen im Screen eignet sich das Wort EDIT. Wenn Sie es mit dem Parameter n aufrufen, wird der Bildschirm gelöscht, und es erscheint der Screen n. Mit

```
 5 EDIT
```

kann z.B. der Screen 5 korrigiert werden.

Während der Auflistung des Screens darf keine Tastatureingabe erfolgen. Dadurch würde der Screen-Inhalt zerstört. Der Cursor befindet sich in der Zeile 0 am Anfang. Mit den Cursorsteuertasten können Sie auf dem gesamten Screen "herumfahren" und irgendwelche Manipulationen durchführen. Folgende Steuertasten haben eine spezielle Wirkung:

INS Auf der Cursorposition wird ein Leerzeichen eingefügt. Alle folgenden Zeichen der Zeile werden weitergeschoben.

DEL Das Zeichen auf der Cursorposition wird gelöscht, alle folgenden Zeichen rutschen nach.

SHIFT-DEL Die gesamte Zeile wird gelöscht.

CLR Das Zeichen auf der Cursorposition wird ein Leerzeichen, der Cursor rückt eine Position nach links.

HOME Der Cursor springt nach links oben.

SHIFT-HOME Der Inhalt des Screens vor dem Aufruf von EDIT wird wieder zurückgerufen.

STOP Ende des EDIT-Modus, der Textinterpreter meldet sich wieder.

Mit dem Wort EDIT werden Sie sich sicher der bequemen und übersichtlichen Handhabung wegen schnell anfreunden. In Zukunft sollten Sie angegebene Programmbeispiele stets in einem Screen ablegen und dann mit LOAD laden. Dann erübrigt sich auch die genaue Darstellung der Dialoge mit dem Textinterpreter bei der Eingabe von Programmen. Unsere Programmbeispiele werden also nur noch aus dem Text bestehen, den Sie im Screen ablegen können. Lediglich beim Testen der Programmbeispiele ist die Darstellung der Dialoge weiterhin zweckmäßig.

5. Programmschleifen und Programmverzweigungen - Die strukturierte Programmierung

Alle unsere bisherigen Programmbeispiele haben etwas gemeinsam. Die bisher von uns definierten Worte sind alle nach dem gleichen Prinzip aufgebaut und arbeiten deshalb sehr ähnlich. Sie bestehen einfach aus einer Folge von Anweisungen, ohne die Möglichkeit, eine Anweisung mehrmals auszuführen oder unter gewissen Umständen zu überspringen. Eine solche Programmstruktur, die einfach eine Folge von Anweisungen darstellt, von denen jede genau einmal ausgeführt wird, nennt man Sequenz.
Die Sequenz ist die einfachste Programmstruktur, außer ihr existieren noch die Verzweigung und der Zyklus. In diesem Kapitel werden Sie mit der Verzweigung und dem Zyklus näher bekanntgemacht. Die ausschließliche Verwendung von Sequenz, Verzweigung und Zyklus bei der Erstellung eines Programms nennen wir strukturierte Programmierung.
Warum eigentlich strukturierte Programmierung? Vielleicht haben Sie schon einmal ein mittelgroßes Programm in BASIC geschrieben. Das ist selbstverständlich möglich, ohne bewußt irgendwelche Programmstrukturen anzuwenden. Die Gefahr liegt nun darin, daß man einfach drauflos programmiert, ohne das Problem tiefer zu durchdenken. Das Programm wird immer größer, und man findet mal hier eine günstige Einsprungstelle, mal da, je nachdem, welche Aktivität gerade erforderlich ist. Zum Schluß, meistens kommt man irgendwie dahin, erhält man ein ineinander verflochtenes Programmungetüm, das man einige Zeit später selbst nicht mehr versteht. Selbst andernfalls wird es schwierig sein, etwa noch versteckte Fehler zu finden und zu korrigieren, da Änderungen an einer Stelle manchmal unübersehbare und nie gewollte Konsequenzen in anderen Programmteilen haben.
Das alles kann man weitgehend vermeiden, wenn das Programm gut durchdacht und strukturiert programmiert wurde. Die Programmiersprache FORTH zwingt Sie zu solchem Vorgehen. Bei BASIC können Sie so verfahren, müssen es aber nicht.

5.1. Unser Computer fällt Entscheidungen

In diesem Abschnitt lernen Sie die Programmverzweigung kennen. Die Verzweigung wird mit den drei Worten IF, ELSE und ENDIF programmiert.
Das Wort IF entscheidet, ob der Programmzweig zwischen IF und ELSE oder der zwischen ELSE und ENDIF abzuarbeiten ist.
Entscheidungsgrundlage ist die Zahl, die von IF auf dem Stack vorgefunden wird und dabei auch "verbraucht" wird. Ist diese Zahl ungleich Null, wird der Zweig zwischen IF und ELSE ausgeführt, anderenfalls der zwischen ELSE und ENDIF. Dazu die folgende Demonstration:

```
 : =0?   IF   ." UNGLEICH NULL     "
         ELSE ." GLEICH NULL       "
         ENDIF ; OK
 77 =0? UNGLEICH NULL     OK
  0 =0? GLEICH NULL       OK
 -2 =0? UNGLEICH NULL     OK
```

Innerhalb der Programmverzweigung dürfen wieder ganz beliebige Programmstrukturen auftreten, nur müssen diese dann in sich abgeschlossen sein. Auch können die Zweige leer sein. Ist der Zweig ELSE und ENDIF leer, kann ELSE auch weggelassen werden.

5.2. Die Vergleichsoperatoren

Entscheidend für die Auswahl des Programmzweiges IF ist, ob die Zahl im TOS gleich oder ungleich Null ist. Diese einfache Entscheidungsmöglichkeit reicht in den meisten Fallen nicht aus, z.B. für Zahlenvergleiche. Hier helfen uns die FORTH-Worte, die man unter dem Namen Vergleichsoperatoren zusammenfassen kann. Allen Vergleichsoperatoren ist gemeinsam, daß sie eine oder zwei Zahlen auf dem Stack verbrauchen und eine wieder dort ablegen. Die abgelegte Zahl ist entweder eine 1 oder eine 0, abhängig vom Ergebnis des Vergleichs:

```
wahre Aussage    --->    1                IF...ELSE-Zweig
falsche Aussage  --->    0                ELSE...ENDIF-Zweig
```

Es handelt sich hierbei um ein Flag, was natürlich keine Einschränkung für die weitere Verarbeitung bedeutet. Auch Flags sind ganz gewöhnliche Zahlen.

```
>       ( n1 n2 --> f )  f=1, wenn n1 > n2
<       ( n1 n2 --> f )  f=1, wenn n1 < n2
=       ( n1 n2 --> f )  f=1, wenn n1 = n2
0<      ( n     --> f )  f=1, wenn n  < 0
0=      ( n     --> f )  f=1, wenn n  = 0
```

Das Wort 0= hat (auf schon erzeugte Flags angewendet) die Eigenschaft, diese ins Gegenteil zu verkehren, also zu negieren. Deshalb ist es in einigen FORTH-Versionen auch unter dem Namen NOT zu finden.

5.3. Wir füttern Elefanten

Ein Elefant frißt 10 4-Pfund-Brote am Tag. Ob das nun ein großer oder ein kleiner Elefant ist, mag dahingestellt sein. Unser Computer soll uns nun sagen, ob der Elefant noch hungrig ist, wenn wir ihn mit einer bestimmten Anzahl Brote gefüttert haben. Was der Elefant bereits verschlungen hat, werden wir in der Variablen Futter speichern.

```
 0 VARIABLE FUTTER OK
```

Jeden Morgen verspürt der Elefant wieder neuen Appetit.

```
 : GUTEN-MORGEN  0 FUTTER ! ;
```

Ob unser Elefant noch hungrig ist, wollen wir anhand der Variablen FUTTER entscheiden.

```
: HUNGRIG?  CR FUTTER @ 3 <
            IF ." JA, SEHR!"
            ELSE FUTTER @ 10 <
             IF ." JA."
             ELSE FUTTER @ 15 <
              IF ." NEIN."
              ELSE ." BIN SCHON LAENGST SATT!"
              ENDIF
             ENDIF
            ENDIF CR ;
```

Wenn der Elefant gefüttert wird, erhöht die Anzahl der Brote den Inhalt der Variablen FUTTER. Natürlich kann auch ein Elefant nur eine positive Anzahl von Broten verspeisen.

```
: BROT CR DUP 0 >
  IF FUTTER +! ." DANKE. "
  ELSE 0= IF    ." ICH HABE NICHTS ZU KAUEN!"
          ELSE  ." ICH KANN DOCH KEINE AUSSPUCKEN!"
          ENDIF
  ENDIF CR ;
: BROTE BROT ;
```

Tippen Sie bei Jeder Fütterung die Anzahl der Brote ein.

```
GUTEN-MORGEN OK
HUNGRIG?
JA, SEHR!
OK
4 BROTE
DANKE!
OK
0 BROTE
ICH HABE NICHTS ZU KAUEN!
OK
HUNGRIG?
JA.
OK
-3 BROTE
ICH KANN DOCH KEINE AUSSPUCKEN!
OK
1 BROT
DANKE
OK
6 BROTE
DANKE
OK
HUNGRIG?
NEIN.
OK
5 BROTE
DANKE.
OK
HUNGRIG?
BIN SCHON LAENGST SATT!
OK
GUTEN-MORGEN OK
HUNGRIG?
JA, SEHR!
OK
```

5.4. Programmschleifen

Programmschleifen sind dadurch gekennzeichnet, daß Teile eines Programms mehrfach durchlaufen werden. Das ist häufiger nötig, als man gewöhnlich glaubt. Der Einsatz und das Programmieren eines Computers wird ja erst dann richtig effektiv, wenn häufig wiederkehrende Vorgänge durch ihn nach einem festen Schema bearbeitet, oder wenn Aufgaben evtl. mit unterschiedlichen Parametern automatisch vielfach wiederholt werden sollen.
Betrachtet man bestimmte Aktivitäten des Computers einmal genauer, so kommt man zum Schluß, daß viele nur mit Programmschleifen realisierbar sind. Ein schönes Beispiel sind die zyklischen Aktivitäten des Textinterpreters. Hier arbeiten sogar viele ineinander verschachtelte Schleifen.
Wir wollen die verschiedenen Arten von Programmschleifen unter zwei Gesichtspunkten sehen.

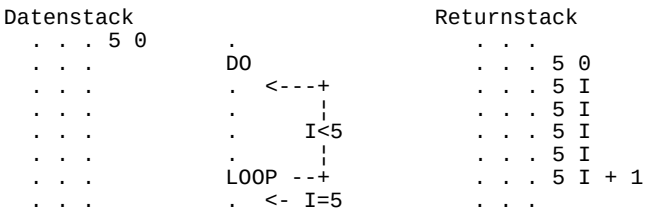
1. Steht die Anzahl der erforderlichen Schleifendurchläufe bereits vor Eintritt in die Schleife fest?
2. An welcher Stelle soll die Schleife verlassen und mit der der Schleife unmittelbar folgenden Anweisung fortgefahren werden?

Programmschleifen haben bei FORTH immer nur einen Eintrittspunkt und einen Austrittspunkt. Das ist zwar eine kleine, aber doch sehr sinnvolle Einschränkung im Sinne der Strukturierten Programmierung. Innerhalb einer Schleife sind wieder alle in sich geschlossenen Programmstrukturen erlaubt.

5.4.1. Schleifen mit fester Durchlaufzahl

Ein typisches Beispiel für ein FORTH-Wort, in dem eine Programmschleife mit fester Drchlaufzahl arbeitet, ist das Wort LIST. Wir wollen ja immer 16 Zeilen dargestellt haben, keine mehr und keine weniger.
Um eine Schleife mit fester Durchlaufzahl zu programmieren, benutzen wir die beiden Worte DO und LOOP. Dem Wort DO übergeben wir auf dem Stack den Endwert und den Startwert des Schleifenindex. Der Startwert muß dabei im TOS stehen. DO hat nichts weiter zu tun, als Start- und Endwert vom Datenstack zu entfernen und auf den Returnstack zu schaffen. LOOP erhöht den Schleifenindex auf dem Returnstack um eins und veranlaßt dann die Fortsetzung des Programmlaufs mit der Anweisung hinter DO, wenn der Endwert noch nicht erreicht ist. Ist der Enwert erreicht, wird die Schleife nicht noch einmal durchlaufen, sondern mit der Anweisung weitergearbeitet, die dem Wort LOOP folgt, nachdem natürlich die beiden Zahlen vom Returnstack wieder entfernt wurden.
Der Startwert wird auf dem Returnstack zuoberst abgelegt und dann als Schleifenindex behandelt. Wir wollen den Schleifenindex mit I bezeichnen.

Zur Veranschaulichung:



Die DO-LOOP-Schleife ist, betrachtet man sie unter dem im vorigen Abschnitt genannten zweiten Gesichtspunkt, eine solche, bei der der Austritt am Schleifenende erfolgt.
Nun noch ein einfaches Beispiel:

```
 : 3PIEPS  3 0 DO 7 EMIT LOOP ;
```

Der Schleifenindex läuft von 0 bis 2. Hat er 3 erreicht, wird die Schleife wieder verlassen.

Das Wort I schafft uns während des Schleifendurchlaufs den Index wieder auf den Datenstack. Es ist eng mit dem Wort R verwandt, welches genau das gleiche tut, nämlich den Top des Returnstack in den TOS zu kopieren.
Übrigens steht es uns völlig frei, wann wir die Parameter für DO bereitstellen, ob innerhalb der Wortdefinition oder erst später beim Auruf des Wortes.

```
 : PIEPS  0 DO CR I . 7 EMIT LOOP ;

 0 PIEPS
 0 OK
 1 PIEPS
 0 OK
 2 PIEPS
 1 OK
 3 PIEPS
 0
 1
 2 OK
 -10 PIEPS
 0 OK
```

Dieses Beispiel hat uns gezeigt, daß die Schleife völlig unhabhängig vom übergebenen Endwert mindestens einmal durchlaufen wird.
Mit dem Wort LEAVE können wir, wenn es innerhalb einer DO-LOOP-Schleife abgearbeitet wird (und nur hier ist es erlaubt), die Schleife vorzeitig verlassen. LEAVE kann irgendwo in der Schleife stehen. Es stellt nicht etwa einen neuen Austrittspunkt aus der Schleife dar, sondern es setzt lediglich den Scheifenendwert gleich dem Index, so daß dann LOOP am Schleifenende ganz verwundert die erfüllte Austrittsbedingung feststellt.

```
: STERNE  CR 0
          DO ?TERMINAL
                IF LEAVE
                ENDIF 42 EMIT
          LOOP ;
```

Zum Wort ?TERMINAL ist zu sagen, das es prüft, ob zwischenzeitlich eine Taste auf der Tastatur betätigt wurde. War dies nicht der Fall, so hinterläßt es auf dem Stack eine Null, anderenfalls eine Zahl ungleich Null.

```
10 STERNE
**********OK
```

Wenn Sie schreiben

```
32767 STERNE
```

und diese Anweisung mit ENTER quittieren, wird ihr Computer sicher einige Zeit benötigen, bis er mit dieser wichtigen Aufgabe fertig ist. Betätigen Sie zwischendurch eine Taste, tritt LEAVE in Aktion, und die Schleife wird augenblicklich verlassen.
Übrigens ist die Anweisungsfolge

```
?TERMINAL IF LEAVE ENDIF
```

auch im Wort LIST enthalten.
Zuweilen wird ein FORTH-Programmierer den Wunsch haben, den Schleifenindex pro Durchlauf nicht um Eins, sondern um beispielsweise 10 zu erhöhen. Man konnte dies mit der Sequenz

```
R> 9 + >R
```

erledigen (LOOP erhöht ja selbst auch um Eins). Es geht aber viel einfacher mit +LOOP. Dieses Wort übernimmt vom Stack eine Zahl, um die der Schleifenindex erhöht wird. Die Zahl kann sogar negativ sein. In diesem Fall wird die Austrittsbedingung genau andersherum getestet, nämlich ob der Index kleiner oder gleich dem Endwert geworden ist.

5.4.2. Wir befassen uns etwas genauer mit dem Stack

Das im vorigen Abschnitt gelernte Wissen wollen wir bei der Definition eines Wortes anwenden, das uns den gesamten Inhalt des Datenstacks auf den Bildschirm schreibt. Eine derartige Wortdefinitionen ist ihrer Nützlichkit wegen in vielen FORTH-Publikationen zu finden. Als Name hat sich .S eingebürgert.
Bevor wir mit der Definition beginnen, müssen wir uns über den Aufbau des Stacks im Klaren sein.
Jede 16-Bit-Zahl belegt im Stack genau 2 Bytes. Das niederwertige Byte belegt dabei auch die niederwertige Adresse, also genauso, wie beim Speichern von Variablen. Das bedeutet, daß wir mit dem Wort @ auch im Stack lesen können.
Das "untere" Ende des Stacks ist fest durch die Systemvariable S0 vorgegeben. S0 enthält die Initialisierungsposition des TOS, denn bei leerem Stack fallen "unteres" und "oberes" Ende des Stacks zusammen.
Werden nun Zahlen auf dem Stack abgelegt, so wächst dieser durch Verschiebung der TOS-Position in Richtung niedrigerer Adressen. Betrachten wir die Speicheradressen, so müssen wir sagen, der Stack

wächst nach unten. Daraus ergibt sich, daß die zuerst eingegebene Zahl auf der höchsten Adresse und die zuletzt eingegebene Zahl auf der niedrigsten Adresse (der TOS-Position) abgelegt ist.

Die TOS-Position, also die Adresse in der das niederwerige Byte des zuletzt eingegebenen Zahl abgelegt ist, läßt sich mit dem Wort SP@ ermitteln.
Nehmen wir an, die Zahlen 1 bis 4 sind der Reihenfolge nach eingegeben worden, die 1 als erste und die 4 als letzte Zahl. Dann sind die 4 im Stack "ganz unten" und die 1 "ganz oben" gespeichert. Betrachten wir aber die Adressen, so ergibt sich ein umgekehrtes Bild. Die zuletzt eingegebene Zahl 4 steht in der Zelle mit der derzeit niedrigsten Stackadresse, dem TOS.

```
                            Adresse                 Inhalt

Inhalt von S0  --->         Adr.    0              undefiniert
                            Adr.   -1                 00
                            Adr.   -2                 01
                            Adr.   -3                 00
                            Adr.   -4                 02
                            Adr.   -5                 00
                            Adr.   -6                 03
                            Adr.   -7                 00
TOS-Position   --->         Adr.   -8                 04
                            Adr.   -9              undefiniert
```

Wollen wir den Stack ausschreiben, so haben wir anfangs zwei Fälle zu unterscheiden.

1. Fall: Der Stack ist leer -> Ausgabe "LEER" -> Ende.
2. Fall: Der Stack ist nicht leer.

Mit Hilfe der DO-LOOP-Schleife und unter Verwendung der Stackadressen als Parameter werden die Stackadressen "SP@", die Stacktiefe "DPH" und die Stackinhalte "(SP@)" ausgegeben.
Die DO-LOOP-Schleife wird jeweils um 2 inkrementiert, da eine Zahl 2 Byte belegt. Die Stacktiefe wird durch die eigens dafür definierte Variable IND um 1 inkrementiert. Die Ausgabe erfolgt in Tabellenform mit einem Tabellenkopf, der die oben genannten Bezeichnungen trägt.

```
0 VARIABLE IND OK
: ZEI 0 DO 45 EMIT LOOP CR ; OK
: .S CR
  SP@ S0 @ -                                    (Stack leer?)
  IF 1 IND !                                    (Tabellenkopf)
     ." SP@  DPH    (SP@)" CR
     17 ZEI
     S0 @ SP@ 2+                                (Tabelleninhalt)
     DO I DUP .
        IND @ 4 .R 1 IND +!
        @ 9 .R CR
     2 +LOOP
  ELSE ." LEER" CR
  ENDIF
; OK
```

```
HEX OK
55 77 33 .S
SP@  DPH    (SP@)
-----------------
DA    1       33
DC    2       77
DE    3       55
OK
```

Beim Testen von FORTH-Programmen können Sie .S mal eben schnell zu Testzwecken in eine Wortdefinition einbinden. Auf das zu testende Programm hat .S keine Wirkung, lediglich dem Progammierer wird .S zuweilen den Anstoß zu einem Geistesblitz geben.

5.4.3. Programmschleifen mit variabler Durchlaufzahl

Nicht immer wird schon beim Eintritt in eine Schleife feststehen, wie oft diese durchlaufen werden muß, um den beabsichtigten Effekt zu erreichen. Deshalb stehen dem FORTH-Programmierer noch weitere Schleifentypen zur Verfügung, bei denen er den Test der Abbruchbedingungen selbst programmieren kann.
Für die Lage des Austrittspunktes innerhalb von Programmschleifen gibt es vier Möglichkeiten:

1. Austrittspunkt am Schleifenende (so wie bei der DO-LOOP-Schleife)
   realisiert durch die BEGIN-UNTIL-Schleife
2. Austrittspnkt innerhalb der Schleife
   realisiert durch die BEGIN-WHILE-REPEAT-Schleife
3. Austrittspunkt am Schleifenanfang
   auch realisierbar durch die BEGIN-WHILE-REPEAT-Schleife
4. kein Austrittspunkt
   realisiert durch die BEGIN-AGAIN-Schleife

Wenn wir die Schleife am Schleifenende verlassen wollen, schachteln wir die Aktivitäten der Schleife in die beiden Worte BEGIN und UNTIL ein. UNTIL entfernt bei jedem Durchlauf eine Zahl vom Stack und prüft dabei, ob diese gleich 0 ist. Ist dies der Fall, wird zu der Anweisung gesprungen, die auf BEGIN folgt und somit die Schleife noch einmal ausgeführt.
Wenn wir beispielsweise eine Schleife auf Tastendruck abbrechen wollen, bietet sich wieder das Wort ?TERMINAL an. Es paßt sehr schön mit UNTIL zusammen, wir können es direkt davor setzen.

```
: SISYPHUS  0
            BEGIN
             1+ CR ." DIES IST DER"
             DUP 4 .R ." . DURCHLAUF."
             ?TERMINAL
            UNTIL CR
            ." DANKE, SIE HABEN MICH ERLOEST!"
            DROP CR ;

SISYPHUS
DIES IST DER    1. DURCHLAUF.
DIES IST DER    2. DURCHLAUF.
DIES IST DER    3. DURCHLAUF.
DIES IST DER    4. DURCHLAUF.
DANKE, SIE HABEN MICH ERLOEST!
OK
```

Eine Schleife mit dem Austrittspunkt innerhalb ist die BEGIN-WHILE-REPEAT-Schleife. While entfernt bei jedem Durchlauf eine Zahl vom Stack und prüft, ob diese gleich 0 ist. Ist dies der Fall, wird die Schleife verlassen und die Programmabarbeitung mit dem Wort hinter REPEAT fortgesetzt. Ansonsten läuft das Programm weiter bis zu REPEAT und dann wieder zu der auf BEGIN folgenden Anweisung.

Das Wort ASCII soll uns nach jedem Tastendruck das entsprechende Zeichen und die dazugehörige Codierung entsprechend dem in unserem Computer verwendeten ASCII-Code auf den Bildschirm schreiben, allerdings mit einer Ausnahme: Beim Betätigen von ENTER wollen wir die Schleife sofort wieder verlassen.

```
: ASCII BEGIN CR KEY
         DUP 13 –
         WHILE
          DUP 31 >
          IF DUP EMIT
          ELSE ." STEUERZEICHEN"
          ENDIF
          ." HAT DEN CODE "
          . ." . "
          REPEAT
          ." DAS WAR ENTER. " CR ;

ASCII
A HAT DEN CODE 65 .
* HAT DEN CODE 42 .
@ HAT DEN CODE 64 .
STEUERZEICHEN HAT DEN CODE 8 .
0 HAT DEN CODE 48 .
DAS WAR ENTER.
OK
```

Auf diese Weise können Sie den ASCII-Code und Ihre Tastatur etwas besser kennenlernen.

Eine spezielle Schleife mit dem Austrittspunkt am Schleifenanfang gibt es in FORTH nicht. Hier läßt sich ebenfalls die BEGIN-WHILE-REPEAT-Schleife verwenden, indem wir dem BEGIN unmittelbar das Testen der Austrittsbedingung und WHILE folgen lassen.

Eine Sonderstellung nimmt die BEGIN-AGAIN-Schleife ein. In ihr wird kein Test auf irgendeine Abbruchbedingung durchgeführt. Es führt also kein Weg wieder hinaus. Dennoch sind auch solche Schleifen sinnvoll. Der Textinterpreter arbeitet zum Beispiel in einer solchen Endlosschleife. Es ist sogar sehr einfach, einen eigenen Textinterpreter zu schreiben, wie das folgende Beispiel zeigt.

```
: MEIN-FORTH  CR CR
              BEGIN CR ." READY " CR
                QUERY INTERPRET
              AGAIN ;
```

Das Wort QUERY holt einen Text von der Tastatur ab und hinterlegt ihn im Terminal-Input-Buffer. INTERPRET bearbeitet anschließend Ihren Text, so wie Sie es bisher von dem Textinterpreter gewohnt sind. Eigentlich müßte MEIN-FORTH solange arbeiten, bis Sie den Rechner ausschalten, da keine Austrittsbedingung aus der BEGIN-

AGAIN-Schleife existiert. Das wäre nicht weiter schlimm, da Sie ja genauso weiterarbeiten können wie bisher, nur mit dem Unterschied, daß anstelle des "OK" immer ein "READY" erscheint.

Praktisch ist es aber so, daß Sie sich nach jedem durch FORTH festgestellten Fehler sofort wieder im Original-Textinterpreter, der durch das Wort QUIT repräsentiert wird, befinden. Im Falle eines Fehlers wird ERROR aufgerufen, das nach der entsprechenden Meldung den Stack löscht und anschließend QUIT aufruft.
Wenn Sie im Wort MEIN-FORTH arbeiten, können Sie auf einfache Weise einen Programmabsturz erzeugen. Schreiben sie einfach

```
FORGET MEIN-FORTH
```

(oder irgendein anderes vorher definiertes Wort) und schon wird sich FORTH sang- und klanglos von Ihnen verabschieden und dabei noch einen mehr oder weniger großen Softwareschaden im Speicher anrichten. Normalerweise sägt man aber nicht an dem Ast, auf dem man gerade sitzt.

Falls Sie bei unserer Elefantenfütterung das ständige "OK" des Textinterpreters gestört hat, wissen Sie nun, wie man das unterdrücken kann:

```
: ZOO CR CR CR
  ." HERZLICH WILLKOMMEN BEI "
  ." UNS ELEFANTEN ! " CR CR
  BEGIN CR
   QUERY INTERPRET
  AGAIN ;
```

6. Für Fortgeschrittene

6.1. Wir definieren eigene Ausgabeoperatoren für Zahlen

Die Worte ., D., .R, D.R und U. arbeiten mit einem kleinen handlichen Satz von Worten, unter deren Verwendung man sich leicht eigene Ausgabeoperatoren definieren kann. Die Aktivität der Ausgabeoperatoren besteht darin, eine auf dem Stack liegende Zahl in eine Zeichenkette umzuwandeln und dann auszugeben. Sie wissen ja bereits, daß wir Zahlen in einer frei wählbaren Zahlenbasis ausgeben können. Da die Zahl auf dem Stack binär vorliegt, muß sie konvertiert werden. FORTH wendet dabei das Divisionsverfahren an: Die auszugebende Zahl wird fortlaufend durch die gerade vereinbarte Zahlenbasis dividiert. Der Divisionsrest entspricht der auszugebenden Ziffer, er muß nur noch in das entsprechende ASCII-Zeichen umgewandelt werden. Nehmen wir einmal an, wir haben als Zahlenbasis die Zahl 8 vereinbart, und die auszugebende Zahl sei 1453.

```
 1453 / 8 = 181  Rest 5     1. Ziffer von rechts
  183 / 8 =  22  Rest 5     2. Ziffer
   22 / 8 =   2  Rest 6     3. Ziffer
    2 / 8 =   0  Rest 2     4. Ziffer
    0 / 8 =   0  Rest 0     5. Ziffer
    0 / 8 =   0  Rest 0     6. Ziffer
```

Wir erhalten also, die führenden Nullen weggelassen, die Zeichenkette "2655". Folgendes soll das Beispiel verdeutlichen:

- Die Länge der entstehenden Zeichenkette ist zu Beginn der Konvertierung noch unbestimmt.
- Zuerst entsteht immer die Einerstelle, also die letzte Ziffer in der Zeichenkette.
- Um führende Nullen zu unterdrücken, muß die Division abgebrochen werden, wenn das Divisionsergebnis gleich Null ist.

Ein kleiner aber leistungsfähiger Satz von FORTH-Worten löst die hieraus entstehenden Probleme auf elegante Weise.
Die auszugebende Ziffernzeichenkette wird beginnend ab der Adresse PAD-1 aufgebaut. Allerdings in Richtung fallender Adressen. Auf diese Weise können keine Konflikte mit auf PAD zwischenlagernden Zeilen (Editor) entstehen. Die Systemvariable HLD zeigt immer auf die aktuelle Spitze der gerade aufzubauenden Zeichenkette. Hier sind nun die Bausteine, aus denen die Ausgabeoperatoren bestehen:

```
 S-> D      ( n --> d )
```

"Verwandelt" eine 16-Bit-Zahl in eine 32-Bit-Zahl. S->D muß aufgerufen werden, wenn 16-Bit-Zahlen auszugeben sind, da die Ausgabe auf 32-Bit-Zahlen abgestimmt ist.

```
 <#         ( --> )
```

Setzt die Variable HLD (Spitze der gerade aufzubauenden Speicherkette) auf PAD (Anfangsadresse des Editorspeichers). Nun kann die eigentliche Ausgabekonvertierung einer auf dem Stack liegenden 32-Bit-Zahl beginnen.

```
 #          ( ud1 --> ud2 )
```

Führt eine vorzeichenlose Division mit Rest der vorzeichenlosen 32-Bit-Zahl ud1 durch die aktuelle Zahlenbasis aus (mit M/MOD), so daß als Quotient die Zahl ud2 entsteht. Der Divisionsrest wird in ein ASCII-Zeichen umgewandelt, indem ganz einfach 48 oder 55 addiert wird, je nachdem, ob er kleiner als 10 ist oder nicht. Das entstandene ASCII-Zeichen wird unter Zuhilfenahme von HLD der Ausgabezeichenkette vorangestellt. HLD zeigt danach auf die neue Spitze der Zeichenkette.

```
 #S         ( ud --> 0 0 )
```

Ruft solange # auf, bis als Quotient eine doppeltgenaue Null entsteht. Damit werden alle erforderlichen Ziffern gebildet, ohne daß führende Nullen entstehen.

```
 #>         ( ud --> adr cnt )
```

Erwartet auf dem Stack das Überbleibsel der vorangegangenen #- und #S-Operationen, also gewöhnlich eine 32-Bit-Null. Diese wird ersetzt durch die Startadresse adr der Zeichenkette (steht in HLD) und deren Länge cnt (die Differenz zwischen PAD und dem Inhalt von HLD).

```
 TYPE       ( adr cnt --> )
```

Schreibt die Zeichenkette mit der Startadresse adr und der Länge cnt auf den Bildschirm.

Somit können wir uns schon ein ganz einfaches Wort zur Ausgabe vorzeichenloser 16-Bit-Zahlen selbst definieren.

```
 : U. S->D <# #S #> TYPE ;
```

Das Wort U. ist in ähnlicher Form bereits im FORTH-Vokabular enthalten.
Betrachten wir nun noch einmal unser Wort UD.BINAER aus dem Abschnitt 3.5.1.:
Mit <# 32 0 DO # LOOP #> erzeugen wir genau 32 Ziffern, die, falls 2 gerade Zahlenbasis ist, genau der bitweisen Zahlenrepräsentation entsprechen.
Wie wird nun das Vorzeichen erzeugt, wenn wir negative Zahlen ausgeben wollen? Bei negativen Zahlen würden ohne besondere Maßnahmen sowieso falsche Zeichenketten erzeugt, weil # und #S kein Vorzeichen berücksichtigen, sie arbeiten ja mit M/MOD. Mit Hilfe der beiden folgenden Worte können wir das negative Vorzeichen sowie beliebige andere Zeichen in die Ausgabezeichenkette einblenden.

```
 HOLD       ( c --> )
```

Erwartet auf dem Stack den Code c eines ASCII-Zeichens und stellt dieses der gerade im Aufbau befindlichen Zeichenkette voran, natürlich unter Benutzung von HLD. Es wird auch von # benutzt.

```
 SIQN       ( n ud --> ud )
```

Greift zerstörend auf die an dritter Position im Stack stehende Zahl n zu. Ist diese negativ, so wird mit Hilfe von HOLD ein Minuszeichen in die Ausgabezeichenkette eingefügt. SIGN kann eben-

falls irgendwo zwischen <# und #> aufgerufen werden. Als Zahl n eignet sich ganz hervorragend der höherwertige Teil der auszugebenden vorzeichenbehafteten 32-Bit-Zahl.

Was ist nun zur Einbeziehung des Vorzeichens zu tun? Ausgehend von einer 32-Bit-Zahl mit Vorzeichen auf dem Stack sind folgende 3 Schritte erforderlich:

1. Erzeugen der Zahl n (für SIGN) mit der Kopie des Vorzeichens, möglich mit der Sequenz SWAP OVER
2. Bilden des Absolutwertes der auszugebenden Zahl mit DABS
3. Aufruf von SIGN irgendwo zwischen <# und #>

Und so könnte unser Wort definiert sein:

```
 : D. <# SWAP OVER DABS #S SIGN #> TYPE SPACE ;
```

Wenn unser Computer Geldbeträge ausgeben soll (natürlich nur rein zahlenmäßig), werden wir das Vorzeichen hintenanstellen und außerdem einen Dezimalpunkt einfügen, so wie das in der Buchhaltung üblich ist (Code 46).

```
 : .GELD <# SWAP OVER DABS SIGN
            # # 46 HOLD #S #>
          TYPE SPACE ." MARK " ;
```

Damit unser Konto nicht auf 327.67 Mark beschränkt ist, arbeitet .GELD natürlich mit doppeltgenauen Zahlen.

```
 1000. .GELD 10.00 MARK OK
 10.00 .GELD 10.00 MARK OK
 -10000. .GELD 100.00- MARK OK
```

6.2. Wie sieht ein FORTH-Wort innen aus?

In den folgenden Abschnitten wollen wir uns etwas genauer mit dem FORTH-Compiler beschäftigen. Sie werden dann nach dem Verständnis dessen in der Lage sein, eigene Worte mit compilierendem Verhalten zu schreiben, etwa in der Art von :, VARIABLE, CONSTANT und USER. Diese Worte, wir nennen sie Definitionsworte, haben eines gemeinsam: Mit ihrer Hilfe werden wiederum Worte definiert, die sich später, wenn sie arbeiten, ähnlich verhalten. Ein Definitionswort erstellt also neue Worte mit gleichem Laufzeitverhalten. Beispielsweise haben alle Worte, die mit CONSTANT definiert wurden, die gemeinsame Eigenschaft, eine bestimmte Zahl zum Datenstack zu transportieren. Ganz ähnlich sieht es bei den Worten VARIABLE und USER aus, nur daß hier diese Zahl eine Speicheradresse darstellt, die Ihnen oder dem Computer frei zur Verfügung steht.
Auch alle Worte, die mit dem Doppelpunkt definiert worden sind, haben gleiches Laufzeitverhalten. Sie enthalten nämlich ausnahmslos eine mehr oder weniger lange Adressliste der in die Wortdefinition eingetragenen Worte. Diese Liste wird interpretiert, das heißt daß die Worte, deren Adressen in der Liste stehen, zur Laufzeit nacheinander ausgeführt werden. Das Programm, das dieses erledigt ist ganz kurz, genau 10 Bytes lang. Es heißt, da es Adressen interpretiert, Adressinterpreter. Die Laufzeitaktivität innerhalb einer Doppelpunktdefinition besteht darin, die alte Position des Adressinterpreters zwischenzuspeichern (auf dem Returnstack) und anschließend den Adressinterpreter die eigene Adressliste abarbeiten zu lassen.

Am Ende der Adressliste wird stets das Wort ;S auftauchen. Dieses stellt den Adressinterpreter wieder auf die alte Position um, die ja noch auf dem Returnstack steht.
Im Gegensatz zur Laufzeit steht die Compile-Zeit. Das ist die Zeit, zu der ein Definitionswort ein neues Wort ins Wörterbuch einträgt. Alle Definitionsworte bauen zur Compile-Zeit zunächst den Wortkopf auf. Der Kopf ist ein kleiner Speicherbereich, in dem die organisatorischen Angaben zum entsprechenden Wort stehen. Da bei FORTH alles sehr einfach zugeht, wird der Wortkopf einfach an das Ende des bestehenden Wörterbuchs angehängt, also ab der Adresse, die von HERE übergeben wird. Danach zeigt HERE hinter den gerade fertiggestellten Kopf. Aus anatomischer Sicht muß man annehmen, daß sich an den Kopf der Rumpf anschließt. Das ist tatsächlich so. Im Rumpf stehen die Angaben, die Worte mit gleichem Laufzeitverhalten voneinander unterscheiden, also bei Doppelpunktdefinitionen die unterschiedlichen Adresslisten, bei Konstanten und Variablen die unterschiedlichen Zahlenwerte.
In den folgenden Abschnitten wollen wir die einzelnen Komponenten des Wortkopfes einer genaueren Betrachtung unterziehen.

6.2.1. Die Namensfeldadresse (NFA)

Jeder Wortkopf beginnt mit der Namensfeldadresse. Hier finden wir als erstes das sogenannte Count-Byte. Count heißt es deshalb, weil es im wesentlichen die Anzahl der Zeichen enthält, die den Namen bilden. Diese Anzahl ist im Count-Byte in den Bits 0 bis 4 gespeichert. Daraus ergibt sich, daß Wortnamen nicht länger als 31 Zeichen werden dürfen (2 hoch 5 - 1).
Das Bit 5 des Count-Bytes ist das Smudge-Bit. Ist dieses Bit gesetzt, wird der Textinterpreter das Wort ignorieren. Das ist ein Schutzmechanismus, mit dem unfertige Worte vor versehentlichem Aufruf geschützt werden. Ein Systemabsturz wäre sonst die Folge. Das Wort SMUDGE schaltet dieses Bit im zuletzt hergestellten Wortkopf um. Bei normalem Ablauf einer Doppelpunktdefinition wird das Smudge-Bit erst beim Erreichen des Semikolons auf 0 gestellt.
Das Bit 6 im Count-Byte ist das Precedence-Bit. Ist dieses Bit gesetzt, wird das entsprechende Wort auch zur Compile-Zeit innerhalb einer Wortdefinition sofort ausgeführt und nicht in die Wortdefinition eingebaut. Das Precedence-Bit im zuletzt definierten Wort kann mit IMMEDIATE gesetzt werden. Wir kennen bereits einige Worte, die mit dieser Eigenschaft versehen sind. Einleuchtend ist das beim Semikolon, denn wie sollte man sonst eine Wortdefinition beenden können? Aber auch alle Worte, mit denen Programme strukturiert werden können (5. Kapitel), gehören dazu.
Das Bit 7 des Count-Bytes ist stets gleich 1.
Nach dem Count-Byte sind die einzelnen ASCII-Zeichen des Wortnamens gespeichert, beim letzten Zeichen ist wiederum das Bit 7 gesetzt, welches ja bei ASCII-Zeichen normalerweise nicht verwendet wird. Damit und mit dem gesetzten Bit 7 im Count-Byte können sehr leicht Anfang und Ende des Wortnamens lokalisiert werden.

So sieht also der vollständige Wortname im Wortkopf aus:

```
 Namensfeldadresse      ---->     Count-Byte
        +1                          1. Zeichen
        +2                          2. Zeichen
        +3                          3. Zeichen
        .                               .
        .                               .
        .                               .
        +n                          letztes Zeichen + 80H
```

Und hier noch einmal das Count-Byte im Einzelnen:

```
        Bit  7      stets gleich 1
             6      Precedence-Bit        (IMMEDIATE)
             5      Smudge-Bit            (SMUDGE   )
             4  -+
             3   ¦
             2    > Länge des Namens
             1   ¦
             0  -+
```

6.2.2. Die Linkfeldadresse (LFA)

Auf der Linkfeldadresse unmittelbar hinter dem letzten Zeichen des Wortnamens beginnt ein 2-Byte-Feld, in dem eine Adresse Platz findet. Es ist die Namensfeldadresse des zuvor definierten Wortes. Das enthält wiederum auf der Linkfeldadresse die Namensfeldadresse des zuvor definierten Wortes usw. Das allererste Wort, bei uns ist es LIT, enthält auf der Linkfeldadresse eine 0. So sind alle Worte miteinander verkettet. Damit wird im Wörterbuch die Suche nach einem bestimmten Wort ermöglicht.

6.2.3. Die Codefeldadresse (CFA)

Der Linkfeldadresse schließt sich die Codefeldadresse an. Es handelt sich hier ebenfalls um ein 2-Byte-Feld, auf dem die Adresse des auszuführenden Maschinenprogramms steht. Dieses Programm wird stets dann ausgeführt,wenn das Wort später einmal aufgerufen wird. Mit der Codefeldadresse werden die Laufzeiteigenschaften der Worte festgelegt.

6.2.4. Die Parameterfeldadresse (PFA)

Ab der Parameterfeldadresse beginnt der eigentliche "Nutzbereich" des FORTH-Wortes, der Rumpf. Bei Variablen ist dies nur ein 2-Byte-Feld, das zur Speicherung der Variablen verwendet wird. Auf der Codefeldadresse steht bei Variablen die Adresse eines kleinen Maschinenprogramms, das die Parameterfeldadresse zum Stack transportiert. Ähnlich ist das bei Konstanten, nur wird hier der Inhalt der Parameterfeldadresse zum Stack transportiert.
Bei Worten, die mit dem Doppelpunkt definiert wurden, beginnt auf der Parameterfeldadresse eine Liste, in der die Codefeldadressen der in die Wortdefinition eingebauten Worte sowie, falls erforderlich, deren Parameter stehen.

6.3. Wir schauen in ein FORTH-Wort hinein

Ein nützliches Wort, mit dessen Hilfe man die korrekte Wirkungsweise der selbst erstellten Definitionsworte überprüfen kann, ist DUMP. Es erwartet die Adresse und die Länge eines Speicherbereichs, der dann hexadezimal und (wenn möglich) als ASCII-Zeichen interpretiert auf dem Bildschirm erscheint. Weiterhin können wir mit den folgenden Worten beliebige Worte im Speicher aufspüren, die speziellen Adressen im Wortkopf suchen und den Namen des gefundenen Wortes wieder ausschreiben.

```
          ( --> pfa )
```

Dieses Wort (ausgesprochen "tick") sucht das nächste im Textpuffer stehende Wort und übergibt dessen Parameterfeldadresse.

```
 NFA       ( pfa --> nfa )
```
NFA erwartet eine Parameterfeldadresse und übergibt die entsprechende Namensfeldadresse.

```
 LFA       ( pfa --> lfa )
```
LFA erwartet eine Parameterfeldadresse und übergibt die entsprechende Linkfeldadresse.

```
 CFA       ( pfa --> cfa )
```
CFA erwartet eine Parameterfeldadresse und übergibt die entsprechende Codefeldadresse.

```
 PFA       ( nfa --> pfa )
```
PFA erwartet eine Namensfeldadresse und übergibt die entsprechende Parameterfeldadresse.

```
 ID.       ( nfa --> )
```
ID. erwartet eine Namensfeldadresse und schreibt den entsprechenden Wortnamen aus.

Schauen wir uns beispielsweise das Wort HEX an.

```
   HEX OK
```

Jetzt liegt die Parameterfeldadresse von HEX auf dem Stack.

```
 NFA OK
```

Statt der Parameterfeldadresse haben wir nun die Namensfeldadresse.

```
 20 DUMP

C973  83 48 45 D5 5E C9 2C C5  .HEX^I,E
C97B  91 C0 10 00 DA C6 E9 C4  .@..ZFiD
C983  6C C3 87 44 45 43 49 4D  1C.DECIM
OK
```

Sie erkennen sicher, daß dem Wort HEX das Wort DECIMAL folgt. Bei Ihnen müssen übrigens nicht die gleichen Adressen und Inhalte erscheinen wie in unserem Beispiel. HEX könnte Ja auch an ganz anderer Stelle im Speicher stehen.

Auf der Linkfeldadresse von HEX steht die Namensfeldadresse des mit HEX verketteten Vorgängerwortes, das wir jetzt suchen wollen.

```
 ' HEX LFA OK
```

Wir haben nun die Linkfeldadresse auf dem Stack. Mit @ ID. Erhalten wir den Namen des Vorgängerwortes.

```
 @ ID. SMUDGE OK
```

Beginnend ab der Parameterfeldadresse steht die eigentliche Wortdefinition mit einer Liste der Codefeldadressen der gerufenen Wörter. Haben wir einmal die Codefeldadresse, bekommen wir mit 2+ die Parameterfeldadresse und mit NFA die Namensfeldadresse.

```
 ' HEX @ 2+ NFA ID. LIT OK
```

LIT ist also das erste von HEX aufgerufene Wort. Es transportiert die auf der nachfolgenden Adresse stehende Zahl (die nun keine Codefeldadresse ist) zum Stack. Welche Zahl ist das hier?

```
 ' HEX 2+ @ . 16 OK
```

Schauen wir uns das Wort HEX weiter an:

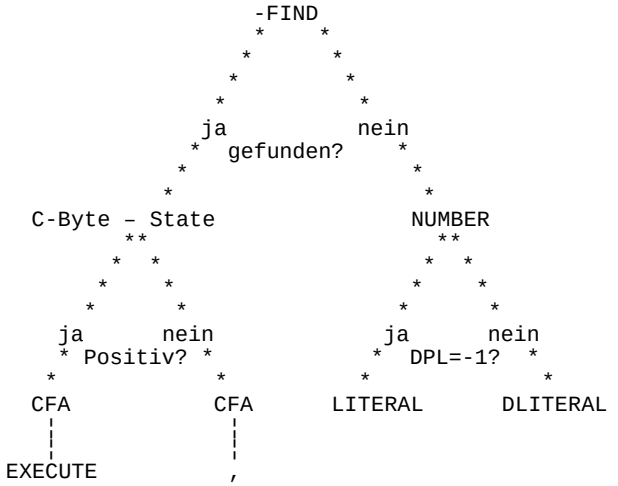
```
 ' HEX 4 + @ 2+ NFA ID. BASE OK
 ' HEX 6 + @ 2+ NFA ID. ! OK
 ' HEX 8 + @ 2+ NFA ID. ;S OK
```

Alles noch einmal zusammengefaßt, ergibt sich folgender innerer Aufbau:

```
 Rumpf von SMUDGE (Vorgängerwort)
-----------------------------------------------------------------------
 Kopf    83        Namensfeldadresse, COUNT-Byte, 3 Zeichen Länge
         48        ASCII - H
         45                E
         D8                X + 80H
         LFA       Linkfeldadresse, zeigt auf NFA von SMUDGE
         CFA       Zeigt zum Adressinterpreter  (Maschinencode)
-----------------------------------------------------------------------
 Rumpf   LIT       CFA von LIT
         16        Binärzahl, die von LIT zum Stack gebracht
                   werden soll
         BASE      CFA von BASE
         !         CFA von !
         ;S        CFA von ;S
-----------------------------------------------------------------------
 Kopf von DECIMAL (Nachfolgewort)
```

6.4. Wie arbeitet der Compiler?

Die entscheidenden Aktivitäten zur Einleitung des Compile-Vorganges werden vom Doppelpunkt ausgelöst. Der ruft zunächst CREATE auf. CREATE erstellt einen Wortkopf mit dem im Textpuffer dem Doppelpunkt folgenden Namen. Das Smudge-Bit ist noch gesetzt. Außerdem zeigt die Codefeldadresse vorläufig nach HERE, eine Eigenschaft, die wir später noch nutzen werden. Die Codefeldadresse wird im weiteren Verlauf so umgestellt, daß sie auf eine Maschinencodesequenz innerhalb des Doppelpunktwortes zeigt, womit die Laufzeiteigenschaften festgelegt werden. Abschließend wird der Compile-Mode abgeschaltet.
Im Compile-Mode arbeitet der Textinterpreter fast genauso weiter, wie bisher (Endlosschleife mit QUERY INTERPRET), nur mit dem kleinen Unterschied, daß nach dem Auffinden der Codefeldadressen nicht EXECUTE aufgerufen wird, sondern , (sprich "Komma"). Das Komma nimmt eine Zahl vom Stack, hier ist es eine Codefeldadresse, und trägt sie ins Wörterbuch ein. Das ist das Grundprinzip. Im Einzelnen laufen folgende Vorgänge ab:
Beim Aktivieren des Compile-Modes wird die Systemvariable STATE von 0 auf C0H gesetzt. Wenn nun der Textinterpreter mit -FIND die Namensfeldadresse des nächsten Wortes im Textpuffer gefunden hat, wird deren Inhalt (das Count-Byte) mit dem Inhalt von STATE verglichen. Ist das Count-Byte größer als der STATE-Inhalt, und das ist bei allen Immediate-Worten der Fall, wird das gefundene Wort mit Hilfe von EXECUTE ausgeführt, anderenfalls wird es mit Komma compiliert. Falls der Textinterpreter Zahlen vorfindet, wird hinterher LITERAL oder bei 32-Bit-Zahlen DLITERAL aufgerufen. Diese beiden Worte haben im Execute-Mode keinerlei Wirkung. Im Compile-Mode hingegen wird zunächst LIT ins Wörterbuch compiliert und dann die Zahl vom Stack. Bei DLITERAL wird das nocheinmal wiederholt.
Das folgende Schema soll die Arbeitsweise des Textinterpreters noch einmal deutlicher zeigen. Es sind, genauer gesagt, die Aktivitäten des Wortes INTERPRET.



Das Wort INTERPRET arbeitet als Endlosschleife, womit sich die Frage stellt, wie diese verlassen werden kann. DES Rätsels Lösung ist die Null, die am Ende jedes Textpuffers zu finden ist. Es gibt auch ein Wort, dessen Name nur aus einer echten 8-Bit-Null (dem NUL-Zeichen) besteht. Dieses Wort wird am Ende eines jeden Textpuffers aufgerufen. Seine wesentliche Aktivität ist R> DROP, also das Entfernen der Rückkehradresse vom Returnstack. Damit wird automatisch in der Programmebene weitergearbeitet, von der aus INTERPRET gerufen wurde.

6.5. <BUILDS und DOES>

Sie wissen jetzt, daß das Laufzeitverhalten aller mit dem Doppelpunkt definierten Worte darin besteht, die im Wort-Rumpf stehende Adressliste abzuarbeiten. Wir können das noch weiter treiben, indem wir eine Gruppe von Worten auch die gleiche Adressliste abarbeiten lassen. Damit statten wir die betreffenden Worte mit gleichen Laufzeitaktivitäten aus. Im Rumpf dieser Worte befindet sich ein Zeiger zur entsprechenden Adressliste. Hinter diesem Zeiger kann außerdem noch eine Parameterliste angelegt werden, deren Adresse zur Laufzeit auf dem Stack übergeben wird.
Praktisch geht das so, daß zunächst ein Definitionswort erstellt wird. Die Aktivitäten im Definitionswort sind in zwei Teile gegliedert:

1. Was ist zu tun, wenn mit Hilfe dieses Definitionswortes ein neues Wort definiert wird? Das betrifft hauptsächlich den Aufbau der Parameterliste.

2. Was soll das mit Hilfe des Definitionswortes neu definierte Wort später bei seinem Aufruf tun? Das betrifft hauptsächlich die Auswertung der Parameterliste und der evtl. auf dem Stack vorgefundenen Parameter.

Den ersten Teil schreiben wir in den <BUILDS-Teil der Definitionswort-Definition, den zweiten in den DOES>-Teil.
Ein Beispiel soll das verdeutlichen. Nehmen wir an, Sie sind Kioskverkäufer und gleichzeitig glücklicher Besitzer eines FORTH-Computers. Dann werden Sie den Computer sicherlich dazu benutzen, sich von anstrengender geistiger Routinetätigkeit zu entlasten, also vom Rechnen. Sie lassen den Computer für jeden Kunden alle Geldbeträge in einer Variablen aufsummieren. Danach wird deren Inhalt mit dem bereits im Punkt 6.1. definierten Wort .GELD ausgegeben. Anschließend wird unsere Variable wieder auf 0 gesetzt.

```
 0 VARIABLE SUMME

 : ZAHLEN  SUMME @ S->D .GELD 0 SUMME ! ;
```

Die gekauften Waren sollen mit der Anzahl und ihrem Namen eingegeben werden. Dabei ergibt sich für jede Ware die gleiche Aktivität: Die auf dem Stack vorgefundene Zahl mit dem Preis multiplizieren und zur Variablen SUMME addieren.
<BUILDS und DOES> sind dafür wie geschaffen. Definieren wir also zuerst ein Definitionswort WARE. Wenn WARE später die einzelnen Waren definiert, übernimmt es vom Stack den Preis und compiliert ihn ins Parameterfeld.

Die Definition des Wortes PILS könnte z.B. dann so aussehen:

```
 92 WARE PILS
```

Wenn PILS später arbeitet, soll es die 92 aus dem Parameterfeld holen, mit der auf dem Stack vorgefundenen Anzahl multiplizieren und zur Variablen SUMME addieren. Wagen wir uns nun an die Definition von WARE, zunächst den <BUILDS-Tei1:

```
 : WARE <BUILDS ,
```

Wenn WARE später arbeitet, hat es bis hierher bereits das neue Wort erstellt und den Preis ins Parameterfeld eingetragen. Wir müssen jetzt angeben, was die Waren später tun sollen. Mit dem Eintritt in den DOES>-Teil liegt die Adresse des Parameterfeldes bereits auf dem Stack, und mit @ haben wir sofort den entsprechenden Preis.

```
 DOES> @ * SUMME +! ;
```

Und nun können wir endlich anfangen, die einzelnen Waren zu definieren, eine kleine Auswahl soll für den Anfang genügen.

```
 92 WARE PILS
 72 WARE HELL
 128 WARE SPEZIAL
 -30 WARE PFAND
```

Haben wir nun die wichtigsten Waren definiert, kann der erste Kunde kommen.

```
 10 SPEZIAL 10 PFAND OK
 ZAHLEN 9.80 MARK OK
 20 PFAND ZAHLEN 6.00- MARK OK
```

6.6. Zum Beispiel POLYGON

POLYGON ist ein anspruchsvolleres Beispiel zu <BUILDS und DOES>. Mit Hilfe von POLYGON wollen wir Wörter definieren, die einen in ihrem Parameterfeld gespeicherten Streckenzug zeichnen. POLYGON soll die einzelnen Vektoren und deren Anzahl vom Stack übernehmen und ins Parameterfeld compilieren. Es soll folgendermaßen funktionieren:

```
 30 0 0 30 -30 0 0 -30
 4 POLYGON QUADRAT
```

Wir haben in diesem speziellen Beispiel vier Vektoren, von denen abwechselnd Delta-x und Delta-y vom Stack übernommen werden. Beim Compilieren müssen wir beachten, daß die Vektoren in umgekehrter Reihenfolge gezeichnet werden. Deshalb wird grundsätzlich vor dem Compilieren das Vorzeichen gewechselt. Das Ende der Vektorliste kennzeichnen wir mit dem Vektor 0 0, der nun freilich innerhalb des Streckenzuges nicht mehr vorkommen darf. Definieren wir also den <BUILDS-Teil:

```
 : POLYGON  <BUILDS 0
            DO MINUS , MINUS ,
            LOOP  00. , ,
```

Im DOES>-Teil soll der Streckenzug wieder gezeichnet werden. Zu Beginn des DOES>-Teiles benötigen wir das Flag für Zeichnen/Löschen und die Koordinaten des Startpunktes des Streckenzuges. Die Adresse der Vektorliste bekommen wir von DOES> noch dazu. Wir lassen nun eine BEGIN-WHILE-REPEAT-Schleife laufen, in der wir mit PLOT solange die Vektoren aus der Parameterliste zeichnen, bis wir auf den Vektor 0 0 stoßen.

```
 DOES>  >R
```

Damit liegt unser Adresszeiger auf der Vektorliste auf dem Returnstack, und so treten wir in die Schleife ein.

```
 BEGIN  R> DUP 4 + >R
```

Nun haben wir den Adresszeiger auf den nächsten Vektor justiert. Auf dem Stack liegt wegen DUP wieder die Adresse des aktuellen Vektors. Diesen holen wir uns auf den Stack, duplizieren ihn und benutzen das Duplikat zur Bildung eines Flags, das genau dann gleich 0 ist, wenn beide Komponenten des Vektors gleich 0 sind.

```
 2@  2DUP OR
```

Ist das Flag gleich 0, wollen wir die Schleife verlassen, anderenfalls haben wir alle Parameter für PLOT parat und zeichnen den Vektor.

```
 WHILE PLOT REPEAT
```

Nun beginnt das Spiel wieder von Neuem. Haben wir jedoch die Schleife verlassen, müssen wir noch beide Stacks in Ordnung bringen. Besonders wichtig ist das beim Returnstack. Nach R> haben wir 6 überflüssige Parameter auf dem Datenstack.

```
 R> DROP DROP DROP DROP DROP DROP ;
```

Noch einmal alles auf einen Blick:

```
 : POLYGON
   <BUILDS 0 DO MINUS , MINUS ,
           LOOP 00. , ,
   DOES>   >R BEGIN R> DUP 4 + >R
                  2@ 2DUP OR
              WHILE PLOT
              REPEAT
           R> DROP DROP DROP DROP DROP DROP ;
```

Wenn wir mit POLYGON arbeiten, ist zu beachten, daß der letzte Vektor zuerst gezeichnet wird. Der anvisierte Endpunkt ist beim Zeichnen der Startpunkt. Das spielt dann eine Rolle, wenn der Streckenzug nicht geschlossen ist, wie im folgenden Beispiel.

```
 -150  150  150    0
  -75   75  -75  -75
    0 -150  150  150
    0 -150 -150    0
 8 POLYGON (HAUS) (ENTER)                  (Definition von (HAUS))
 : HAUS_VOM_NIKOLAUS  1 70 15 (HAUS) ;     (Definition mit (HAUS))
```

Bitte, probieren Sie!

6.7. FORTH und Maschinencode

Dieser Abschnitt wendet sich an diejenigen Leser, die sich bereits mit der Maschinenprogrammierung des in unserem Computer verwendeten Mikroprozessors U880 beschäftigt haben. Sie sollten die interne Registerstruktur und die Wirkung der Befehle des Befehlssatzes kennen.
Bisher war es so, daß unsere Programmbeispiele auf jedem FORTH-Computer funktionieren, solange die verwendeten Worte dem Standardwortschatz entstammen.
Werden jedoch einzelne Worte in Maschinencode programmiert, so wie wir das jetzt vorhaben, gilt die Einschränkung, daß Programme nur noch auf solchen FORTH-Computern laufen, die mit dem Mikroprozessor U880 oder mit dessen Vergleichstyp, dem international weit verbreiteten Z80 arbeiten.
Warum kann eigentlich das Programmieren im Maschinencode sinnvoll sein? Es ist doch viel schwieriger und umständlicher.
Jedes Secondary-Wort, so wollen wir in Zukunft die Doppelpunktdefinitionen nennen, führt eine Reihe von Aktivitäten aus, die zum eigentlichen Programm nichts beitragen, sondern lediglich den Adressinterpreter steuern. Gelingt es jedoch, zeitkritische Worte im Maschinencode zu formulieren, so treten zwar die obenerwähnten Konsequenzen auf, das Programm wird aber wesentlich an Geschwindigkeit gewinnen.
Maschinencode-Worte, wir nennen sie Primitive-Worte, unterscheiden sich in ihrer Handhabung nicht von Secondary-Worten.
Wenn wir ein Primitive-Wort definieren wollen, werden wir unsere Maschinencode-Sequenz in den Rumpf eintragen. Dabei muß nur gesichert sein, daß der Adresszeiger in der Codefeldadresse auf den Anfang des Maschinencodes zeigt. Das ist bei der Erstellung eines Wortkopfes mit CREATE automatisch der Fall. Die Codefeldadresse enthält die Adresse von HERE, so daß wir mit , und C, unseren Maschinencode eintragen können. Am Ende der Maschinencodesequenz müssen wir einen Sprung in den Adressinterpreter programmieren. Dessen Eintrittspunkt übergibt uns die Systemkonstante NEXT.
Ganz wichtig ist, daß wir die Registerinhalte von BC, IX, IY und SP nicht zerstören. Das BC-Register enthält den aktuellen Zeiger auf die Adressliste innerhalb von Secondary-Worten. IX wird vom Betriebssystem CAOS benutzt und hat mit FORTH nichts zu tun. IY arbeitet bei uns als Returnstack-Zeiger. Der eigentliche Stackpointer des U880, das Register SP, verwaltet den Datenstack. Die Hintergrundregister dürfen nicht genutzt werden, da diese von FORTH-Worten benötigt werden.
Um den Sinn von Primitive-Worten zu demonstrieren, werden wir das gleiche Problem einmal als Secondary und einmal als Primitive programmieren.
Bei Adressberechnungen kommt es häufig vor, daß eine Multiplikation mit 2 erforderlich ist. Das läßt sich ganz einfach mit 2 * erledigen. Zu bedenken ist aber, daß Multiplikation und Division relativ zeitaufwendig sind. Es ist günstiger, anstelle 2 * die Sequenz DUP + anzuwenden. Wir wollen jedoch ein Wort 2* im Maschinencode definieren.

```
CREATE 2*
```

Nun ist unser Wortkopf bereits fertig, lediglich das Smudge-Bit ist noch gesetzt.

```
 HEX  E1    C,    ( POP  HL     )
      29    C,    ( ADD  HL,HL  )
      E5    C,    ( PUSH HL     )
      C3    C,
      NEXT   ,    ( JMP  NEXT   )
 DECIMAL SMUDGE
```

Zum Wort 2* definieren wir noch das Wort 2/.

```
 CREATE 2/
 HEX  E1    C,    ( POP  HL     )
      CB    C,
      2C    C,    ( SRA H       )
      CB    C,
      1D    C,    ( RR  L       )
      E5    C,    ( PUSH HL     )
      C3    C,
      NEXT   ,    ( JMP NEXT    )
 DECIMAL SMUDGE
```

Nun definieren wir drei Testworte, mit deren Hilfe wir die Verarbeitungsgeschwindigkeit gut abschätzen können.
Wir lassen einfach eine DO-LOOP-Schleife 10000 mal laufen und stoppen von Hand die Programmlaufzeit.

```
 : TEST1     10000 0 DO     LOOP ;

 : TEST2   4 10000 0 DO 2 * 2 / LOOP DROP ;

 : TEST3   4 10000 0 DO  2*  2/ LOOP DROP ;
```

Nun lassen wir alle drei Worte ablaufen und stoppen die Zeit, bis der Textinterpreter sich wieder meldet. Die Laufzeiten richten sich nach der Taktfrequenz des Computers, die folgenden Werte sind deshalb nur relativ zueinander zu betrachten.

```
 TEST1    ca. 2 Sekunden
 TEST2    ca. 75 Sekunden
 TEST3    ca. 3 Sekunden
```

Die reine Arbeitszeit in der Schleife erhalten wir, indem wir die Werte der leeren Schleife bei TEST2 und TEST3 subtrahieren und durch 10000 dividieren.
Wir erhalten deshalb die Ausführungzeiten von

```
 2 * 2 /    mit 7300 Mikrosekunden
```

und von

```
 2* 2/      mit 100 Mikrosekunden.
```

Wir konnten also in diesem speziellen Fall die Verarbeitungsgeschwindigkeit auf das mehr als 70-fache erhöhen.

6.8. Noch einmal neue Definitionsworte

Auch dieser Abschnitt wendet sich wieder an die mit dem Maschinencode bereits vertrauten Leser. Das Neue, das wir jetzt probieren wollen, besteht darin, Definitionsworte zu schaffen, bei denen das Laufzeitverhalten der damit definierten Worte im Maschinencode formuliert ist.
Das Verfahren ist ähnlich, wie bei <BUILDS und DOES>. Wir haben wieder zwei Teile des Definitionswortes. Der erste Teil sieht wie eine gewöhnliche Doppelpunktdefinition aus. Wir schließen ihn jedoch nicht mit dem Semikolon ab, sondern mit dem Wort ;CODE. Anschießend tragen wir wieder (wie gewohnt mit , und C,) unseren Maschinencode ein. Dieser Maschinencode wird später zur Laufzeit des neu definierten Wortes (ähnlich wie der DOES>-Teil) ausgeführt. Auch hier benötigen wir wieder den Zugriff auf das aufgebaute Parameterfeld. Dessen Adresse steht beim Eintritt in den Maschinencode als nützliches Überbleibsel der vorangegangenen Operationen um 1 vermindert im DE-Register. Um die Füllung des Parameterfeldes kümmern wir uns wie bei <BUILDS im vorderen Teil der Doppelpunkt-Definitionswort-Definition.
Definieren wir beispielsweise ein neues Wort VARIABLE, das die Variable nicht bereits zur Definitionszeit mit einem Wert belegt, sondern lediglich im Parameterfeld den entsprechenden Platz reserviert. In einigen FORTH-Installationen ist VARIABLE generell so definiert.

```
: VARIABLE  CREATE    ( Kopf erstellen                    )
    2 ALLOT           ( Platz reservieren                 )
    SMUDGE            ( neue Variable für gültig erklären )
; CODE
```

Im Parameterfeld haben wir Platz für eine 16-Bit-Zahl geschaffen (2 ALLOT). Deren Adresse steht nach einem Inkrement im DE-Register, dessen Inhalt noch zum Stack geschafft werden muß. Nun definieren wir das Laufzeitverhalten:

```
HEX  13    C,     ( INC DE   )
     D5    C,     ( PUSH DE  )
     C3    C,
     NEXT   ,     ( JMP NEXT )
DECIMAL SMUDGE
```

Das SMUDGE ist hier erforderlich, weil es in ;CODE im Gegensatz zu ; nicht enthalten ist. Unser neues Variablendefinitionswort ist jetzt fertig, und wir können es folgendermaßen anwenden:

```
VARIABLE HASE OK
1111 HASE ! OK
HASE ? 1111 OK
```

Lassen Sie sich von der Meldung Nr.4 bei der Definition von VARIABLE nicht stören. Sie sollen lediglich daran erinnert werden, daß es das Wort VARIABLE schon einmal gibt. Damit ist für alle zukünftigen Anforderungen das neue Wort VARIABLE gültig.

Als krönenden Abschluß definieren wir ein Wort, mit dem wir eindimensionale Felder erzeugen und verwalten können. Wir nennen es VEKTOR. Zur Definitionszeit übernimmt VEKTOR die Anzahl der Feldelemente (die Länge des Feldes) und reserviert entsprechenden Platz im Wörterbuch. Als Feldelemente wollen wir 16-Bit-Zahlen speichern. Zur Laufzeit wird der Feldindex übernommen und dafür

die Adresse des entsprechenden Feldelementes übergeben.

```
: VEKTOR CREATE  ( Kopf erstellen                          )
   2 * ALLOT     ( 2 * Feldgröße Bytes reservieren         )
   SMUDGE        ( Feld gültig erklären                    )
; CODE           ( nun folgt der Laufzeitcode              )
HEX   E1   C,    ( POP  HL     ; Feldindex                 )
      29   C,    ( ADD  HL,HL  ; mal 2                     )
      13   C,    ( INC  DE     ; Parameterfeldadresse      )
      19   C,    ( ADD  HL,DE  ; Elementadresse=2*Index+Pfa )
      E5   C,    ( PUSH HL     ; Elementadresse zum Stack  )
      C3   C,
      NEXT  ,    ( JMP NEXT                                )
DECIMAL SMUDGE
```

Da die mit VEKTOR definierten Worte Adressen übergeben, dürfen wir mit ! und @ Speichern und Lesen.

```
10 VEKTOR ZIFFERN
: FUELL  10 0
  DO I 48 + I ZIFFERN ! LOOP ;

FUELL OK
0 ZIFFERN @ CR EMIT CR
0
OK
9 ZIFFERN @ CR EMIT CR
9
OK
```

Das Wort VEKTOR birgt so, wie es hier definiert wurde, in sich eine Gefahr. Es wird keinerlei Kontrolle des übergebenen Index durchgeführt, d.h. bei zu großem oder zu kleinem Index werden uns Adressen übergeben, deren Inhalt auf keinen Fall zerstört werden darf. Da aber (fast) kein Programm von Anfang an fehlerfrei ist, könnte es während der Erprobungsphase durchaus einmal passieren, daß der Index nicht im zulässigen Bereich liegt. Wenn wir dann in (genauer gesagt: neben) das Feld schreiben, ist ein Systemabsturz sehr wahrscheinlich.
Nun könnte man das Wort VEKTOR so definieren, daß eine Fehlerkontrolle enthalten ist. Zum einen wird aber dadurch das Programm langsamer, zum anderen wird die Fehlerkontrolle beim fertigen und korrekten Programm nicht mehr ansprechen müssen.
Ein Ausweg könnte der sein, daß man ein Wort mit nach außen gleicher Wirkung zunächst mit <BUILDS und DOES> definiert und dabei eine komfortable Fehlerortung einbezieht.
Ist man sich der fehlerfreien Programmfunktion sicher, wird das "vorsichtige" VEKTOR gegen das schnelle ausgetauscht.

Das "vorsichtige" VEKTOR könnte so aussehen:

```
 : VEKTOR
   <BUILDS DUP 1 - , 2 * ALLOT
   DOES>   DUP @ ROT MIN 0 MAX
           2 * 2+ +
   ;
```

Im Falle der Nichteinhaltung des zulässigen Indexbereichs wird entweder 0 oder der mögliche Maximalwert angenommen. Versuchen Sie selbst, die genaue Fuktionsweise anhand des Stackdiagramms zu verstehen. Im nächsten Kapitel werden wir VEKTOR noch einmal definieren. Dabei werden wir im Fehlerfall eine entsprechende Fehlermeldung erzeugen.

7. FORTH-Organisation

7.1. Die Meldungen

Nach dem Programmanlauf von FORTH werden alle Systemmeldungen nur mit ihrer Nummer ausgegeben. Die Texte, die diesen Meldungen entsprechen, sind nicht im Wort MESSAGE enthalten, dessen Aufgabe es ist, Systemmeldungen auszugeben. Dadurch wird Speicher gespart. Wenn Sie aber den Komfort lieben, so schreiben sie die im Anhang B gegebenen Texte in die Screens 1 und 2. Dann teilen Sie FORTH mit, daß die Texte verfügbar sind, indem Sie WARNINS (eine Systemvariable) auf 1 setzen. Benötigen Sie jedoch auch diese beiden Screens für eigene Zwecke, so verzichten Sie auf die Meldungstexte und setzen WARNING wieder auf 0.
Das Wort MESSAGE erwartet auf dem Stack die Nummer der entsprechenden Meldung. Die Nummern sind den Zeilen in den beiden Screens fest zugeordnet, d.h. der erste Screen enthält die Meldungen 0 bis 15, der zweite die von 16 bis 31. Sie können selbst in den freien Zeilen und auch in den nächsten Screens weitere Meldungen festlegen, die dann weiter durchnumeriert sind.
Bei Diskettenorientierten FORTH-Installationen beginnen die Meldungen erst auf dem Screen 4. Das würde sich aber hier störend auswirken, wenn insgesamt nur 8 Screens bei minimaler Speicherausstattung zur Verfügung stehen.
WARNING bietet noch eine dritte Möglichkeit, Fehler zu bearbeiten. Enthält WARNING -1, so wird nach jedem Fehler (ABORT) ausgeführt. Sie können nun in die Parameterfeldadresse von (ABORT) die Codefeldadresse eines Wortes zur Fehlerbehandlung eintragen, das Sie selbst definiert haben. Voraussetzung dafür ist aber, daß sich (ABORT) im RAM-Bereich befindet.
Bauen wir nun beispielsweise eine Fehlermeldung in das Wort VEKTOR ein, die erscheinen soll, wenn beim betreffenden eindimensionalen Feld der zulässige Indexbereich nicht eingehalten wurde. Wir erteilen der Meldung die Nummer 9. Mit dem Editor tragen wir in Screen 1 Zeile 9 den gewünschten Fehlertext ein, z.B. "index out of range". Nach 1 WARNING ! probieren wir:

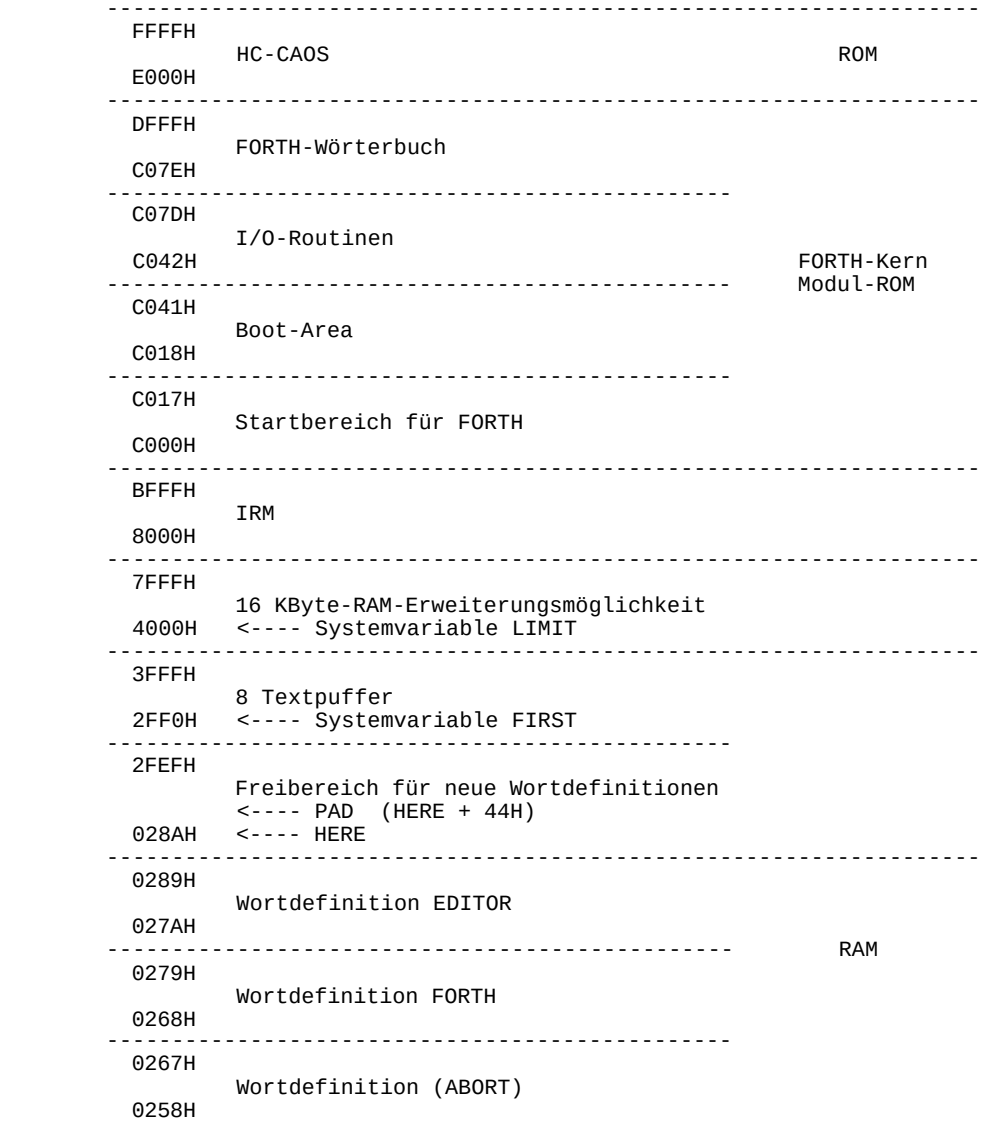
```
 9 MESSAGE index out of range OK
```

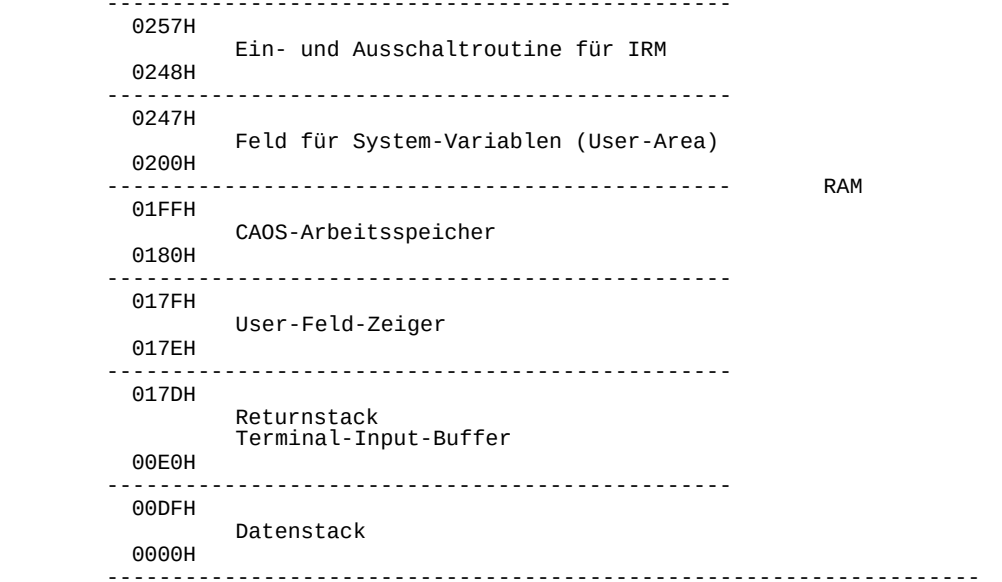
VEKTOR wird nun unter Verwendung von ?ERROR noch einmal definiert. Versuchen Sie wieder selbst, die Funktion anhand des Stackdiagramms zu verstehen.

```
: VEKTOR
  <BUILDS DUP 1 - , 2 * ALLOT
  DOES>   DUP ROT SWAP @
          OVER < 9 ?ERROR
          DUP 0< 9 ?ERROR
          2 * 2+ + ;
10 VEKTOR HUT OK
0 HUT 20 ERASE OK
0 HUT ? 0 OK
-1 HUT HUT? index out of range
0 WARNING ! OK
10 HUT HUT? MSG # 9
-1 WARNING ! OK
10 HUT
KC-FORTH
1 WARNING !
```

7.2. Die Speicherbelegung

In diesem Abschnitt werden Sie damit vertraut gemacht, wie FORTH den Arbeitsspeicher des Computers verwaltet und belegt. Sie können selbst die Speicherbelegung Ihren eigenen Erfordernissen anpassen, indem Sie entsprechende Systemvariablen mit eigenen Werten belegen.
Der FORTH-Kern ist im Modul auf ROM ab der Adresse C000H gespeichert. Die folgende Übersicht stellt die Speicheraufteilung dar.





7.2.1. Der FORTH-Kern

Der FORTH-Kern ist der Grundbaustein des FORTH-Systems. Er läßt sich in 4 Bereiche unterteilen.

- Das FORTH-Wörterbuch enthält alle FORTH-Worte, es macht den weitaus größten Teil des Kerns aus.
- Die I/O-Routinen realisieren die Arbeit mit dem Bildschirm und der Tastatur, über diese Routinen sind FORTH und CAOS miteinander verbunden.
- Das Boot-Area ist ein Feld, auf dem Startwerte fuer Systemvariablen stehen, die beim Anlauf unbedingt erforderlich sind. Die Anfangsadresse des Boot-Area erhält man mit 0 +ORIGIN.
- Startbereich für FORTH. Diesen Bereich benutzt FORTH nicht, hierüber wird aber die Eintragung ins CAOS-Menue realisiert.

Beim Start des FORTH-Systems sind der Kaltstart und der Warmstart zu unterscheiden. Der Kaltstart (über das Menuewort FORTH) muß stets beim erstmaligen Systemanlauf ausgeführt werden, außerdem dann, wenn die Situation bezüglich des vorausgegangenen (möglicherweise unkorrekten) Programmlaufs unklar ist. Einen Warmstart (Menuewort REFORTH) können Sie immer dann ausführen, wenn Sie FORTH vorher korrekt verlassen haben (mit BYE). Alle hinzudefinierten Worte bleiben Ihnen dann erhalten.
Die Einsprungadresse erhalten Sie folgendermaßen:

```
 0 +ORIGIN      für Kaltstart
 4 +ORIGIN      für Warmstart
```

7.2.2. Der Freibereich für neue Wortdefinitionen

Die Anfangsadresse des freien Wörterbuchbereichs steht in der Systemvariablen DP und läßt sich hierüber auch manipulieren.

7.2.3. Der Textpufferbereich

Die Lage des Textpufferbereichs wird durch die Systemvariablen FIRST und LIMIT festgelegt. Man muß dabei einen Kompromiß zwischen einer ausreichenden Anzahl von Textpuffern und einem genügend großen Freibereich für neue Wortdefinitionen finden.

7.2.4. Das User-Feld

Im User-Feld stehen die Werte der Systemvariablen. Eigentlich heißen diese Variablen User-Variablen. Diese Bezeichnung ist aber nur bei solchen FORTH-Systemen sinnvoll, an denen mehrere Benutzer gleichzeitig arbeiten. Jeder Benutzer hat hier seinen eigenen User-(Nutzer-) Variablensatz.
Die Lage des User-Feldes im Speicher ist mit dem User-Feld-Zeiger festgelegt.

7.2.5. Der User-Feld-Zeiger

Im User-Feld-Zeiger steht die Anfangsadresse des User-Feldes. Sollte es sich erforderlich machen, irgendwo ein anderes User-Feld anzulegen, so trägt man dessen Anfangsadresse in den User-Feld-Zeiger ein. Zu beachten ist dabei aber, daß vorher (!) das neue User-Feld bereits mit sinnvollen Werten belegt sein muß. Im einfachsten Fall kopiert man das alte Feld einfach ins neue (CMOVE).

7.2.6. Die Stack-Bereiche und der Terminal-Input-Buffer

Der Returnstack und der TIB belegen einen gemeinsamen Speicher-bereich. Konflikte sind jedoch weitgehend ausgeschlossen, da der Returnstack diesen Bereich von "oben nach unten" und der TIB von "unten nach oben" belegt. Sollte der Bereich zu klein werden, können mit den Systemvariablen R0 und TIB sowohl der Returnstack als auch der Terminal-Input-Buffer an anderer Stelle angelegt werden.
Auch der Raum für den Datenstack ist mit ca. 100 freien Plätzen für 16-Bit-Zahlen recht knapp bemessen. Mit S0 läßt sich die Position dieses Stacks beeinflussen. Eine günstige Alternative ist die Lage des Datenstacks am Ende des freien Wörterbuchbereichs, also z.B. mit FIRST @ S0 ! SP ! .

7.3. Vokabulare

Unter einem Vokabular verstehen wir eine Gruppe von Worten, die unter einem Namen, nämlich dem Vokabularnamen zusammengefaßt sind. Bei der Arbeit mit Vokabularen dürfen sie einerseits das Vokabular auswählen, dem Wortdefinitionen anzufügen sind und andererseits das Vokabular, das bei Suchläufen zuerst durchlaufen werden soll. Ersteres nennen wir Current-Vokabular, das zweite ist das Context-Vokabular. Die Systemvariable VOC-LINK ist ein Zeiger auf das zuletzt definierte Vokabularnamenswort, in diesem befindet sich wieder ein Zeiger auf das zuvor definierte usw. bis zum Namenswort des FORTH-Vokabulars (das natürlich FORTH heißt), wo der Zeiger den Wert 0 hat. Damit sind die einzelnen Vokabulare ähnlich miteinander verkettet, wie die Worte innerhalb der Vokabu-lare.

Das Definitionswort VOCABULARY erzeugt ein neues Vokabularnamenswort, das aber noch in dem Vokabular liegt, das gerade als Current-Vokabular vereinbart war.

```
VOCABULARY ZOO
```

erzeugt uns ein Vokabularwort mit dem Namen ZOO, das allerdings noch zum FORTH-Vokabular gehört. Andere Konsequenzen hat die Definition von ZOO noch nicht, weder das Current- noch das Context-Vokabular sind verändert worden. Weitere Wortdefinitionen würden zum FORTH-Vokabular gehören. Wenn nun ZOO ausgeführt wird, erklärt es sich selbst zum Context-Vokabular. Dies hat aber immer noch keine Konsequenzen für weitere Wortdefinitionen, weil FORTH immer noch das Current-Vokabular ist. Erst nach der Ausführung von DEFINITIONS wird CURRENT gleich CONTEXT (Systemvariablen) gesetzt und damit ZOO zum Current-Vokabular erklärt.
Um ein neues Vokabular aufzubauen, sind also drei Aktivitäten erforderlich:

```
VOCABULARY ZOO          ( Definition des Namenswortes )

ZOO                     ( neues Vokabular als Context-Vokabular
                          deklarieren )

DEFINITIONS.            ( Gleichsetzen des Current-Vokabulars
                          mit dem Context-Vokabular )
```

Alle folgenden Wortdefinitionen gehören nun zum Vokabular ZOO. Innerhalb von ZOO können weitere Vokabulare definiert werden.

```
VOCABULARY AFFEN
VOCABULARY REPTILIEN
VOCABULARY RAUBKATZEN
```

Diese Vokabulare sind nur über ZOO erreichbar, also wenn ZOO Context-Vokabular ist. Den Vokabularen können in zwangloser Folge Wortdefinitionen zugeordnet werden.

```
AFFEN      DEFINITIONS
: GORILLA     ;
: SCHIMPANSE  ;
RAUBKATZEN DEFINITIONS
: PUMA        ;
REPTILIEN  DEFINITIONS
: BOA         ;
RAUBKATZEN DEFINITIONS
: TIGER       ;
: LOEWE       ;
REPTILIEN  DEFINITIONS
: KROKODIL    ;
```

Auch das Vokabular ZOO kann noch erweitert werden.

```
ZOO        DEFINITIONS
VOCABULARY RAUBVOEGEL
RAUBVOEGEL DEFINITIONS
: BUSSARD     ;
: FALKE       ;
: HABICHT     ;
FORTH DEFINITIONS
```

Wir können nun auf einfache Weise verfolgen, wie unser künstlich geschaffenes Durcheinander von Vokabularen und Wortdefinitionen durch FORTH geordnet wird.
Alle Suchläufe beginnen stets im Context-Vokabular. Dann geht es weiter in dem Vokabular, in dem Context seinen Ursprung hat usw. bis das FORTH-Vokabular erreicht ist. Beginnen wir beispielsweise im Vokabular AFFEN. Wir müssen dann erreichen, daß AFFEN Context-Vokabular wird.

```
 AFFEN AFFEN? MSG # 0
```

Die AFFEN sind über das FORTH-Vokabular nicht zu erreichen, da AFFEN ja seinen Ursprung im Vokabular ZOO hat. Also machen wir zunächst ZOO zum Context-Vokabular.

```
 ZOO OK
 VLIST

 RAUBVOEGEL  RAUBKATZEN  REPTILIEN  AFFEN  ZOO  TASK
 DUMP  EDITOR  ...  OK
```

Jetzt sind die Affen zu erreichen.

```
 AFFEN OK
 VLIST

 SCHIMPANSE  GORILLA  RAUBVOEGEL  RAUBKATZEN  REPTILIEN
 AFFEN  ZOO  TASK  DUMP  EDITOR  ...  OK

 RAUBVOEGEL OK
 VLIST

 HABICHT  FALKE  BUSSARD  RAUBVOEGEL  RAUBKATZEN  REPTILIEN
 AFFEN  ZOO  TASK  DUMP  EDITOR  ...  OK
```

Man kann die Ordnung der Vokabulare mit einem Baum vergleichen. Der Stamm ist das FORTH-Vokabular. Jedes Vokabular, das definiert wurde, als FORTH Context war, ist ein Zweig, der vom Stamm ausgeht. Ein Zweig kann sich beliebig oft weiter verzweigen.
Vokabularsuchläufe beginnen stets an der Spitze des Context-Zweiges in Richtung Wurzel. Alle Vokabulare auf dem Weg zur Wurzel werden stets vollständig durchsucht, d.h. es spielt keine Rolle, wann die einzelnen Wortdefinitionen einem Vokabular hinzugefügt wurden.
Sollen innerhalb einer Wortdefinition Vokabulare umgeschaltet werden (etwa, um Worte aus unterschiedlichen Vokabularen in eine Wortdefinition einzubauen), so ist es zweckmäßig, die Vokabularnamensworte als "immediate" zu erklären, um zu erreichen, daß sie auch im Compile-Mode ausgeführt werden.
Zu beachten ist auch, daß zu Beginn einer Doppelpunktdefinition das Current-Vokabualar automatisch auch als Context-Vokabular deklariert wird, eine Aktivität des Doppelpunktes. Innerhalb der Wortdefiniton kann nach vorübergehendem Verlassen des Compile-Modes (mit "((") das Context-Vokabular umgeschaltet und danach in den Compile-Mode zurückgekehrt werden. Definieren wir beispielsweise im FORTH-Vokabular ein Wort, das das Editorwort EDIT aufruft und vorher noch eine Zahl aus dem Stack in der Systemvariablen SCR abspeichert.

```
FORTH DEFINITIONS
: EDIT SCR ! (( EDITOR )) EDIT ;
```

So müßte man verfahren, wäre EDITOR kein Immediate-Wort. Da das bei EDITOR aber der Fall ist, geht es auch einfacher.

```
: EDIT SCR ! EDITOR EDIT ;
FORTH
```

Abschließend ist es zweckmäßig, FORTH wieder zum Context-Vokabular zu erklären.
Nun haben wir zwei Worte, die EDIT heißen. Durch ihre Zugehörigkeit zu unterschiedlichen Vokabularen sind sie aber leicht zu trennen.
Als ganz besonderer Service wird durch das Wort -FIND, welches allgemein für Suchläufe zuständig ist, nach dem Context-Vokabular auch das Current-Vokabular durchsucht, wenn das zu suchende Wort nicht schon im Context-Vokabular gefunden worden ist.

7.4. Die Systemvariablen

Alle Systemvariablen sind in einem speziellen Feld zusammengefaßt, dem User-Feld. Das Vorhandensein dieses Feldes resultiert aus zwei Gründen. Zum Einen ist es bei FORTH-ROM-Versionen nicht möglich, Variablen im Kern abzuspeichern. Zum Anderen ist damit ein Mehrnutzerbetrieb bestens vorbereitet. In einem solchen Fall arbeiten mehrere Benutzer parallel mit dem gleichen Computer und dem gleichen FORTH-Kern, wobei jeder den Eindruck hat, der Computer sei für ihn allein da. Das funktioniert jedoch nur, wenn der Computer auch die hardwareseitigen Vorraussetzungen erfüllt. In einem solchen Fall steht jedem Nutzer ein eigener Speicherbereich zu, in den die Wörterbucherweiterungen eingetragen werden, in dem sich ein eigenes User-Feld, eigene Daten- und Returnstackbereiche sowie ein eigener Terminal-Input-Buffer befinden. Das FORTH-System schaltet nun der Reihe nach von einem auf den nächsten Nutzer um. Jedem Nutzer steht der Computer also nur eine bestimmte Zeit zur Verfügung. Dieser merkt aber nichts davon, weil der Zeitrythmus sehr schnell ist.
Die Aktivitäten beim Umschalten von einem Nutzer auf den Nächsten sind sehr einfach. Es ist nur der User-Area-Pointer umzustellen und die im User-Feld geretteten Positionen des Daten- und des Returnstack-Pointers und des Instruction-Pointers neu einzustellen. Zu diesem Zweck befinden sich am Anfang des User-Feldes drei freie Plätze für die entsprechenden Adressen.
Zum Definieren eigener Variablen, die noch mit im User-Feld gespeichert werden sollen, bedient man sich des Definitionswortes USER. Dabei muß man natürlich die Größe und die Belegung des User-Feldes kennen. Entnehmen Sie dies bitte dem Anhang C.
Formal gleicht die Definition einer User-Variablen der einer "normalen" Variablen. Wir übergeben dem Definitionswort USER jedoch nicht den Inhalt, den die Variable nach der Definition erhalten soll, sondern die relative Adresse zum Startpunkt des User-Feldes. Damit erhält eine User-Variable nach der Definition noch keinen bestimmten Wert.
Definieren wir beispielsweise 2 User-Variablen, in denen wir uns die Positionen der beiden Stackpointer merken wollen. Wir dürfen dazu die er ten drei freien Zellen des User-Feldes benutzen.

```
 0 USER DSP    ( Datenstackpointer )
 2 USER RSP    ( Returnstackpointer )
```

DSP und RSP verhalten sich genauso, wie Variablen, d.h. bei ihrem Aufruf legen sie ihre entsprechende Adresse auf den Stack. Mit DSP erhalten wir sogar die genaue Startadresse des User-Feldes, die natürlich gleich der Adresse sein muß, die wir im User-Area-Pointer vorfinden.

```
 HEX OK
 DSP . 200 OK
 17E ? 200 OK
```

7.5. Arbeit mit Peripheriegeräten

Im Normalfall stehen uns im KC-System die Tastatur, der Bildschirm und evtl. ein Recorder zur Verfügung. Um nun aber auch andere Geräte (Drucker, Schreibmaschine usw.) von FORTH aus betreiben zu können, gibt es FORTH-Wörter, die die Arbeit mit diesen Geräten unterstützen. Im KC 85 wird die Ein- und Ausgabe über das Betriebssystem mit Zeigern auf Unterprogrammen gesteuert.
Im Normalfall steht der Zeiger für den Eingabekanal auf dem Unterprogramm Nr. 04, d.h. die Eingaben erfolgen über die Tastatur. Mit dem Wort SETIN ist es möglich, den Zeiger auf ein anderes Eingabeunterprogramm des Betriebssystems zu setzen. Die Eingabe

```
 6 SETIN
```

bewirkt, daß jetzt ASCII-Zeichen über den Anwendereingabekanal 1 empfangen werden können.
Die Spezifikation der Anwenderkanale kann der Anwender vornehmen. So ist es z.B. möglich, eine V24-Schnittstelle zu verwenden, und man kann Daten von Tastaturen mit V24-Schnittstelle oder anderen Computern empfangen und FORTH damit bedienen.
Mit dem FORTH-Wort NORMIN kann auf die normale Eingabe (Tastatur) umgeschaltet werden.

Der Zeiger für den Ausgabekanal steht im Normalfall auf dem Unterprogramm Nr. 0 (Ausgabe auf den Bildschirm). Mit dem Wort SETOUT kann der Zeiger auf eine andere Unterprogramm-Nr. gesetzt werden. Mit

```
 2 SETOUT
```

können ASCII-Zeichen über den Anwenderausgabekanal 1 ausgegeben werden (z.B. zur Drucker- oder Schreibmaschinensteuerung).
Mit dem FORTH-Wort NORMOUT kann auf die normale Ausgabe (Bildschirm) umgeschaltet werden.

Folgende CAOS-Unterprogramme stehen dem Anwender für die Ein- und Ausgabe zur Verfügung:

UP-NR.	Funktion	Bemerkungen
0	Bildschirmausgabe	Standart für die Ausgabe
2	Anwenderausgabe-kanal 1	Startadresse des Treiberprogramms muß in UOUT1 (0B7BEH) eingetragen werden /1//2/
3	Anwenderausgabe-kanal 2	Startadresse des Treiberprogramms muß in UOUT2 (0B7C4H) eingetragen werden
4	Tastatureingabe	Standart für die Eingabe
6	Anwendereingabe-kanal 1	Startadresse der Empfangsroutine muß in UIN1 (0B7C1H) eingetragen werden
7	Anwendereingabe-kanal 2	Startadresse der Empfangsroutine muß in UIN2 (0B7C7H) eingetragen werden

Beispiel:
Ein KC 85/2 oder KC 85/3 soll mit einem FORTH-Modul M 026 betrieben werden und einen Matrixdrucker K6313 über einen V24-Modul M 003 steuern.

Vorgehensweise:

- KC 85 ausschalten
- V24-Modul in Kanal 8 kontaktieren
- FORTH-Modul in Kanal C kontaktieren
- Verbinden von Drucker und V24-Modul (Kanal 1) mittels 5-poligen Diodenkabel /3/
- Drucker und KC 85 einschalten
- Laden der Treiberroutine V24K6313COM von Kassette C 0171
- Starten des FORTH-Moduls
- Eingabe 2 SETOUT

Durch die Treiberroutine wurde das KC-System bereits so initialisiert, daß mit " SETOUT schon eine Druckerausgabe möglich ist. Die Eingabe

```
2 SETOUT
3 LIST
```

bewirkt z.B., daß der Screen 3 auf dem Drucker ausgedruckt wird. Mit dem Wort NORMOUT wird der Zeiger auf die Normale Ausgabetabelle eingestellt (Bildschirmausgabe).

8. FORTH im Überblick

In diesem Kapitel finden Sie eine Erläuterung aller FORTH-Worte, die Ihr Computer "versteht". Die meisten dieser Worte sind im Standart der FORTH-Interrest-Group (fig-Standard) enthalten. Zuweilen treten jedoch hinsichtlich der Wirkung Abweichungen auf, die in einer besseren Ausnutzung der Hardware des Kleincomputers begründet sind, bzw. in dessen anderer Massenspeicher-Organisation (Magnetband anstelle Diskette). Andere Worte sind gar nicht im fig-Standard enthalten. Sie stellen entweder eine mögliche Erweiterung dieses Standards dar, oder es wird mit ihnen ein Anschluß von FORTH an das Betriebssystem CAOS realisiert, um über FORTH auf einfache Weise die wichtigsten Fähigkeiten von CAOS zu nutzen.
Bei allen Worten, die nicht mit dem fig-Standard übereinstimmen, ist dies gesondert vermerkt.
Alle Worte werden mit der Stackbelegung vor und nach dem Aufruf und ihren Aktivitäten beschrieben. Für die Darstellung der Stackbelegung gelten folgende Vereinbarungen:

- Links stehen die Übernahme-, rechts die Übergabeparameter.
- Der TOS befindet sich auf beiden Seiten jeweils rechts
- Die Parameter werden mit folgender Symbolik dargestellt:

n	16 Bit-Zahl mit Vorzeichen
u	16 Bit-Zahl ohne Vorzeichen
d	32 Bit-Zahl mit Vorzeichen
ud	32 Bit-Zahl ohne Vorzeichen
b	8 Bit-Byte
c	7 Bit-ASCII-Zeichen
f	logisches Flag, das entweder ein True- oder ein False-Flag ist
tf	True-Flag, ist immer ungleich 0
ff	False-Flag, ist immer gleich 0
adr	16-Bit-Speicheradresse
TOS	Top of stack

Die Worte sind nach Sachgebieten geordnet.

8.1. Arithmetik

8.1.1. einfach genaue Operatoren

```
+     ( n1 n2 --> n3 )
----------------------
Addiert n1 und n2 zur Summe n3.

-     ( n1 n2 --> n3 )
----------------------
Aus n1 und n2 wird die Differenz n3 gebildet.

*     ( n1 n2 --> n3 )
----------------------
Multipliziert n1 mit n2 zum Produkt n3.

/     ( n1 n2 --> n3 )
----------------------
Bildet den Quotienten n3 aus n1/n2.
```

```
/MOD     ( n1 n2 --> n3 n4 )
----------------------------
```
Übergibt den Quotienten n4=n1/n2 und den Teilerrest n3.

```
MOD     ( n1 n2 --> n3 )
-------------------------
```
Übergibt den Teilerrest n3 aus der Division n1/n2.

```
MIN     ( n1 n2 --> n3 )
-------------------------
```
Übergibt n3 als die kleinere der beiden Zahlen n1 und n2.

```
MAX     ( n1 n2 --> n3 )
-------------------------
```
Übergibt n3 als die größere der beiden Zahlen n1 und n2.

```
+!     ( n adr --> )
--------------------
```
Addiert n zu der Variablen, deren Adresse auf dem Stack liegt.

8.1.2. gemischt und doppelt genaue Operatoren

```
*/     ( n1 n2 n3 --> n4 )
--------------------------
```
Multipliziert n1 und n2 zu einem 32-Bit-Zwischenergebnis und dividiert dieses dann durch n3, so daß der Quotient n4 entsteht.

```
*/MOD     ( n1 n2 n3 ---> n4 n5 )
---------------------------------
```
Multipliziert n1 und n2 zu einem 32-Bit-Zwischenergebnis und dividiert dieses dann durch n3, so daß der Quotient n5 entsteht. Zusätzlich wird der Teilerrest n4 übergeben.

```
M*     ( n1 n2 --> d )
----------------------
```
Multipliziert die beiden einfach genauen Zahlen n1 und n2 zum doppelt genauen Produkt d.

```
M/     ( d n1 --> n2 n3 )
-------------------------
```
Dividiert die doppelt genaue Zahl d durch die einfach genaue Zahl n1. Die Zahl n2 ist der Teilerrest mit dem gleichen Vorzeichen wie d, und n3 ist der Quotient.

```
D+     ( d1 d2 --> d3 )
-----------------------
```
Addiert zwei doppeltgenaue Zahlen zu einem doppeltgenauen Ergebnis.

M/MOD (ud1 u1 --> u2 ud2)

Führt eine vorzeichenlose Division der doppelt genauen Zahl ud1 durch u1 aus. Die Zahl ud2 ist der doppelt genaue Quotient und u2 der Teilerrest.

U* (u1 u2 --> ud)

Die beiden vorzeichenlosen Zahlen u1 und u2 werden zum doppelt genauen vorzeichenlosen Produkt ud multipliziert.

U/ (ud u1 --> u2 u3)

Die doppelt genaue vorzeichenlose Zahl ud wird durch die einfach genaue vorzeichenlose Zahl u1 dividiert. Es entstehen der einfach genaue Quotient u3 und der Teilerrest u2.

8.1.3. Sonstiges

ABS (n1 --> n2)

Bildung des Absolutwertes n2 von n1.

DABS (d1 --> d2)

Bildung des Absolutwertes d2 der doppelt genauen Zahl d1.

MINUS (n1 --> n2)

Vorzeichenwechsel, n2 = -n1.

DMINUS (d1 --> d2)

Vorzeichenwechsel einer doppelt genauen Zahl, d2 = -d1.

+- (n1 n2 --> n3)
---
n3 ist der Absolutwert von n1 mit dem Vorzeichen von n2.

D+- (d1 n --> d2)
---
d2 ist der Absolutwert der doppelt genauen Zahl d1 mit dem Vorzeichen von n.

S->D (n --> d)

Wandelt die einfach genaue Zahl n in die doppelt genaue Zahl d um.

```
1+     ( n1 --> n2 )
--------------------
```
Erhöhung der Zahl im TOS um 1.

```
2+     ( n1 --> n2 )
--------------------
```
Erhöhung der Zahl im TOS um 2.

8.2. Zahlensysteme

```
DECIMAL     ( --> )
-------------------
```
Setzt die aktuelle Zahlenbasis auf 10.

```
HEX     ( --> )
---------------
```
Setzt die aktuelle Zahlenbasis auf 16.

```
BASE     ( --> adr )
--------------------
```
Systemvariable, die die Zahlenbasis enthält, mit der alle Ein- und Ausgaben abgewickelt werden.

8.3. Logik-Operatoren

```
AND     ( n1 n2 --> n3 )
------------------------
```
Die Zahlen n1 und n2 werden bitweise logisch UND verknüpft, so daß n3 entsteht.

```
OR     ( n1 n2 --> n3 )
-----------------------
```
Die Zahlen n1 und n2 werden bitweise logisch ODER verknüpft, so daß n3 entsteht.

```
XOR     ( n1 n2 --> n3 )
-----------------------
```
Die Zahlen n1 und n2 werden bitweise logisch EXCLUSIV-ODER-verknüpft, so daß n3 entsteht. Ein 1-Bit in n2 bewirkt, daß das entsprechende Bit in n1 negiert wird.

8.4. Vergleichsoperatoren

```
=     ( n1 n2 --> f )
---------------------
```
Ersetzt n1 und n2 durch ein Flag, das genau dann wahr ist (ungleich 0), wenn n1 gleich n2 ist.

```
>     ( n1 n2 --> f )
---------------------
```
Ersetzt n1 und n2 durch ein Flag, das genau dann wahr ist, wenn n1 größer als n2 ist.

```
<     ( n1 n2 --> f )
---------------------
```
Ersetzt n1 und n2 durch ein Flag, das genau dann wahr ist, wenn n1 kleiner als n2 ist.

```
U<     ( u1 u2 --> f )
----------------------
```
Ersetzt die beiden vorzeichenlosen Zahlen u1 und u2 durch ein Flag, das genau dann wahr ist, wenn u1 kleiner als u2 ist. Dieses Wort ist nicht im fig-Standart enthalten.

```
0=     ( n --> f )
------------------
```
Das Flag f ist dann wahr, wenn n = 0 ist. Dieses Wort führt eine logische Negation aus. In einigen FORTH-Installationen steht es deshalb auch unter dem Namen NOT.

```
0<     ( n --> f )
------------------
```
Ersetzt n durch ein Flag, daß genau dann wahr ist, wenn n kleiner als 0 ist.

8.5. Stackmanipulationen

8.5.1. Datenstack

```
DROP     ( n --> )
------------------
```
Entfernt die Zahl n vom Stack.

```
DUP     ( n --> n n )
---------------------
```
Dupliziert die Zahl n auf dem Stack.

```
-DUP     ( n --> n   )  falls n=0
         ( n --> n n )  falls n ungleich 0
-----------------------------------------
```
Dupliziert eine Zahl n auf dem Stack nur, wenn diese ungleich 0 ist.

```
SWAP     ( n1 n2 --> n2 n1 )
---------------------------
```
Vertauscht die beiden obersten Zahlen auf dem Stack.

2DUP (d --> d d)

Dupliziert eine doppelt genaue Zahl auf dem Stack. Dieses Wort ist nicht im fig-Standard enthalten.

OVER (n1 n2 --> n1 n2 n1)

Kopiert die an zweiter Stackposition stehende Zahl zum jetzt neuen TOS.

ROT (n1 n2 n3 --> n2 n3 n1)

Vertauscht zyklisch die drei obersten Zahlen im Stack. Die Zahl an dritter Position gelangt zum TOS.

SP@ (--> adr)
Übergibt die Position des TOS, wie sie vor dem Aufruf von SP@ vorlag.

8.5.2. Returnstack

>R (n -->)

Übertragt eine Zahl aus dem TOS in den Top des Returnstacks, eine einfache Möglichkeit zur Zwischenspeicherung. Das Wort ist nur innerhalb von Wortdefinitionen erlaubt, Vorsicht auch bei DO...LOOP!

R> (--> n)

Überträgt eine Zahl vom Top des Returnstacks in den TOS.

R (--> n)

Kopiert den Top des Returnstacks zum TOS. Der Returnstack bleibt erhalten.

RP@ (--> adr)

Übergibt die augenblickliche Position des Top vom Returnstack.

8.6. Manipulation im Speichers

8.6.1. Bytes und Zellen

C@ (adr --> b)

Ersetzt die Adresse adr durch das dort vorgefundene Byte.

C! (n adr -->)

Speichert das niederwertige Byte von n auf die Adresse adr.

@ (adr --> n)

Ersetzt adr durch die auf adr und adr+1 vorgefundene Zahl n.

! (n adr -->)

Speichert das niederwertige Byte von n auf adr und das höherwertige Byte auf adr+1.

2@ (adr --> d)

Ersetzt adr durch die auf adr, adr+1, adr+2 und adr+3 vorgefundene doppelt genaue Zahl d. Dieses Wort gehört nicht zum fig-Standard.

2! (d adr -->)

Speichert die doppelt genaue Zahl d beginnend ab Adresse adr bis adr+3. Dieses Wort gehört nicht zum fig-Standard.

TOGGLE (adr b -->)

Verknüpft das auf adr vorgefundene Byte bitweise EXCLUSIV-ODER mit b. Die Bits auf adr werden negiert (toogle), wenn deren Entsprechung in b gleich 1 ist.

8.6.2. Speicherbereiche

CMOVE (adr1 adr2 u -->)

Kopiert einen Speicherbereich ab Adresse adr1 mit der Länge u nach adr2.

FILL (adr u b -->)

Füllt einen Speicherbereich der Länge u ab Adresse adr mit dem Byte b.

ERASE (adr u -->)

Füllt einen Speicherbereich der Länge u ab Adresse adr mit Nullen (Hex 00).

BLANKS (adr u -->)

Füllt einen Speicherbereich der Länge u ab Adresse adr mit Leerzeichen (Hex 20).

8.7. Ein- und Ausgabe

8.7.1. Ein- und Ausgabe einzelner Zeichen

KEY (--> c)

Wartet auf eine Tastaturbetätigung und übergibt dann den ASCII-Cade c der gedrückten Taste.

?TERMINAL (--> f)

Testet den Status der Tastatur und übergibt ein Flag, das genau dann wahr ist, wenn zwischenzeitlich eine Taste gedrückt wurde.

EMIT (c -->)

Das Zeichen c wird auf dem Ausgabegerät ausgegeben. Normalerweise ist dies der Bildschirm. Die System-Variable OUT wird um 1 inkrementiert.

8.7.2. Spezielle Ausgaben

CR (-->)

Die Kombination carriage return – line feed (Wagenrücklauf - Zeilenschaltung) wird an das Ausgabegerät ausgegeben. Auf dem Bildschirm rückt der Cursor an den Anfang der nächsten Zeile. Die Systemvariable OUT wird nicht beeinflußt.

SPACE (-->)

An das Ausgabegerät wird ein Leerzeichen ausgegeben.

SPACES (u -->)

An das Ausgabegerät werden u Leerzeichen ausgegeben.

8.7.3. Ausgabe von Zeichenketten

TYPE (adr u -->)

Eine ab Adresse adr im Speicher stehende Zeichenkette der Länge u wird an das Ausgabegerät ausgegeben.

COUNT (adr1 --> adr2 b)

Erwartet die Adresse einer Zeichenkette im FORTH-Standard-Format. Hierbei ist der eigentlichen Zeichenkette ein Byte vorangestellt, das die Länge der Zeichenkette enthält. Dieses Count-Byte wird auf adr1 erwartet. COUNT übergibt das Count-Byte b auf dem TOS und die eigentliche Anfangsadresse adr2 des Textes. So wird der Anschluß

zum Wort TYPE hergestellt.

OUT (--> adr)

OUT ist eine Systemvariable, die bei jedem ausgegebenen Zeichen um 1 erhöht wird. Sie wird vom FORTH-System nicht benutzt und kann deshalb beliebig, zum Beispiel zur Ausgabeformatierung, verwendet werden.

-TRAILING (adr n1 --> adr n2)

Für eine Zeichenkette ab Adresse adr wird deren Länge n1 so verändert, daß Leerzeichen am Textende unterdrückt werden.

." (-->)

Im Execute-Mode wird der folgende Text bis zu einem abschließenden " an das Ausgabegerät ausgegeben.
Im Compile-Mode wird das Wort (.") zusammen mit dem folgenden Text (bis zum abschließenden ") in das Wörterbuch compiliert. Der Text hat dann das FORTH-Standard-Format, beginnend mit dem Count-Byte. ." ist ein Immedate-Wort.

(.") (-->)

Dieses Wort wird von ." zusammen mit einem Text im FORTH-Standard-Format in Wortdefinitionen eingebaut. Wenn das Wort, das das Wort (.") enthält, aufgerufen wird, gibt (.") den im Wörterbuch folgenden Text an das Ausgabegerät aus.

8.7.4. Eingabe von Zeichenketten

(EXPECT) (adr n -->)

Erwartet die Eingabe einer durch ENTER abgeschlossenen Zeichenkette, die höchstens n Zeichen lang ist. Diese wird beginnend ab adr in den Speicher eingetragen. Das Ende wird mit einer 0 markiert.
Im Unterschied zum fig-Standard werden hier die Zeichen von der Cursorposition abgelesen, und alle Steuerzeichen haben ihre bekannte Wirkung und werden nicht im Speicher eingetragen.

EXPECT (adr n -->)

Ist das aktuelle Ausgabegerät der Bildschirm, wird (EXPECT) aufgerufen. Bei Verwendung anderer Ausgabegeräte (Schreibmaschine, Drucker) werden als Steuerzeichen nur Cursor links und ENTER akzeptiert. Alle anderen Steuerzeichen und alfanumerischen Zeichen werden in die Eingabezeichenkette eingebaut.

TIB (--> adr)

TIB ist eine Systemvariable, sie enthält die Adresse des Terminal-Input-Buffers.

```
QUERY     ( --> )
-----------------
```
Erwartet die Eingabe von maximal 80 Zeichen Text (mit EXPECT) und hinterläßt diesen Text im Terminal-Input-Buffer. Die Systemvariable IN wird auf 0 gesetzt.

8.8. Ausgabe von Zahlen

8.8.1. Die Bausteine der Ausgabeoperatoren

```
<#     ( --> )
--------------
```
Ist die erste Aktivität bei der Umwandlung einer doppelt genauen Zahl in eine Zeichenkette. <# initialisiert die Systemvariable HLD, die einen Zeiger auf den aktuellen Anfang der im Aufbau befindlichen Zeichenkette enthält, mit der Adresse PAD.

```
#     ( ud1 --> ud2 )
---------------------
```
Dividiert ud1 durch die aktuelle Zahlenbasis, so daß ud2 entsteht. Der Divisionsrest wird in ein ASCII-Zeichen umgewandelt und auf die in HLD enthaltene Adresse eingetragen. Anschließend wird HLD inkrementiert.

```
#S     ( ud --> 0 0 )
---------------------
```
Ruft solange # auf, bis als Quotient eine doppelt genaue Null entsteht. Die Erzeugung weiterer Zeichen wird dann abgebrochen, wenn führende Nullen entstehen würden.

```
SIGN     ( n ud --> ud )
-----------------------
```
Greift zerstörend auf die an dritter Position im Stack stehende Zahl n zu. Falls n negativ ist, wird ein Minuszeichen in die Ausgabezeichenkette eingefügt.

```
HOLD     ( c --> )
------------------
```
Fügt in die Ausgabezeichenkette das ASCII-Zeichen c ein.

```
#>     ( ud --> adr n )
-----------------------
```
Abschluß der Zahlen-Ausgabe-Konvertierung. #> erwartet das Überbleibsel der vorangegangenen #- und #S-Operationen ohne es auszuwerten und übergibt die Anfangsadresse der Zeichenkette sowie deren Länge, passend zu TYPE.

```
HLD     ( --> adr )
-------------------
```
Systemvariable, die die Anfangsadresse einer im Aufbau befindlichen Ziffern-Zeichenkette enthält.

8.8.2. Fertige Ausgabeoperationen

```
.     ( n --> )
```
Die Zahl n wird als vorzeichenbehaftete einfach genaue Zahl unter Verwendung der aktuellen Zahlenbasis an das Ausgabegerät ausgegeben.

```
.R     ( n1 n2 --> )
```
Die Zahl n1 wird als vorzeichenbehaftete einfach genaue Zahl unter Verwendung der aktuellen Zahlenbasis an das Ausgabegerät ausgegeben. Sie wird dabei rechtsbündig in ein Feld der Breite n2 eingetragen.

```
D.     ( d --> )
```
Die Zahl d wird als vorzeichenbehaftete doppelt genaue Zahl unter Verwendung der aktuellen Zahlenbasis an das Ausgabegerät ausgegeben.

```
D.R     ( d n --> )
```
Die Zahl d wird als vorzeichenbehaftete doppelt genaue Zahl unter Verwendung der aktuellen Zahlenbasis an das Ausgabegerät ausgegeben. Sie wird dabei rechtsbündig in ein Feld der Breite n eingetragen.

```
U.     ( u --> )
```
Die Zahl u wird als vorzeichenlose einfach genaue Zahl unter Verwendung der aktuellen Zahlenbasis an das Ausgabegerät ausgegeben. Dieses Wort gehört nicht zum fig-Standart.

```
?     ( adr --> )
```
Die auf adr vorgefundene Zahl wird als vorzeichenbehaftete einfach genaue Zahl unter Verwendung der aktuellen Zahlenbasis an das aktuelle Ausgabegerät ausgegeben.

8.9. Programmstruktuierung

8.9.1 Verzweigung

```
IF    ( f --> )        Laufzeit
      ( --> adr n )    Compile-Zeit
```
Wird verwendet in der Form
```
     IF ... ELSE ... ENDIF   oder
     IF ... ENDIF .
```
Ist das vorgefundene Flag f wahr, wird der Programmteil zwischen IF und ELSE bzw. zwischen IF und ENDIF ausgeführt, anderenfalls der Teil zwischen ELSE und ENDIF, der auch entfallen kann.

IF ist ein Immediate-Wort. Zur Compile-Zeit compiliert es 0BRANCH ins Wörterbuch und reserviert weitere 2 Bytes im Wörterbuch, in die später die Länge des IF-ELSE-Zweiges bzw. die des IF-ENDIF-Teils eingetragen wird. Die Adresse dieses Doppelbytes und eine Zahl, die der Fehlerkontrolle dient, werden übergeben.

```
ELSE     ( --> )                Laufzeit
         ( adr1 n --> adr2 n )  Compile-Zeit
-----------------------------------------
```

Wird verwendet in der Form

```
     IF ... ELSE ... ENDIF,
```

um Programmteile voneinander abzugrenzen, siehe IF.
ELSE ist ein Immediate-Wort. Zur Compile-Zeit wird anhand von n überprüft, ob ELSE ausschließlich mit IF kombiniert wurde. Dann wird in das von IF reservierte Doppelbyte die Länge des IF-ELSE-Zweiges eingetragen. Abschließend wird BRANCH ins Wörterbuch compiliert, ein Doppelbyte reserviert und die Adresse adr2 dieses Doppelbytes sowie die Zahl n wieder auf dem Stack übergeben.

```
ENDIF    ( --> )        Laufzeit
         ( adr n - -> ) Compile-Zeit
-------------------------------------
```

Wird verwendet in der Form

```
     IF ... ELSE ... ENDIF  oder
     IF ... ENDIF,
```

und bewirkt den korrekten Abschluß einer Programmverzweigung.
ENDIF ist ein Immediate-Wort. Zur Compile-Zeit wird überprüft, ob die Kombination ausschließlich mit IF oder ELSE erfolgte. Abschließend wird in das von IF oder ELSE reservierte Doppelbyte die Länge des vorangegangenen Programmzweiges eingetragen.

8.9.2. Schleifen mit unbestimmter Durchlaufzeit

```
BEGIN    ( --> )        Laufzeit
         ( --> adr n )  Compile-Zeit
-------------------------------------
```

Wird verwendet in der Form

```
     BEGIN ... UNTIL             oder
     BEGIN ... WHILE ... REPEAT  oder
     BEGIN ... AGAIN .
```

BEGIN markiert den Eintrittspunkt in eine Programmschleife.
BEGIN ist ein Immediate-Wort. Zur Compile-Zeit wird der Rückkehrpunkt an den Schleifenanfang (adr) sowie die Zahl n zur Fehlerkontrolle auf dem Stack hinterlassen, zur späteren Verwendung durch UNTIL, WHILE, REPEAT oder AGAIN.

```
UNTIL    ( f --> )      Laufzeit
         ( adr n --> )  Compile-Zeit
-------------------------------------
```

Wird verwendet in der Form

```
     BEGIN ... UNTIL .
```

Es wird das Flag f auf dem Stack erwartet. Ist dieses wahr, wird das Programm mit der auf UNTIL folgenden Anweisung fortgesetzt. Anderenfalls wird wieder zu dem durch BEGIN markierten Schleifenanfang zurückgekehrt.
UNTIL ist ein Immediate-Wort. Zur Compile-Zeit überprüft es, ob es

ausschließlich mit BEGIN kombiniert wurde. Anschließend compiliert es 0BRANCH und die negative Länge der BEGIN-UNTIL-Schleife ins Wörterbuch. Hierdurch wird später der Rücksprung an den Schleifenanfang realisiert.

```
WHILE     ( f --> )                          Laufzeit
          ( adr1 n1 --> adr1 n1 adr2 n2 )    Compile-Zeit
-----------------------------------------------------
```

Wird verwendet in der Form

```
     BEGIN ... WHILE ... REPEAT .
```

Ist das auf dem Stack erwartete Flag wahr, wird die Ausführung der Schleife mit dem WHILE-REPEAT-Teil fortgesetzt und dann zu BEGIN zurückgekehrt. Anderenfalls wird ein Sprung zum auf REPEAT folgenden Programm ausgeführt, d.h. die Schleife wird verlassen.
WHILE ist ein Immediate-Wort. Zur Compile-Zeit überprüft es, ob es ausschließlich mit BEGIN kombiniert wurde, compiliert 0BRANCH ins Wörterbuch, reserviert ein Doppelbyte und hinterläßt dessen Adresse adr2 sowie die der Fehlerkontrolle dienende Zahl n2 auf dem Stack.

```
REPEAT     ( --> )                    Laufzeit
           ( adr1 n1 adr2 n2 - -> )   Compile-Zeit
-------------------------------------------------
```

Wird verwendet in der Form

```
     BEGIN ... WHILE ... REPEAT .
```

Es bewirkt einen unbedingten Sprung zurück an den Schleifenanfang, der durch BEGIN markiert wurde.
REPEAT ist ein Immediate-Wort. Zur Compile-Zeit überprüft es, ob es ausschließlich mit WHILE kombiniert wurde. Anschließend werden BRANCH und die negative Länge der BEGIN-WHILE-REPEAT-Struktur ins Wörterbuch compiliert, um den Rückwärtssprung zum Schleifenanfang zu realisieren. Schließlich wird noch in das durch WHILE reservierte Doppelbyte auf Adresse adr2 die Länge des WHILE-REPEAT-Teiles eingetragen, um später den Sprung von WHILE hinter REPEAT zu realisieren.

```
AGAIN     ( --> )          Laufzeit
          ( adr n --> )    Compile-Zeit
-----------------------------------
```

Wird verwendet in der Form

```
     BEGIN ... AGAIN .
```

Es bewirkt einen unbedingten Sprung zu dem durch BEGIN markierten Schleifenanfang. Die BEGIN-AGAIN-Schleife ist eine Endlosschleife, die nur durch Manipulation des Returnstacks wieder verlassen werden kann.
AGAIN ist ein Immediate-Wort. Zur Compile-Zeit überprüft es anhand von n, ob die Kombination ausschließlich mit BEGIN erfolgte. Anschließend compiliert es BRANCH zusammen mit der negativen Länge der Schleife ins Wörterbuch, um später den Rückwärtssprung zu der durch BEGIN markierten Stelle zu realisieren.

8.9.3. Schleifen mit fester Durchlaufzahl

```
DO      ( n1 n2 --> )  Laufzeit
        ( --> adr n )  Compile-Zeit
-----------------------------------
```

Wird verwendet in der Form

```
     DO ... LOOP   oder
     DO ... +LOOP .
```

DO markiert den Beginn einer DO-LOOP-Schleife. Es erwartet auf dem Stack die beiden Zahlen n1 und n2. Diese werden lediglich zum Returnstack transportiert, und zwar so, daß n2 auf dem TOP des Returnstacks zu liegen kommt. Weitere Aktivitäten gibt es nicht. D.h. insbesondere auch, daß die Schleife unabhängig von n1 und n2 mindestens einmal durchlaufen wird. n1 ist der Grenzwert für den Schleifenindex, n2 der Startwert.
DO ist ein Immediate-Wort. Zur Compile-Zeit compiliert es (DO) ins Wörterbuch und hinterläßt mit adr und n einen Zeiger auf das (DO) folgende Wort, um den Schleifenanfang zu markieren, und eine Zahl, die der Fehlerkontrolle dient.

```
LOOP     ( --> )        Laufzeit
         ( adr n --> )  Compile-Zeit
-----------------------------------
```

Wird verwendet in der Form

```
     DO ... LOOP .
```

Der Schleifenindex auf dem Returnstack wird um 1 erhöht und danach (!) mit dem ebenfalls dort liegenden Endwert verglichen. Ist der Endwert erreicht, werden die Schleifenparameter vom Returnstack entfernt und das Programm hinter LOOP fortgesetzt. Anderenfalls wird die Schleife beginnend mit dem DO folgenden Wort noch einmal abgearbeitet.
LOOP ist ein Immedate-Wort. Zur Compile-Zeit überprüft es, ob es ausschließlich mit DO kombiniert wurde. Anschließend compiliert es (LOOP) zusammen mit der negativen Länge der DO-LOOP-Struktur ins Wörterbuch, um später den Rücksprung zum Schleifenanfang zu realisieren.

```
+LOOP     ( --> )        Laufzeit
          ( adr n --> )  Compile-Zeit
-----------------------------------
```

Wird verwendet in der Form

```
     DO ... +LOOP .
```

Eine Zahl n wird auf dem Stack erwartet. Diese Zahl wird zu dem auf dem Returnstack liegenden Schleifenindex addiert. Danach wird bei positiver Zahl n die Differenz Index-Endwert und bei negativem n die Differenz Endwert-Index gebildet. Ist diese Differenz negativ, wird die Schleife noch einmal abgearbeitet. Anderenfalls wird die Schleife wie bei LOOP verlassen.
+LOOP ist ein Immedate-Wort. Zur Compile-Zeit prüft es, ob es ausschließlich mit DO kombiniert wurde. Anschließend compiliert es (+LOOP) zusammen mit der negativen Länge der DO-+LOOP-Struktur ins Wörterbuch, um später den Rücksprung zum Schleifenanfang zu realisieren.

LEAVE (-->)

Setzt den Endwert des Schleifenindex auf dem Returnstack gleich dem aktuellen Index. Auf diese Weise wird danach beim Erreichen von LOOP oder +LOOP die Schleife garantiert verlassen. Der aktuelle Schleifenindex wird durch LEAVE nicht beeinflußt.
Achtung ! LEAVE darf nur innerhalb von DO-Schleifen verwendet werden.

I (--> n)

I wird innerhalb von DO-Schleifen aufgerufen, um den Schleifenindex auf den Datenstack zu bringen. Da der Index auf dem Top des Returnstack liegt, hat I die gleiche Funktion wie R.

8.9.4. Laufzeitaktivitäten der Strukturwörter

BRANCH (-->)

Bewirkt einen unbedingten Sprung um die unmittelbar hinter BRANCH ins Wörterbuch hineincompilierte Distanz. Diese kann positiv (Vorwärtssprung) oder negativ (Rückwärtssprung) sein.

0BRANCH (f -->)

Bewirkt einen bedingten Sprung, der dann erfolgt, wenn das Flag f falsch ist. Unmittelbar hinter 0BRANCH wird die Sprungdistanz analog zu BRANCH ins Wörterbuch compiliert.

(DO) (n1 n2 -->)

(DO) ist die Laufzeitaktivität von DO, die von DO compiliert wird. n1 wird als Endwert für den Schleifenindex und n2 als Startwert zum Returnstack transportiert (siehe DO).

(LOOP) (-->)

Laufzeitaktivität von LOOP, die von LOOP compiliert wird. (LOOP) inkrementiert den Schleifenindex und testet auf Erreichen der Abbruchbedingung, siehe LOOP. Hinter (LOOP) ist die Sprungdistanz zum Schleifenanfang compiliert, die der negativen Schleifenlänge entspricht.

(+LOOP) (n -->)

Laufzeitaktivität, die von +LOOP compiliert wird. Der Schleifenindex wird um n inkrementiert. Zum Test der Abbruchbedingung siehe +LOOP. Analog zu (LOOP) steht hinter (+LOOP) wieder die Sprungdistanz zum Schleifenanfang.

```
BACK     ( adr --> )
--------------------
```
BACK ist ein Hilfswort für den Compiler. Es errechnet die Distanz von HERE zu der auf dem Stack erwarteten Adresse und compiliert die Distanz in die nächste freie Stelle des Wörterbuches. Wird verwendet von REPEAT, UNTIL, AGAIN, LOOP und +LOOP.

8.10. Verwaltung der Screens

8.10.1. Systemkonstanten und –variablen

```
B/BUF     ( --> n )
-------------------
```
Systemkonstante, die die Länge eines Textbuffers in Byte enthält. Hier ist B/BUF=512. Üblich sind bei anderen FORTH-Installationen Werte zwischen 128 und 1024.

```
B/SCR     ( --> n )
-------------------
```
Systemkonstante, die die Anzahl der Textpuffer enthält, die einen Screen bilden. Hier ist B/SCR=1, üblich sind je nach B/BUF Werte bis 8.

```
C/L     ( --> n )
-----------------
```
Systemkonstante, die die Anzahl der Zeichen in einer Screen-Zeile enthält. Hier ist C/L=32, üblich ist allerdings eine Länge von 64 Zeichen bei anderen FORTH-Installationen.

```
FIRST     ( --> adr )
---------------------
```
Systemvariable, die die Anfangsadresse des Textpuffer-Speicherbereiches enthält.
Im fig-Standard ist FIRST eine Konstante. Davon wurde hier abgewichen, um den Speicherbereich für die Textpuffer dynamisch ändern zu können.

```
LIMIT     ( --> adr )
---------------------
```
Systemvariable, die die Endadresse des Textpuffer-Speicherbereichs enthält.
Im fig-Standard ist LIMIT eine Konstante (siehe FIRST).

```
IOTAB     ( --> adr )
---------------------
```
Systemvariable, die im niederwertigen Byte die UP-Nr. für die Zeichen-Ausgabe und im höherwertigen Byte die UP-Nr. für die Zeichen-Eingabe enthält.

8.10.2. Verwaltung des Textpuffer-Speicherbereichs

#BUFF (--> n)

Übergibt die Anzahl n der zwischen FIRST und LIMIT zur Verfügung stehenden Textpuffer.

EMPTY-BUFFERS (-->)

Füllt den gesamten Speicherbereich zwischen FIRST und LIMIT mit Nullen.

BLOCK (n --> adr)

Übergibt die Adresse, auf der die Daten des Textpuffers n zur Verfügung stehen.
Das im fig-Standard definierte Wort BLOCK führt bei Bedarf eine Datentransfer mit der Diskette durch. Da beim Kleincomputer alle Textpuffer ständig im Speicher zur Verfügung stehen, ist hier kein Datentransfer notwendig. In der Wirkung ist BLOCK kompatibel zum fig-Standard.

INIT-BUFFERS (-->)

An das Ende eines jeden Blocks werden die beiden Null-Bytes zur Markierung des Blockendes geschrieben.

8.10.3. Verwaltung von Screens und Zeilen

(LINE) (n1 n2 --> adr n3)

Für die Zeile n1 im Screen n2 wird die zuständige Adresse adr im Textpuffer errechnet. Zusätzlich wird die Zeilenlänge n3 übergeben, die der Konstanten C/L entspricht.

.LINE (n1 n2 -->)

Die Zeile n1 im Screen n2 wird an das Ausgabegerät ausgegeben. Leerzeichen am Zeilenende werden unterdrückt.

LIST (n -->)

Der Screen n wird an das Ausgabegerät ausgegeben. n wird in die Systemvariable SCR eingeschrieben, die Zahlenbasis wird auf 10 gestellt.

INDEX (n1 n2 -->)

Die Kopfzeilen der Screens n1 bis n2 werden an das Ausgabegerät ausgegeben. Damit kann man sich schnell einen Überblick über den Inhalt des Textspeicherbereiches verschaffen.

```
TRIAD     ( n --> )
-------------------
```
Es werden drei aufeinanderfolgende Screens an das Ausgabegerät ausgegeben. Der erste dieser drei Screens ist der mit der von n aus nächstkleineren durch drei teilbaren Nummer. Abschließend wird die Meldung Nr. 15 ausgegeben.

8.11. Speichern auf Kassette und Laden von Kassette

```
REC/BLK     ( --> n )
---------------------
```
Konstante, die die Anzahl von Magnetband-Datenblöcken pro Textpuffer enthält. Hier ist REC/BLK=4.
Dieses Wort ist im fig-Standard nicht enthalten.

```
REC     ( --> adr )
-------------------
```
Systemvariable, die die Nummer des gerade bearbeiteten Magnetband-Datenblocks enthält.
Dieses Wort ist nicht im fig-Standard enthalten.

```
R/W
-------------------
```
Führt den Datenaustausch mit dem Magnetband durch. Je nach übergebenen Parametern wird eine von 6 Funktionen ausgeführt. Die Adresse adr ist stets die Adresse des Pufferspeichers, der in den entsprechenden Datenblock auf das Magnetband geschrieben werden soll, bzw. in den der Magnetbandblock eingelesen werden soll.

1. (adr 0 -->)

   Schreiben des ersten Blockes einer Datei. Dieser Block enthält üblicherweise den Dateinamen.

2. (adr 1 -->)

   Schreiben des fortlaufend nächsten Blockes einer Datei.

3. (adr 2 -->)

   Schreiben des letzten Blockes mit der Blocknummer 255.

4. (adr 3 --> n f)

   Lesen eines Blockes einer Datei mit vorheriger Initialisierung der Magnetbandeingabe im Computer. Erst nach der Initialisierung ist das Einlesen von Blöcken möglich. Es wird die Nummer n des gelesenen Blockes und ein Fehlerflag f übergeben. f=1 heißt, daß der Block fehlerhaft gelesen wurde. Bei f=0 ist der Block in Ordnung.

5. (adr 4 --> n f)

   Lesen eines Blockes einer Magnetbanddatei ohne Initialisierung der Magnetbandeingabe. Parameter wie 4.

6. (5 -->)

Abschluß der Magnetbandeingabe. Eine freie Adresse wird hier nicht benötigt. Der Computer wird wieder auf Normalbetrieb umgeschaltet.

R/W ist nicht zu dem R/W des fig-Standards kompatibel. Dort führt R/W den Datenaustausch mit der Diskette durch.

CSAVE (n1 n2 -->)

Schreiben einer kompletten Datei auf Magnetband. Die Datei enthält die Screens n1 bis einschließlich n2. CSAVE holt das nächste Wort aus dem aktiven Eingabe-Puffer und interpretiert es bis zu einer Länge von 8 Zeichen als Dateinamen. An den Namen werden die Zeichen "(F)" als Dateityp angefügt.
CSAVE realisiert keine Bildschirmausschriften.
Beispiel: 1 2 CSAVE MESSAGES
Die Screens 1 und 2 werden unter dem Namen MESSAGES(F) auf Magnetband geschrieben.
CSAVE ist nicht im fig-Standard enthalten.

VERIFY (-->)

Aufruf des CAOS-Systemprogramms VERIFY. Siehe Bedienungsanleitung des Computers.
VERIFY ist nicht im fig-Standard enthalten.

CLOAD (n1 n2 -->)

Einlesen einer Datei vom Magnetband in die Screens von n1 bis maximal n2. Hat die Datei in den angegebenen Screens keinen Platz, wird nach Screen n2 mit der Meldung MSG#6 (memory range) abgebrochen. Die gelesenen Blöcke werden auf dem Bildschirm angezeigt. Falls der gleiche Block dreimal hintereinander fehlerhaft eingelesen wird, erfolgt die Meldung MSG#8 (tape error), und das Einlesen wird mit dem nächsten Block fortgesetzt. Die Systemvariable OFFSET enthält die Anzahl der Screens, die vom Dateianfang an ignoriert werden. Siehe auch Abschnitt 4.3 .
CLOAD ist nicht im fig-Standard enthalten.

OFFSET (--> adr)

Systemvariable, enthält die Anzahl der Screens, die durch CLOAD am Anfang einer Datei unterdrückt werden.
Im fig-Standard hat OFFSET eine andere Bedeutung.

8.12. Textverarbeitung und Interpretation

8.12.1. Suche im Wörterbuch und im Eingabepuffer

```
ENCLOSE     ( adr c --> adr n1 n2 n3 )
```

Durchsucht einen Text beginnend ab Adresse adr nach Zeichen, die ungleich dem Zeichen c sind. Mit ENCLOSE werden aus einem Text einzelne Worte isoliert, die durch die Begrenzungszeichen c (meistens Leerzeichen) voneinander getrennt sind. Die Adresse wird wieder übergeben.
Adr+n1 zeigt auf das erste Zeichen, das verschieden von c ist.
Adr+n2 zeigt hinter das erkannte Textstück, also auf ein Zeichen, das wieder gleich c ist.
Adr+n3 zeigt auf die Stelle, wo ENCLOSE zum nächsten Mal aufzurufen ist. Das ist die Adresse adr+n2n+1.
Findet ENCLOSE hinter einem Textstück ein NULL-Zeichen, so ist n3=n2.
Findet ENCLOSE kein Textstück, sondern lediglich ein NULL-Zeichen, so ist n3=n2-1.

```
WORD     ( c --> )
```

Liest das nächste Textelement aus dem Eingabepuffer, der gerade interpretiert wird. bei BLK=0 ist das der Terminal-Input-Buffer, ansonsten der Textpuffer, dessen Nummer in BLK steht.
Der Text, der durch das Zeichen c begrenzt wird (meistens Leerzeichen) wird nach HERE transportiert und liegt dort im FORTH-Standard-Format, d.h. beginnend mit einem Count-Byte, vor. Dort ist das Textelement mit mindestens zwei Leerzeichen abgeschlossen.

```
(FIND)     ( adr1 adr2 --> adr3 b tf )  falls vorhanden
           ( adr1 adr2 --> tf )         falls nicht gefunden
```

In Adresse adr1 steht eine Zeichenkette im FORTH-Standard-Format. Beginnend bei der Namensfeldadresse adr2 wird das Wörterbuch nach einem der Zeichenkette entsprechenden Wortnamen durchsucht. Bei erfolgloser Suche wird ein false-Flag übergeben, andernfalls die Parameterfeldadresse adr3 des gefundenen Wortes, dessen Count-Byte b und ein true-Flag.

```
-FIND     ( --> adr b tf )  falls gefunden
          ( -->       tf )  falls nicht gefunden
```

Holt mit Word das nächste Textstück aus dem aktiven Eingabepuffer. Dann wird mit (FIND) das Context-Vokabular nach einem entsprechenden Worteintrag durchsucht. War die Suche hier erfolglos, wird das Current-Vokabular abgesucht. Bei insgesamt erfolgloser Suche wird ein false-Flag übergeben, anderenfalls die Parameterfeldadresse adr des gefundenen Wortes, dessen Count-Byte und ein true-Flag.

```
BLK     ( --> adr )
------------------
```
Systemvariable, die die Nummer des gerade aktiven Eingabepuffers hält. Bei BLK=0 ist dies der Terminal-Input-Buffer.

```
IN      ( --> adr )
-----------------
```
Systemvariable, die die Anzahl der bereits verarbeiteten Zeichen des gerade aktiven Eingabepuffers enthält. IN wird von WORD verwaltet.

8.12.2. Konvertierung von Zeichenketten in Binärzahlen

```
DIGIT     ( c n1 --> n2 tf )  falls gültig
          ( c n1 --> n2 tf )  falls ungültig
-------------------------------------------
```
Umwandlung des ASCII-Zeichens c in eine Binärzahl. Dabei wird anhand der Zahlenbasis n1 geprüft, ob c ein im vereinbarten Zahlensystem gültiges Ziffernsymbol ist. Ist dies der Fall, wird das binäre Zahlenäquivalent zu c und ein true-Flag, anderenfalls ein false-Flag übergeben.

```
(NUMBER)     ( d1 adr1 --> d2 adr2 )
-----------------------------------
```
Umwandlung einer in adr1+1 beginnenden Ziffern-Zeichenkette. Die Zahl d1 wird fortlaufend mit der Zahlenbasis multipliziert und anschließend die nächste durch DIGIT umgewandelte Ziffer addiert. Das wird solange fortgesetzt, bis ein ungültiges Ziffernsymbol angetroffen wird. d2 ist die auf diese Weise veränderte Zahl d1 und adr2 ist die Adresse des ersten ungültigen Ziffernsymbols.

```
NUMBER     ( adr --> d )
------------------------
```
Umwandlung einer Ziffernzeichenkette in eine doppelt genaue Zahl. Auf adr wird eine Zeichenkette im FORTH-Standard-Format erwartet. Das Count-Byte wird aber nicht verwendet, sondern nur berücksichtigt. Am Ende der Zeichenkette muß mindestens ein Leerzeichen vorhanden sein. Falls die Zeichenkette einen Dezimalpunkt enthält, wird dessen Position an DPL übergeben.
NUMBER führt die Konvertierung unter Anwendung der aktuellen Zahlenbasis durch.
Falls in der Zeichenkette ein ungültiges Ziffernsymbol angetroffen wird, erzeugt NUMBER die Fehlermeldung 0.

```
DPL     ( --> adr )
-------------------
```
Systemvariable, die die Position eines Dezimalpunktes in einer durch NUMBER konvertierten Ziffern-Zeichenkette enthält. DPL gibt an, wieviele Stellen der Dezimalpunkt vom rechten Ende der Zahl entfernt ist. Bei DPL=-1 enthielt die konvertierte Zeichenkette keinen Dezimalpunkt.
DPL wird nicht vom System ausgewertet.

8.12.3. Textinterpretation

EXECUTE (adr -->)

Erwartet auf dem Stack die Codefeldadresse eines Wortes und führt dieses dann aus.

INTERPRET (-->)

Der Text aus dem aktiven Eingabe-Puffer wird interpretiert. Je nach Systemzustand (Variable STATE) wird entweder ausgeführt oder compiliert. Näheres siehe Abschnitt 6.4 .

LOAD (n -->)

Der Inhalt des Screens n wird durch INTERPRET interpretiert. Die Interpretation wird entweder am Screenende oder an der Stelle, wo ;S angetroffen wird, beendet.

--> (-->)

Darf nur innerhalb von Screens verwendet werden. --> bewirkt, daß die mit LOAD gestartete Textinterpretation mit dem nächsten Screen fortgesetzt wird.

% (-->)

% steht für ein Wort, dessen Name lediglich aus einem NULL-Zeichen besteht (ASCII-Code 0). Trifft INTERPRET auf dieses Wort, beendet es augenblicklich die Interpretation des aktiven Eingabepuffers. Alle Textpuffer müssen an ihrem Ende eine NULL stehen haben.

((-->)

Kennzeichnet Kommentar in einem Textpuffer. Der auf (folgende Text bis zu einem) wird ignoriert.

QUIT (-->)

Setzt BLK auf 0, initialisiert den Returnstack mit dem Inhalt von R0 und schaltet den EXECUTE-Mode ein (STATE=0). Dadurch ist ein Grundzustand erreicht, in dem Texteingaben von der Tastatur angefordert werden können.
Anschließend wird abwechselnd eine Eingabezeile von der Tastatur angefordert und diese dann interpretiert. Die wesentlichen Aktionen stehen in einer Endlosschleife mit QUERY INTERPRET.

8.13. Definition neuer Worte

8.13.1. Definitionsworte

CREATE (-->)

Wird verwendet in der Form

```
CREATE XXX
```

Ein Wortkopf mit dem Namen XXX wird erstellt. Das Wort XXX darf noch nicht aufgerufen werden, da der Rumpf noch leer ist. Deshalb ist das Smudge-Bit noch gesetzt. Die Codefeldadresse zeigt nach HERE, so daß XXX als Primitive vorbereitet ist und nur noch der Maschinencode eingetragen werden muß.

CONSTANT (n -->)

Wird verwendet in der Form

```
n CONSTANT XXX
```

Erstellt ein Wort mit dem Namen XXX und verleiht ihm die Eigenschaft einer Konstanten, d.h. wenn XXX später aufgerufen wird, bringt es die Zahl n zum TOS.

VARIABLE (n -->)

Wird verwendet in der Form

```
n VARIABLE XXX
```

Erstellt ein Wort mit dem Namen XXX und verleiht ihm die Eigenschaft einer Variablen, d.h. wenn XXX später aufgerufen wird, bringt es die Adresse zum Stack, auf der der Wert der Variablen zu erreichen ist. VARIABLE initialisiert die neu definierte Variable mit dem Wert n.

USER (n -->)

Wird verwendet in der Form

```
n USER XXX
```

USER definiert eine Variable im User-Feld. Später verhält sich XXX genauso, wie eine "normale" Variable, lediglich die Speicheradresse, auf der der Wert erreicht werden kann, liegt im User-Feld. n ist der Byte-Offset zum Anfang des User-Feldes. Der Anwender muß sich über die Belegung des User-Feldes im Klaren sein.

: (-->)

Leitet die Definition eines Secondary-Wortes ein. Der Doppelpunkt wird verwendet in der Form

```
: XXX ... ;
```

Ein Wortkopf mit dem Namen XXX wird erstellt und das System in den Compile-Mode geschaltet. Damit werden vom Textinterpreter die nun folgenden Worte nicht mehr direkt ausgeführt, sondern in den Rumpf von XXX eingebaut und gelangen dann zur Ausführung, wenn XXX später aufgerufen wird. Mit ; wird der Compile-Mode wieder verlassen und die gerade aufgebaute Wortdefinition beendet.

8.13.2. Abschluß von Doppelpunkt-Definitionen

; (-->)

Das ist ein Immediate-Wort. Das Wort ;S wird an das Ende der gerade aufgebauten Wortdefinition compiliert, das Smudge-Bit rücksetzt und das System wieder in den EXECUTE-Mode geschaltet.

;S (-->)

Ist das letzte Wort jeder Doppelpunktdefinition, das von ; compiliert wird. ;S holt die Adresse der übergeordneten Programmebene vom Returnstack und schreibt diese in den Instruction-Pointer ein. Als Instruction-Pointer wird das BC-Register verwendet. Damit wird die Programmabarbeitung in der übergeordneten Ebene fortgesetzt.

8.13.3. Definition spezieller Laufzeitaktivitäten

;CODE (-->)

Beendet den ersten Teil einer Definitionswortdefinition, in dem ein neuer Wortkopf und das zugehörige Parameterfeld aufgebaut werden müssen. ;CODE ist immediate und compiliert (;CODE) an das Ende dieses ersten Teils. Nach ;CODE wird in jedem Fall noch eine Maschinencodesequenz als Laufzeitaktivität der neu definierten Worte erwartet. siehe Abschnitt 6.8 .

(;CODE) (-->)

Wird von ;CODE an das Ende des ersten Definitionsteils einer Definitionswortdefinition compiliert. Sorgt dafür, daß die Codefeldadresse des neu definierten Wortes auf den hinter (;CODE) compilierten Maschinencode zeigt, welcher bei Aufruf dieses Wortes zur Ausführung gelangt.

<BUILDS ... DOES>

Wird verwendet in der Form

```
: XXX <BUILDS ... Builds-Teil ... DOES> ... Does-Teil ... ;
```

Ein Definitionswort mit dem Namen XXX wird erstellt. XXX wird nun wiederum in der Form XXX YYY zur Definition von YYY verwendet. Bei der Definition von YYY gelangt der Builds-Teil zur Ausführung, wobei zweckmäßigerweise Eintragungen in das Parameterfeld von YYY gemacht werden. Wird YYY später aufgerufen, dann gelangt der Does-Teil zur Ausführung, wobei die Adresse des Parameterfeldes von YYY bereits auf dem TOS liegt.

8.13.4. Sonstiges

SMUDGE (-->)

Schaltet das Smudge-Bit des zuletzt definierten Wortes um, so daß dieses je nach vorherigem Zustand entweder für gültig oder für ungültig erklärt wird.

IMMEDIATE (-->)

Deklariert das zuletzt definierte Wort als "immediate". Dadurch wird dieses Wort auch dann durch den Textinterpreter ausgeführt, wenn sich das System im Compile-Mode befindet.

WIDTH (--> adr)

Systemvariable, die festlegt, bis zu welcher Länge der Wortname im Wortkopf gespeichert wird. Das Count-Byte bleibt hiervon unberührt. Ist WIDTH beispielsweise 3, kann trotzdem noch zwischen ZAUN und ZAUBER unterschieden werden. Normalerweise hat WIDTH den Wert 31. Eine Verringerung kann den Platzbedarf neuer Definitionen reduzieren.

8.14. Compiler

8.14.1. Verwaltung des Systems

DP (--> adr)

Systemvariable, die die Adresse der ersten freien Zelle im Wörterbuch enthält. (DP – dictionary Pointer)

STATE (--> adr)

Systemvariable, die Auskunft über die Systemzustände Compile-Mode und EXECUTE-Mode gibt.

```
    STATE  gleich   0 -- Execute-Mode
    STATE  ungleich 0 -- Compile-Mode
```

CSP (--> adr)

Systemvariable, die der Fehlerkontrolle bei der Programmstrukturierung innerhalb von Doppelpunktdefinitionen dient. In CSP wird zu Beginn der Definition der augenblickliche Stackpointerstand eingeschrieben.
CSP - current stack pointer

!CSP (-->)

Schreibt die aktuelle Adresse des TOS in die Systemvariable CSP ein.

```
 ((     ( --> )
--------------
```
Immediate-Wort, mit dem während einer Doppelpunktdefinition der Compile-Mode vorübergehend verlassen werden kann, etwa um eine Zwischenrechnung auf dem Stack auszuführen, deren Ergebnis später mit LITERAL compiliert werden kann.
ACHTUNG ! Im fig-Standard hat dieses Wort den Namen [.

```
))     ( --> )
--------------
```
Bewirkt das Einschalten des Compile-Modes und wird zweckmäßigerweise nach ((verwendet.
ACHTUNG ! Im fig-Standard hat dieses Wort den Namen].

8.14.2. Compilieren ins Wörterbuch

```
HERE     ( --> adr )
--------------------
```
Legt die Adresse der ersten freien Zelle des Wörterbuches auf den Stack.

```
ALLOT     ( u --> )
-------------------
```
Reserviert einen freien Bereich der Länge u Bytes im Wörterbuch, d.h. DP wird um u erhöht.

```
,     ( n --> )
---------------
```
Compiliert die Zahl n auf die Adresse HERE ins Wörterbuch. Danach zeigt HERE hinter n auf die nächste freie Zelle.

```
C,     ( b --> )
----------------
```
Compiliert das Byte b in die Adresse HERE. DP wird um 1 erhöht.

```
COMPILE     ( --> )
-------------------
```
Wird in Wortdefinitionen eingebaut in der Form
```
     : XXX ... COMPILE YYY ... ; .
```
Wenn XXX später aufgerufen wird, bewirkt Compile, daß die Codefeldadresse von YYY ins Wörterbuch compiliert wird.

```
(COMPILE)     ( --> )
---------------------
```
Immediate-Wort, das den Einbau von Immediate-Worten in Wortdefinitionen gestattet. (COMPILE) wird verwendet in der Form
```
          : XXX ... (COMPILE) YYY ... ; .
```
Das Immediate-Wort YYY wird nicht ausgeführt, sondern in die Definition XXX eingebaut.
ACHTUNG ! Im fig-Standard hat dieses Wort den Namen [COMPILE].

```
LITERAL     ( n --> )   im Compile-Mode
              ( --> )   im Execute-Mode
---------------------------------------
```
Immediate-Wort, das nur im Compile-Mode eine Wirkung hat. LITERAL compiliert zunächst LIT ins Wörterbuch und anschließend die Zahl n (siehe LIT).

```
DLITERAL     ( d --> )  im Compile-Mode
               ( --> )  im Execute-Mode
---------------------------------------
```
Immediate-Wort, das nur im Compile-Mode eine Wirkung hat. DLITERAL ruft zweimal LITERAL auf und compiliert dadurch den höherwertigen und den niederwertigen Teil der doppelt genauen Zahl d getrennt ins Wörterbuch.

```
LIT     ( --> n )
------------------
```
Transportiert die unmittelbar hinter LIT ins Wörterbuch compilierte Zahl n zum Stack.

8.15. Vokabulare

8.15.1. Variablen zur Verwaltung der Vokabulare

```
CONTEXT     ( --> adr )
-----------------------
```
Systemvariable, die einen Adresszeiger in das als Context erklärte Vokabular-Namenswort enthält.

```
CURRENT     ( --> adr )
-----------------------
```
Systemvariable, die einen Adresszeiger in das als Current erklärte Vokabular-Namenswort enthält. Nach Aktivieren des Moduls ist FORTH zum Current-Vokabular deklariert.

```
VOC-LINK     ( --> adr )
------------------------
```
Systemvariable, die einen Adresszeiger in das zuletzt definierte Vokabular-Namenswort enthält.

8.15.2. Vokabular-Arbeit

```
VOCABULARY     ( --> )
----------------------
```
Definitionswort für ein neues Vokabular-Namenswort. VOCABULARY wird verwendet in der Form

```
     VOCABULARY XXX .
```
Ein (im Current-Vokabular definiertes) Vokabular-Namenswort XXX wird erstellt. Wenn XXX später aufgerufen wird, erklärt es sich selbst zum Context-Vokabular.

```
DEFINITIONS     ( --> )
-----------------------
```
Setzt CURRENT gleich CONTEXT, so daß beispielsweise nach
```
     VOCABULARY XXX XXX DEFINITIONS
```
dem Vokabular XXX Wortdefinitionen angefügt werden können.

```
LATEST     ( --> adr )
----------------------
```
Legt die Namensfeldadresse des zuletzt im Current-Vokabular definierten Wortes auf den Stack.

```
VLIST     ( --> )
-----------------
```
Gibt die Namen aller Worte beginnend im Context-Vokabular an das Ausgabegerät aus. Auch Wortnamen, in denen das Smudge-Bit gesetzt ist, werden mit ausgegeben. Durch Druck auf eine beliebige Taste kann VLIST vorzeitig abgebrochen werden.

8.15.3. Bereits definierte Vokabulare

```
FORTH     ( --> )
-----------------
```
Das FORTH-Vokabular wird zum Context-Vokabular erklärt.

```
EDITOR     ( --> )
------------------
```
Das EDITOR-Vokabular wird zum Context-Vokabular erklärt.

8.16. Mitteilungen und Fehler

8.16.1. Ausgabe von Mitteilungen und Reaktionen auf Fehler

```
WARNING     ( --> adr )
-----------------------
```
Systemvariable,
WARNING=1 bewirkt, daß Mitteilungen den dafür reservierten Screens entnommen werden.
WARNING=0 bewirkt, daß lediglich die Nummer der Mitteilung ausgegeben wird.
WARNING<0 bewirkt, daß durch ERROR (ABORT) aufgerufen wird.

```
MESSAGE     ( n --> )
--------------------
```
Gibt die Mitteilung Nr. n aus. Wenn WARNING ungleich 0 ist, wird die Zeile n des Screens 1 ausgegeben. Die Numerierung wird auf den folgenden Screens fortgesetzt, siehe 7.1 .

```
ERROR     ( n1 --> n2 n3 )  falls BLK ungleich 0
          ( n1 --> )        falls BLK   gleich 0
-------------------------------------------------
```
Gibt Fehlermeldungen aus.
Ist WARNING<0 wird (ABORT) ausgeführt. Ansonsten gibt MESSAGE die Mitteilung n1 aus (abhängig von WARNING). Anschließend wird der Datenstack geleert und, falls der TIB nicht der aktuelle Eingabepuffer war, n2 als Inhalt von IN und n3 als Inhalt von BLK auf dem Stack hinterlassen. Dies ist günstig für eine schnelle Fehlerlokalisation. Abschließend wird mit QUIT wieder der Textinterpreter aufgerufen.

```
?ERROR     ( f n --> )
----------------------
```
Die Fehlermeldung n wird ausgegeben, wenn f wahr ist, siehe ERROR.

```
ABORT     ( --> )
-----------------
```
Initialisiert beide Stacks mit den in S0 und R0 gehaltenen Positionen, gibt die Systemidentifikation aus und ruft den Textinterpreter QUIT auf.

```
(ABORT)     ( --> )
-------------------
```
Wird im Anschluß an eine Fehlermeldung ausgeführt, wenn WARNING<0 ist. (ABORT) ruft selbst wieder ABORT auf.

8.16.2. Erkennen von speziellen Fehlersituationen

```
?COMP      ( --> )
------------------
```
Gibt die Fehlermeldung MSG#17 aus, wenn sich das System nicht im Compile-Mode befindet.
Text: compilation only, use in definition

```
?CSP     ( --> )
----------------
```
Gibt die Fehlermeldung MSG#20 aus, wenn die aktuelle TOS-Position von der in der Systemvariablen CSP abweicht.
Text: definition not finished

```
?EXEC     ( --> )
-----------------
```
Gibt die Fehlermeldung MSG#18 aus, wenn sich das System nicht im Execute-Mode befindet.
Text: execution only

```
?LOADING     ( --> )
--------------------
```
Gibt die Fehlermeldung MSG#22 aus, falls nicht mit LOAD geladen wird.
Text: use only when loading

```
?PAIRS     ( n1 n2 --> )
------------------------
Gibt  die Fehlermeldung MSG#19 aus,  wenn n1 und n2 ungleich  sind.
?PAIRS wird durch die Strukturworte verwendet, siehe 8.9 .
Text: conditionals not paired


?STACK     ( --> )
-------------------
Installationsabhängiges Wort,  mit dem überprüft wird,  ob sich der
Stack noch in dem vorgesehenen Bereich befindet,  wenn nicht   dann
wird eine Fehlermeldung ausgegeben.
Text: empty stack  ( falls Stackunterlauf eingetreten ist ) MSG#1
      full stack   ( bei Stacküberlauf )                    MSG#7
```

8.17. Der Editor

```
8.17.1. Hilfswörter
-------------------

SCR     ( --> adr )
-------------------
Systemvariable,  die  die  Nummer des gerade  bearbeiteten  Screens
enthält.


PAD     ( --> adr )
Stellt  die Adresse eines Zwischenspeichers im  freien  Wörterbuch-
bereich, der 68 Bytes über HERE liegt, zur Verfügung.


TEXT     ( c --> )
------------------
Holt  aus dem aktiven Eingabepuffer den nächsten mit dem Zeichen  c
begrenzten Text ab und hinterlegt ihn auf PAD.


LINE     ( n --> adr )
----------------------
Übergibt  die  Textpuffer-Adresse  adr der Zeile  n  des  aktuellen
Screens.


-MOVE     ( adr n --> )
-----------------------
Schafft  die auf Adresse adr stehende Zeile in den Platz der  Zeile
n des aktuellen Screens im Textpuffer.


8.17.2. Manipulation mit Zeilen
-------------------------------

R     ( n --> )
---------------
Die  auf  PAD  bereit  gehaltene  Zeile wird in  die  Zeile  n  des
aktuellen Screens eingeschrieben.
R -- replace
```

```
P     ( n --> )
---------------
```
Der nachfolgend eingegebene Text wird in die Zeile n des aktuellen Screens eingeschrieben.
P -- put

```
H     ( n --> )
---------------
```
Die Zeile n des aktuellen Screens wird in den Zwischenspeicher PAD transportiert.
H -- hold

```
T     ( n --> )
---------------
```
Die Zeile n des aktuellen Screens wird an das Ausgabegerät ausgegeben.
T -- type

```
E     ( n --> )
---------------
```
Die Zeile n des aktuellen Screens wird mit Leerzeichen gefüllt.
E -- erase

```
S     ( n --> )
---------------
```
Die Zeile n des aktuellen Screens wird freigemacht, indem alle folgenden Zeilen nach unten verschoben werden. Zeile 15 geht dabei verloren.
S -- spread

```
D     ( n --> )
---------------
```
Die Zeile n des aktuellen Screens wird nach PAD transportiert und dann im Screen gelöscht, indem alle folgenden Zeilen nach oben verschoben werden.
D -- delete

8.17.3. Manipulation mit Screens

```
L     ( --> )
-------------
```
Der aktuelle Screen wird an das Ausgabegerät ausgegeben.

```
CLEAR     ( n --> )
-------------------
```
Der Screen n wird mit Leerzeichen überschrieben und zum aktuellen Screen erklärt.

```
MOVE     ( n1 n2 --> )
----------------------
```
Der Screen n1 wird in den Screen n2 kopiert.

```
EDIT     ( n --> )
------------------
```
Aufruf des Edit-Modus für den Screen n (siehe 4.6).

8.18. Sonstige nützliche FORTH-Worte

8.18.1. Konstanten

```
0 1 2 3     ( --> n )
---------------------
```
Wegen ihrer häufigen Verwendung sind 0, 1, 2 und 3 bereits als Konstanten definiert. Man sollte sich daher nicht wundern, wenn der Textinterpreter die 2 und die 3 widerspruchslos "schluckt", obwohl die Zahlenbasis 2 ist.

```
BL     ( --> c )
----------------
```
Konstante, die den ASCII-Code des Leerzeichens übergibt.

```
NEXT     ( --> c )
------------------
```
Konstante, die den Eintrittspunkt in den Adressinterpreter übergibt.

8.18.2. Systemvariablen

```
S0     ( --> adr )
------------------
```
Enthält die Initialisierungsposition des Datenstacks.

```
R0     ( --> adr )
------------------
```
Enthält die Initialisierungsposition des Returnstacks.

```
FENCE     ( --> adr )
---------------------
```
Enthält eine Adresse, unterhalb der das "Vergessen" von Wortdefinitionen nicht mehr möglich ist.

```
FLD     ( --> adr )
-------------------
```
Wird zur Zeit noch nicht vom FORTH-System verwendet.

```
R#     ( --> adr )
------------------
```
Wird zur Zeit noch nicht vom FORTH-System verwendet.

8.18.3. Analyse von fertigen Wortdefinitionen

```
TRAVERSE     ( adr1 n --> adr2 )
```

Sucht das andere Ende des Wortnamens in einer Worteintragung. Die Suche beginnt bei adr1. Bei n=1 erfolgt die Suche in Richtung steigender, bei n=-1 in Richtung fallender Adressen.
Adresse adr1 kann Anfang oder Ende der Namens-Zeichenkette sein. Auf Adresse adr2 steht entweder das Count-Byte (bei n=-1) oder das letzte Zeichen des Namenseintrages (bei n=1).

```
LFA     ( adr1 --> adr2)
```

Erwartet die Parameterfeldadresse adr1 eines Wortes und übergibt dessen Linkfeldadresse adr2.

```
CFA     ( adr1 --> adr2)
```

Erwartet die Parameterfeldadresse adr1 eines Wortes und übergibt dessen Codefeldadresse adr2.

```
NFA     ( adr1 --> adr2)
```

Erwartet die Parameterfeldadresse adr1 eines Wortes und übergibt dessen Namensfeldadresse adr2.

```
PFA     ( adr1 --> adr2)
```

Erwartet die Namensfeldadresse adr1 eines Wortes und übergibt dessen Parameterfeldadresse adr2.

```
ID.     ( adr1 --> )
```

Erwartet die Namensfeldadresse eines Wortes und bibt dafür den Wortnamen an das Ausgabegerät aus.

```
      ( --> adr )  im Execute-Mode
      ( --> )      im Compile-Mode
```

Wird verwendet in der Form

```
       XXX
```

Im Execute-Mode wird die Parameterfeldadresse adr des Wortes auf dem Stack übergeben.
Im Compile-Mode wird zunächst LIT und danach die Parameterfeld-adresse des Wortes XXX ins Wörterbuch compiliert.

8.18.4. Streichen von Worten

FORGET (-->)

Wird verwendet in der Form

```
FORGET XXX .
```

Das Wort XXX und alle im Wörterbuch nach steigenden Adresse folgenden Worte werden gestrichen.

8.18.5. Systeminitialisierung

SP! (-->)

Initialisiert den Datenstackpointer mit dem in S0 gehaltenen Wert.

RP! (-->)

Initialisiert den Returnstackpointer mit dem in R0 gehaltenen Wert.

COLD (-->)

Führt eine vollständige Systeminitialisierung aus, alle zusätzlichen Worteintragungen gehen verloren.

WARM (-->)

Führt eine teilweise Systeminitialisierung aus, hier gleiche Wirkung wie ABORT.

8.18.6. Sonstige

BUSIN (b1 --> b2)

Führt eine Einleseoperation vom E/A-Port b1 aus und übergibt die dort vorgefundenen Daten als b2.

BUSOUT (b1 b2 -->)

Gibt die Daten b1 an das E/A-Port b2 aus.

DUMP (adr u -->)

Realisiert einen Hex-Dump ab Adresse adr mit der Länge u Bytes.

TASK (-->)

Funktionsloses Wort, das beispielsweise in andere Wortdefinitionen eingebaut werden kann, um dort provisorisch einen Platz frei zu halten.

```
+ORIGIN     ( n --> adr )
-------------------------
```
Übergibt die Adresse des n. Bytes relativ zum Anfang der Boot-Area, siehe 7.2.1 .

8.19. Computerspezifische Erweiterungen, die nicht zum fig-Standard gehören

8.19.1. Farb- und Cursorsteuerung für Zeichenausgabe

```
INK     ( n --> )
-----------------
```
Setzt n als neue Vordergrundfarbe. Nur die unteren vier Bit von n werden verwendet.

```
PAPER     ( n --> )
-------------------
```
Setzt n als neue Hintergrundfarbe. Nur die unteren drei Bit von n werden verwendet.

```
BLINK     ( --> )
-----------------
```
Umschalten des Blink-Modus. War dieser vorher gesetzt, wird er nun zurückgesetzt und umgekehrt.

```
COLOR     ( n1 n2 --> )
-----------------------
```
Setzt n1 als neue Vorder- und n2 als neue Hintergrundfarbe.

```
LOC     ( n1 n2 --> )
---------------------
```
Der Cursor wird auf Zeile n1 und auf Spalte n2 des eingestellten Fensters positioniert.

```
WINDOW     ( n1 n2 n3 n4 --> )
------------------------------
```
Erzeugung eines Fensters ab Spalte n1 mit der Breite n2 und ab Zeile n3 mit der Höhe n4. Sollte das Fenster nicht korrekt sein (paßt nicht in den Bildschirm), so wird die Fehlermeldung MSG#5 erzeugt.
Text: invalid window

8.19.2. Grafikausgabe

```
INKP     ( n --> )
------------------
```
Setzt n als neue Vordergrundfarbe für die Grafikausgabe. Nur die unteren vier Bit von n werden verwendet.

```
BLINKP     ( --> )
------------------
```
Umschalten des Blinkmodus für die Grafikausgabe, siehe BLINK .

```
PIX     ( f n1 n2 --> )
-----------------------
```
Setzen oder löschen eines Bildpunktes mit der x-Koordinate n1 und der y-Koordinate n2.
 f gleich 0 --- Punkt wird gelöscht
 f ungleich 0 --- Punkt wird gesetzt

```
PLOT     ( f n1 n2 n3 n4 --> f n5 n6 )
-------------------------------------
```
Zeichnen oder Löschen einer Strecke
 f gleich 0 -- Strecke wird gelöscht
 f ungleich 0 -- Strecke wird gezeichnet
 n1 -- Startpunkt in X-Richtung
 n2 -- Startpunkt in y-Richtung
 n3 -- Länge in x-Richtung
 n4 -- Länge in y-Richtung
 n5 -- Endpunkt in x-Richtung
 n6 -- Endpunkt in y-Richtung

8.19.3. Tonausgabe

```
Sound     ( n1 n2 n3n 4 n5 n6 --> )
-----------------------------------
```
Erzeugung eines Tones. n1 und n3 werden als Zeitkonstanten und n2 und n4 als Vorteiler in CTC-Kanäle eingeschrieben. Mit n5 (0...31) wird die Lautstärke und mit n6 (0...255) die Tondauer eingestellt.

8.19.4. Sonstiges

IC@ (adr --> b)

Hat die gleiche Wirkung wie C@, nur daß hier zusätzlich der IRM vorher ein- und hinterher ausgeschaltet wird.

IC! (adr --> n)

Wie C!, nur mit Ein- und Ausschalten des IRM.

I@ (adr --> n)

Wie @, nur mit Ein- und Ausschalten des IRM.

I! (n adr -->)

wie !, nur mit Ein- und Ausschalten des IRM.

CAOS (n1 n2 -->)

Aufruf des CAOS-Unterprogramms n1. Diesem Unterprogramm wird n2 als Parameter im HL-Register übergeben.

BYE (-->)

Verlassen von FORTH und Rückkehr in CAOS.

SWITCH (b1 b2 -->)

Das Modulsteuerbyte b1 wird an die Moduladresse b2 ausgegeben.

MODUL (b1 --> b2 b3)

Das Modulsteuerbyte b3 und der Modultyp b2 der Moduladresse b1 werden übergeben.

8.20. Ein- und Ausgabe von bzw. zu Peripheriegeräten

SETIN (n -->)

Setzt neuen Zeiger auf CAOS-Eingabeunterprogramm (normal: Tastatureingabe, entspricht der Unterprogramm-Nr. 4).

SETOUT (n -->)

Setzt neuen Zeiger auf CAOS-Ausgabeunterprogramm (normal: Bildschirmausgabe, entspricht der Unterprogramm-Nr. 0).

```
NORMIN    ( --> )
-----------------
Setzt   den  Zeiger  auf  das   normale   CAOS-Eingabeunterprogramm
(Tastatureingabe). NORMIN entspricht 4 SETIN.


NORMOUT     ( --> )
-------------------
Setzt  den Zeiger auf das normale CAOS-Ausgabeunterprogramm  (Bild-
schirmausgabe). NORMOUT entspricht 0 SETOUT.
```

9. Literatur

/1/* System-Handbuch zum KC85/3
VEB Mikroelektronik "Wilhelm Pieck" Mühlhausen

/2/* Übersichten zum KC85/3
VEB Mikroelektronik "Wilhelm Pieck" Mühlhausen

/3/* Beschreibung zur Programmkassette C0171 V24-Software
VEB Mikroelektronik "Wilhelm Pieck" Mühlhausen

/4/* Systembeschreibung zum HC-CAOS
VEB Mikroelektronik "Wilhelm Pieck" Mühlhausen

* bei Verwendung eines KC 85/2 gilt /4/

Anhang A

A.1 Farbcodierungen

Vordergrund	Kode	Hintergrund	Vordergrund	Kode
Schwarz	0	Schwarz	Schwarz	8
Blau	1	Blau	Violett	9
Rot	2	Rot	Orange	10
Purpur	3	Purpur	Purpurrot	11
Grün	4	Grün	Grünblau	12
Türkis	5	Türkis	Blaugrün	13
Gelb	6	Gelb	Gelbgrün	14
Weiß	7	Grau	Weiss	15

A2. Der ASCII-Code

A.2.1. Die Steuerzeichen (Code 0–31)

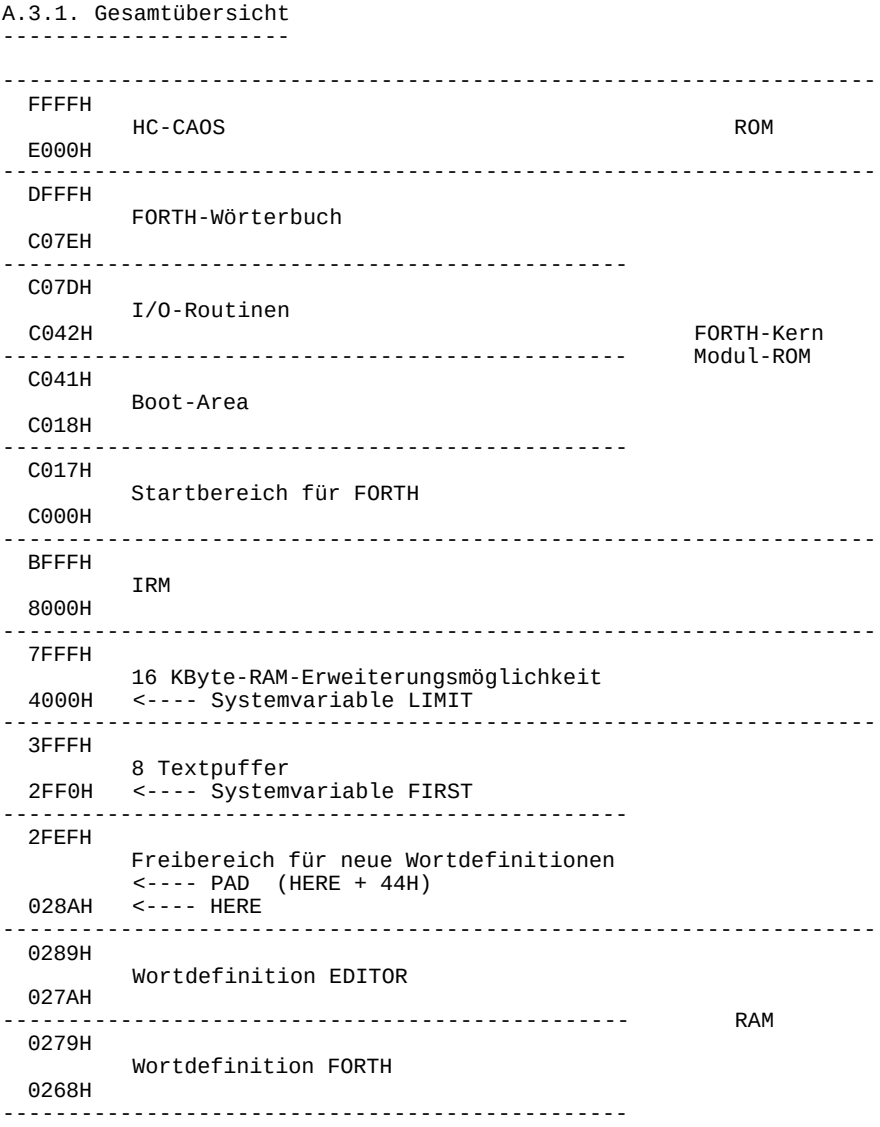
Code dezimal	Code Hexadezimal	Name	Bedeutung
0	00	DUMMY	
1	01	CLEAR	1 Zeichen löschen
2	02	ESCAPE	1 Zeile löschen
3	03	BREAK	
4	04	-	
5	05	-	
6	06	-	
7	07	BEEP	Tonausgabe
8	08	CUL	Cursor links
9	09	CUR	Cursor rechts
10	0A	CUD	Cursor runter
11	0B	CUU	Cursor hoch
12	0C	CLS	Bildschirm löschen
13	0D	CR	ENTER
14	0E	-	
15	0F	HCOPY	Sonderprogramm
16	10	HOME	Cursor links oben
17	11	PAGE	Page-Mode ein
18	12	SCROLL	Scroll-Mode ein
19	13	STOP	STOP
20	14	CLICK	Tasten-Click ein
21	15	-	
22	16	-	
23	17	-	
24	18	-	
25	19	CCR	Cursor an Zeilenanfang
26	1A	INS	INS (Zeichen einfügen)
27	1B	-	
28	1C	-	
29	1D	-	
30	1E	-	
31	1F	DEL	DEL (Zeichen löschen)

A.2.2. Die Zeichencodes von 32 bis 127

HEX	DEC		HEX	DEC		HEX	DEC	
20	32	Sp	40	64	@	60	96	@
21	33	!	41	65	A	61	97	a
22	34	"	42	66	B	62	98	b
23	35	#	43	67	C	63	99	c
24	36	$	44	68	D	64	100	d
25	37	%	45	69	E	65	101	e
26	38	&	46	70	F	66	102	f
27	39	'	47	71	G	67	103	g
28	40	(	48	72	H	68	104	h
29	41	)	49	73	I	69	105	i
2A	42	*	4A	74	J	6A	106	j
2B	43	+	4B	75	K	6B	107	k
2C	44	,	4C	76	L	6C	108	l
2D	45	-	4D	77	M	6D	109	m
2E	46	.	4E	78	N	6E	110	n
2F	47	/	4F	79	O	6F	111	o
30	48	0	50	80	P	70	112	p
31	49	1	51	81	Q	71	113	q
32	50	2	52	82	R	72	114	r
33	51	3	53	83	S	73	115	s
34	52	4	54	84	T	74	116	t
35	53	5	55	85	U	75	117	u
36	54	6	56	86	V	76	118	v
37	55	7	57	87	W	77	119	w
38	56	8	58	88	X	78	120	x
39	57	9	59	89	Y	79	121	y
3A	58	:	5A	90	Z	7A	122	z
3B	59	;	5B	91	■	7B	123	ä
3C	60	<	5C	92	|	7C	124	ö
3D	61	=	5D	93	¬	7D	125	ü
3E	62	>	5E	94	^	7E	126	ß
3F	63	?	5F	95	_	7F	127	

A.3. Speicheraufteilung

A.3.1. Gesamtübersicht



```
-----------------------------------------------
  0267H
         Wortdefinition (ABORT)
  0258H
-----------------------------------------------
  0257H
         Ein- und Ausschaltroutine für IRM
  0248H
-----------------------------------------------
  0247H
         Feld für System-Variablen (User-Area)
  0200H
-----------------------------------------------       RAM
  01FFH
         CAOS-Arbeitsspeicher
  0180H
-----------------------------------------------
  017FH
         User-Feld-Zeiger
  017EH
-----------------------------------------------
  017DH
         Returnstack
         Terminal-Input-Buffer
  00E0H
-----------------------------------------------
  00DFH
         Datenstack
  0000H
-----------------------------------------------------------------------
```

A.3.2. Image Repetition Memory (IRM)

Dezimale Adresse	Hexadezimale Adresse	Verwendung
32768 ... 43007	8000 ... A7FF	Pixel – RAM
43008 ... 45567	A800 ... B1FF	Color – RAM
45568 ... 46847	B200 ... B6FF	ASCII – RAM
46848 ... 46975	B700 ... B77F	Kassettenpuffer
46976 ... 47103	B780 ... B7FF	Monitor–RAM
47104 ... 47359	B800 ... B8FF	Modul–Steuerwort
47360 ... 47487	B900 ... B97F	Funktionstasten
47488 ... 47615	B980 ... B9FF	Fenstervektorsp.
47646 ... 49151	BA00 ... BFFF	frei

A.3.2. Read only Memory (ROM) – Betriebssystem

Dezimale Adresse	Hexadezimale Adresse	Verwendung
49152 ... 57343	C000H ... DFFFH	BASIC-Interpreter (beim KC85/2 nicht belegt) bzw. FORTH-Modul
57344 ... 65535	E000H ... FFFFH	HC–CAOS

Anhang B

Belegung der Screens 1 und 2 mit Mitteilungen

```
Achtung  !  Nach dem fig-Standard stehen diese Mitteilungen auf den
            Screens 4 und 5.
SCR  #  1

 0 ( ******** MESSAGES ******** )
 1 empty stack                                    (Stack ist leer)
 2 dictionary full                               (Wörterbuch voll)
 3 has incorrect adress mode          (hat inkorrekten Adress-Modus)
 4 isn´t unique                                  (schon vorhanden)
 5 invalid window                              (verkehrtes Fenster)
 6 memory range                                  (Speicher-Ende)
 7 full stack                                    (Stack ist voll)
 8 tape error                                       (Band-Fehler)
 9
10
11
12
13
14
15 *** Screen listing KC 85/2 ***

SCR  #  2

 0 ( ******** MESSAGES ******** )
 1 comp. only, use in definition    (nur bei Kompilation verwenden)
 2 execution only                          (nur im Execute-Modus)
 3 conditionals not paired               (Bedingungen nicht paarig)
 4 definition not finished               (Definition nicht beendet)
 5 in protected mode                      (in geschütztem Speicher)
 6 use only when loading                   (nur bei Load verwenden)
 7 off current editing screen              (außerhalb des Screens)
 8 declare vocabulary                        (definiere Vokabular)
 9
10
11
12
13
14
15
```

Die Mitteilungen 5, 6 und 8 entsprechen nicht dem fig-Standard.
Die Mitteilung 15 ist beliebig modifizierbar, sie wird immer als Abschluss von TRIAD ausgegeben.

Anhang C

Die Belegung des USER-Feldes

Offset (hexadezimal)	Variable
00	--
02	--
04	--
06	S0
08	R0
0A	TIB
0C	WIDTH
0E	WARNING
10	FENCE
12	DP
14	VOC-LINK
16	FIRST
18	LIMIT
1A	--
1C	OFFSET
1E	BLK
20	IN
22	OUT
24	SCR
26	CONTEXT
28	CURRENT
2A	STATE
2C	BASE
2E	DPL
30	FLD
32	CSP
34	R#
36	HLD
38	REC
3A	--
3C	--
3E	--
40	--
42	--
44	--
46	--

Anhang D

Die Boot–Area

```
00 +ORIGIN   NOP
01           JP    COLD        ;Kaltstart–Eintrittspunkt
04           NOP
05           JP    WARM        ;Warmstart–Eintrittspunkt
08           DEFW   VERS       ;Versionsnummer
0A           DEFW   REL        ;Überarbeitungsnummer
0C           DEFW   TASK-7     ;Namensfeldadresse des letzten
                               ;im Kernvokabular
                               ;definierten Wortes
0E           DEFW   BACKS      ;ASCII–Code für BACKSPACE
10           DEFW   UA         ;Init  User-Area-Pointer
12           DEFW   STACK      ; "    Datenstack-Pointer
14           DEFW   RETST      ; "    Returnstack-Pointer
16           DEFW   TIB        ; "    TIB
18           DEFW   1FH        ; "    WIDTH
1A           DEFW   0          ; "    WARNING
1C           DEFW   FEN        ; "    FENCE
1E           DEFW   DPNTR      ; "    DP
20           DEFW   EDITR+8    ; "    VOC-LINK
22           DEFW   ....       ; "    FIRST
24           DEFW   ....       ; "    LIMIT
26           DEFW   ....       ; "    IOTAB
28           DEFW   0          ;Initialisierung OFFSET
```

Anhang E

Forth-Word-Register

FORTH-Wort	Seiten-Nr. im Text	Seiten-Nr. im Überblick
EXECUTE		108
EXPECT		95
FENCE		118
FILL	44	93
FIRST	42	102
FLD		118
FORGET	18	120
FORTH	48	114
H	49	117
HERE	48	112
HEX	28	90
HLD	62	96
HOLD	63	96
I	56	101
I!		123
I@		123
IC!		123
IC@		123
ID.	67	119
IF	52	97
IMMEDIATE	65	111
IN	48	107
INDEX		103
INIT-BUFFERS		103
INK		121
INKP		122
INTERPRET	60	108
IOTAB		102
KEY	14	94
L		117
LATEST		114
LEAVE	56	101
LFA	67	119
LIMIT	42	102
LINE	49	116
LIST	43	103
LIT		113
LITERAL	69	113
LOAD	47	108
LOC		121
LOOP	55	100
M*	34	88
M/	34	88
M/MOD	35	89
MAX	33	88
MESSAGE	78	114
MIN	33	88
MINUS	32	89
MOD	34	88
MODUL		123
MOVE	49	117
NEXT		118
NFA	67	119
NORMIN	85	124
NORMOUT	85	124
NUMBER	15/38	107
OFFSET		105
OR		90
OUT		95
OVER		92
P	49	117
PAD	48	116

FORTH-Wort	Seiten-Nr. im Text	Seiten-Nr. im Überblick
PAPER		121
PFA	67	119
PIX		122
PLOT	16	122
QUERY	60	96
QUIT	61	108
R#		118
R	22	92
R	49	116
R0	81	118
R/W		104
R>	22	92
REC		104
REC/BLK		104
REPEAT	59	99
ROT	22	92
RP!		120
RP@		92
S	49	117
S->D	62	89
S0	57	118
SCR	44	116
SETIN	85	123
SETOUT	85	123
SIGN	63	96
SMUDGE	65	111
SOUND		122
SP!	39/81	120
SP@	37/58	92
SPACE		94
SPACES		94
STATE	69	111
SWAP	22	91
SWITCH		123
T	49	117
TASK		120
TEXT	49	116
TIB	41/81	95
TOGGLE		93
TRAVERSE		119
TRIAD		104
TYPE	63	94
U*	34	89
U.	30	97
U/	34	89
U<		91
UNTIL	59	98
USER	64/84	109
VARIABLE	23	109
VERIFY	45	105
VLIST	15	114
VOC-LINK	81	113
VOCABULARY		113
WARM		120
WARNING	78	114
WHILE	59	99
WIDTH		111
WINDOW		121
WORD		106
XOR		90
Za.hl	28	
Zahl.	28	

Abschrift erstellt

Elmar Klinder
Götz Hupe

mikroelektronik

RFT

veb mikroelektronik »wilhelm pieck« mühlhausen
im veb kombinat mikroelektronik