Handbuch

zum

PC-volksFORTH83

(rev. 3.81)

Handbuch zum PC-volksFORTH83 rev. 3.81

Die Autoren haben sich in diesem Handbuch um eine vollständige und akkurate Darstellung bemüht. Die im Handbuch enthaltenen Informationen dienen jedoch allein der Produktbeschreibung und sind nicht als zugesicherte Eigenschaften im Rechtssinne aufzufassen. Etwaige Schadensersatzansprüche gegen die Autoren gleich aus welchem Rechtsgrund, sind ausgeschlossen. Es wird keine Gewähr übernommen, daß die angegebenen Verfahren frei von Schutzrechten Dritter sind.

Alle Rechte vorbehalten. Ein Nachdruck, auch auszugsweise, ist nur zulässig mit Einwilligung der Autoren und genauer Quellenangabe sowie Einsendung eines Belegexemplars an die Autoren.

(c) 1985,1986 Bernd Pennemann, Georg Rehfeld, Dietrich Weineck
(c) 1988,1989 Klaus Schleisiek, Jörg Staben, Klaus Kohl
- Mitglieder der FORTH Gesellschaft e.V. -

Unser Dank gilt der gesamten FORTH-Gemeinschaft,
insbesondere Charles Moore, Michael Perry und Henry Laxen

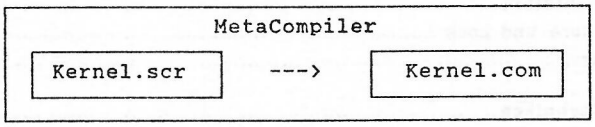
1. Auflage Mai 1989

Inhaltsverzeichnis

1. Prolog

volksFORTH83 ist eine Sprache, die in verschiedener Hinsicht ungewöhnlich ist.
Denn FORTH selbst ist nicht nur eine Sprache, sondern ein im Prinzip grenzenloses Programmiersystem. Eines der Hauptmerkmale des Programmiersystems FORTH ist seine Modularität. Diese Modularität wird durch die kleinste Einheit eines FORTH-Systems, das **WORT**, gewährleistet.
In FORTH werden die Begriffe Prozedur, Routine, Programm, Definition und Befehl alle gleichbedeutend mit Wort gebraucht. FORTH besteht also, wie jede andere natürliche Sprache auch, aus Wörtern.

Diese FORTH-Worte kann man als bereits kompilierte Module betrachten, wobei immer ein Kern aus einigen hundert Worten durch eigene Worte erweitert wird. Diese Worte des Kerns sind in einem FORTH83-Standard festgelegt und stellen sicher, daß Standard-Programme ohne Änderungen auf dem jeweiligen FORTH-System lauffähig sind.
Ungewöhnlich ist, daß der Programmtext des Kerns selbst ein FORTH-Programm ist, im Gegensatz zu anderen Programmiersprachen, denen ein Maschinensprach-Programm zugrunde liegt. Aus diesem Kern wird durch ein besonderes FORTH-Programm, den MetaCompiler, das lauffähige KERNEL.COM erzeugt:



Wie fügt man nun diesem lauffähigen Kern eigene Worte hinzu?
Das Kompilieren der Worte wird in FORTH mit COLON ":" eingeleitet und mit SEMICOLON ";" abgeschlossen:

: erwartet bei der Ausführung einen Namen und ordnet diesem Namen alle nachfolgenden Wörter zu.

; beendet diese Zuweisung von Wörtern an den Namen und stellt das neue Wort unter diesem Namen für Aufrufe bereit.

1.1 Interpreter und Compiler

Ein klassisches FORTH-System stellt immer sowohl einen Interpreter als auch einen Compiler zur Verfügung. Nach der Einschaltmeldung oder einem Druck auf die <CR>-Taste wartet der FORTH-Interpreter mit dem FORTH-typischen "ok" auf Ihre Ein-

gabe. Sie können ein oder mehrere Befehlswörter in eine Zeile schreiben. volksFORTH beginnt erst nach einem Druck auf die RETURN-Taste <CR> mit seiner Arbeit, indem es der Reihe nach jeden Befehl in der Eingabezeile abarbeitet. Dabei werden Befehlsworte durch Leerzeichen begrenzt. Der Delimiter (Begrenzer) für FORTH-Prozeduren ist also das Leerzeichen, womit auch schon die Syntax der Sprache FORTH beschrieben wäre.

Der Compiler eines FORTH-Systems ist also Teil der Interpreteroberfläche. Es gibt daher keinen Compiler-Lauf zur Erstellen des Programmtextes wie in anderen Compiler-Sprachen, sondern der Interpreter wird mit allen zur Problemlösung notwendigen Worten als Anwenderprogramm abgespeichert.

Auch : (COLON) und ; (SEMICOLON) sind kompilierte Worte, die aber für das System den Compiler ein- und ausschalten. Da sogar die Worte, die den Compiler steuern, "normale" FORTH-Worte sind, fehlen in FORTH die in anderen Sprachen üblichen Compiler-Optionen oder Compiler-Schalter. Der FORTH-Compiler wird mit FORTH-Worten gesteuert.

Der Aufruf eines FORTH-Wortes erfolgt über seinen Namen ohne ein explizites CALL oder GOSUB. Dies führt zum FORTH-typischen Aussehen der Wortdefinitionen:

```
: <name>
   <wort1> <wort2> <wort3> ... ;
```

Die Standard-Systemantwort in FORTH ist das berühmte "ok". Ein Anforderungszeichen wie 'A>' bei DOS oder ']' beim guten APPLE II gibt es nicht! Das kann dazu führen, daß nach einer erfolgreichen Aktion der Bildschirm völlig leer bleibt; getreu der Devise:

Keine Nachrichten sind immer gute Nachrichten !

Und - ungewöhnlicherweise - benutzt FORTH die sogenannte Postfix notation (UPN) vergleichbar den HP-Taschenrechnern, die in machen Kreisen sehr beliebt sind. Das bedeutet, FORTH erwartet immer erst die Argumente, dann die Aktion. Statt

```
                3 + 2  und  (5 + 5) * 10
heißt es        2 3 +  und  5 5 + 10 *  .
```

Da die Ausdrücke von links nach rechts ausgewertet werden, gibt es in FORTH keine Klammern.

Stack

Ebenso ungewöhnlich ist, daß FORTH nur ausdrücklich angeforderte Aktionen ausführt: Das Ergebnis Ihrer Berechnungen bleibt solange in einem speziellen Speicherbereich, dem Stack, bis es mit einem Ausgabebefehl (meist .) auf den Bildschirm oder dem Drucker ausgegeben wird.
Da die FORTH-Worte den Unterprogrammen und Funktionen anderer Programmiersprachen entsprechen, benötigen sie gleichfalls die Möglichkeit, Daten zur Verarbeitung zu übernehmen und das Ergebnis abzulegen. Diese Funktion übernimmt der

STACK. In FORTH werden Parameter für Prozeduren selten in Variablen abgelegt, sondern meist über den Stack übergeben.

Assembler

Innerhalb einer FORTH-Umgebung kann man sofort in der Maschinensprache des Prozessors programmieren, ohne den Interpreter verlassen zu müssen. Assembler-Definitionen sind den FORTH-Programmen gleichwertige FORTH-Worte.

Vokabular-Konzept

Das volksFORTH verfügt über eine erweiterte Vokabular-Struktur, die von W. Ragsdale vorgeschlagen wurde. Dieses Vokabular-Konzept erlaubt das Einordnen der FORTH-Worte in logische Gruppen.
Damit können Sie notwendige Befehle bei Bedarf zuschalten und nach Gebrauch wieder wegschalten. Darüberhinaus ermöglichen die Vokabulare den Gebrauch gleicher Namen für verschiedene Worte, ohne in einen Namenskonflikt zu kommen. Eine im Ansatz ähnliche Vorgehensweise bietet das UNIT-Konzept moderner PASCAL- oder MODULA-Compiler.

FORTH-Dateien

FORTH verwendet oftmals besondere Dateien für seine Programme. Dies ist historisch begründet und das Erbe einer Zeit, als FORTH noch sehr oft Aufgaben des Betriebssystems übernahm. Da gab es ausschließlich FORTH-Systeme, die den Massenspeicher vollständig selbst ohne ein DOS verwalteten und dafür ihre eigenen Dateistrukturen benutzten.
Diese Dateien sind sogenannte Blockfiles und bestehen aus einer Aneinanderreihung von 1Ø24 Byte großen Blöcken. Ein solcher Block, der gerne **SCREEN** genannt wird, ist die Grundlage der Quelltext-Bearbeitung in FORTH.
Allerdings können mit dem volks4TH auch Dateien bearbeitet werden, die im Dateiformat des MS-DOS vorliegen, sog. "Stream Files".

Generell steht hinter jeder Sprache ein bestimmtes Konzept, und nur mit Kenntnis dieses Konzeptes ist möglich, eine Sprache effizient einzusetzen. Das Sprachkonzept von FORTH wird beschrieben in dem Buch "In FORTH denken" von Leo Brodie (Hanser Verlag).

Einen ersten Eindruck vom volksFORTH83 und von unserem Stolz darüber soll dieser Prolog vermitteln. volksFORTH83 ist ein "Public-Domain"-System, bei dessen Leistungsfähigkeit sich die Frage stellt:

1.2 Warum stellen wir dieses System frei zur Verfügung ?

Die Verbreitung, die die Sprache FORTH gefunden hat, war wesentlich an die Existenz von figFORTH geknüpft. Auch figFORTH ist ein public domain Programm, d.h. es darf weitergegeben und kopiert werden. Trotzdem haben sich bedauerlicherweise verschiedene Anbieter die einfache Adaption des figFORTH an verschiedene Rechner sehr teuer bezahlen lassen.

Das im Jahr 1979 erschienene figFORTH ist heute nicht mehr so aktuell, weil mit der weiteren Verbreitung von Forth eine Fülle von eleganten Konzepten entstanden sind, die z.T. in den Forth83-Standard Eingang gefunden haben. Daraufhin wurde von Laxen und Perry das F83 geschrieben und als Public Domain verbreitet. Dieses freie 83er-Standard-FORTH mit seinen zahlreichen Utilities ist recht komplex und wird auch nicht mit Handbuch geliefert.

Wir haben ein neues Forth für verschiedene Rechner entwickelt. Das Ergebnis ist das volksFORTH83, eines der besten Forth-Systeme, die es gibt. Das volksFORTH soll an die Tradition der oben genannten Systeme, insbesondere des F83, anknüpfen und die Verbreitung der Sprache FORTH fördern.

volksFORTH wurde unter dem Namen ultraFORTH zunächst für den C64 geschrieben. Nach Erscheinen der Rechner der Atari ST-Serie entschlossen wir uns, auch für sie ein volksFORTH83 zu entwickeln. Die erste ausgelieferte Version 3.7 war, was Editor und Massenspeicher betraf, noch stark an den C64 angelehnt. Sie enthielt jedoch schon einen verbesserten Tracer, die GEM-Bibliothek und die anderen Tools für den ST. Der nächste Schritt bestand in der Einbindung der Betriebssystem-Files. Nun konnten Quelltexte auch vom Desktop und mit anderen Utilities verarbeitet werden. Die dritte Adaption des volksFORTH entstand für die CP/M-Rechner (8Ø8Ø-Prozessoren), wobei speziell für den Schneider CPC auch die Grafikfähigkeit unterstützt wird. Zuletzt wurde das volksFORTH für die weitverbreiteten Rechner der IBM PC-Serie angepaßt. Mit der jetzt ausgelieferten Version 3.81 ist das volksFORTH vollständig.

1.3 Warum soll man in volksFORTH83 programmieren ?

Das volksFORTH83 ist ein ausgesprochen leistungsfähiges und kompaktes Werkzeug.

Durch residente Runtime-library, Compiler, Editor und Debugger sind die ermüdenden ECLG-Zyklen ("Edit, Compile, Link and Go") überflüssig. Der Code wird Modul für Modul entwickelt, kompiliert und getestet.

Der integrierte Debugger ist die perfekte Testumgebung für Worte. Es gibt keine riesigen Hexdumps oder Assemblerlistings, die kaum Ähnlichkeit mit dem Quelltext haben.

Ein anderer wichtiger Aspekt ist das Multitasking. So wie man ein Programm in einzelne, unabhängige Module oder Worte aufteilt, so sollte man es auch in ein-

zelne, unabhängige Prozesse aufteilen können. Das ist in den meisten Sprachen nicht möglich. Das volksFORTH83 besitzt einen einfachen, aber leistungsfähigen Multitasker.

Schließlich besitzt das volksFORTH83 noch eine Fülle von Details, über die andere FORTH-Systeme nicht verfügen:

- Es benutzt an vielen Stellen Vektoren und sog. deferred Worte, die eine einfache Umgestaltung des Systems für verschiedene Gerätekonfigurationen ermöglichen.
- Es besitzt einen Heap für "namenlose" Worte oder für Code, der nur zeitweilig benötigt wird.
- Der Blockmechanismus ist so schnell, daß er auch sinnvoll für die Bearbeitung großer Datenmengen, die in Files vorliegen, eingesetzt werden kann.
- Das System umfaßt Tracer, Decompiler, Multitasker, Assembler, Editor, Printer-Interface ...

Das volksFORTH83 erzeugt, verglichen mit anderen FORTH-Systemen, relativ schnellen Code, der aber langsamer als der anderer Compilersprachen ist.

Mit diesem Handbuch soll die Unterstützung des volksFORTH83 noch nicht Zuende sein. Die FORTH Gesellschaft e.V., ein gemeinnütziger Verein, bietet dafür die Plattform. Sie gibt die Vereins-FORTH-Zeitschrift "VIERTE DIMENSION" heraus und betreibt den FORTH-Tree, eine ungewöhnliche, aber sehr leistungsfähige Mailbox.

Die im Frühsommer 1989 aktuelle Adressen:

FORTH-Büro:

FORTH Gesellschaft e.V.
Postfach 1110
8044 Unterschleißheim
Tel. 089/3173784

volksFORTH-Vertrieb:

Michael & Klaus Kohl
Pestalozzistr. 69
8905 Mering
Tel. 08233/30524

1.4 Hinweise des Lektors

Diesem Handbuch zum PC-volksFORTH83 ist sowohl als Nachschlagewerk als auch als Lehrbuch für FORTH (speziell volksFORTH) gedacht. Deshalb handelt es sich nicht, wie bei den anderen volksFORTH-Handbücher, um eine Auflistung des Vokabulars. Statt dessen wird mit ausführlichen Beschreibungen und Programmbeispielen in vielen Kapiteln die Möglichkeiten des FORTH-Systems erklärt. Ergänzt werden die einzelnen Kapitel jeweils um Wortbeschreibungen der darin vorkommenden Befehle (Glossar). Sollten bestimmte Befehle gesucht werden, so ist die Seitennummer aus dem ausführlichen Index zu entnehmen.

Zur Unterscheidung von Beschreibung, FORTH-Worten, Programm-Eingaben und -ausgaben wird mit unterschriftlichen Schrifttypen gearbeitet:

Beschreibungen	erfolgen in Proportionalschrift mit Randausgleich.
FORTH-Befehle	werden im Text durch Fettschrift hervorgehoben.
`Eingaben`	und
`Programmlistings`	verwenden die nichtproportionale Schriftart Courier.
<u>Ausgaben</u>	des FORTH-Interpreter/Compiler sind unterstrichen.

2. Einstieg ins volksFORTH

Damit Sie sofort beginnen können, wird in diesem Kapitel beschrieben,

- wie man das System startet
- wie man sich im System zurechtfindet
- wie man ein fertiges Anwendungsprogramm erstellt
- wie man ein eigenes Arbeitssystem zusammenstellt

2.1 Die Systemdiskette

Zu Ihrem Handbuch haben Sie eine Diskette erhalten. Fertigen Sie auf jeden Fall mit dem DOS-Befehl **diskcopy** eine Sicherheitskopie dieser Diskette an. Die Gefahr eines Datenverlustes ist groß, da FORTH Ihnen in jeder Hinsicht freie Hand läßt - auch beim versehentlichen Löschen Ihrer Systemdisketten !!

Die Diskette, auf der Ihr volksFORTH-System ausgeliefert wird, enthält folgende Dateien:

INSTALL	BAT	ist ein Installationsprogramm, das volks4TH auf einem angegebenen Laufwerk einrichtet.
PKXARC	COM	ist ein Dienstprogramm zum Komprimieren und Dekomprimieren von Dateien.
KERNEL	COM	
MINIMAL	COM	
VOLKS4TH	COM	diese drei COM-Files sind drei volksFORTH-Systeme in verschiedenen Ausbaustufen.
FORTH1	ARC	
FORTH2	ARC	diese beiden ARC-Dateien enthalten die Quelltexte des gesamten FORTH-Systems, müssen aber erst von PKXARC entpackt werden.
VOLKS4TH	DOC	ist eine ergänzende Dokumentation, die Nachträge enthält.
READ	ME	enthält zusätzliche wichtige Hinweise.

Wenn Sie Ihr System, wie in der Datei READ.ME auf der Diskette beschrieben, idealerweise auf einer Festplatte installiert haben, finden Sie deutlich mehr Files

vor. Ein Zeichen für die Platzersparnis durch das Datei-Kompressionsprogramm PKARC.

VOLKS4TH	COM	als Ihr komplettes Arbeitsssystem enthält resident das Fileinterface, den Editor, den Assembler und von Ihnen eingefügte Werkzeuge (tools).
MINIMAL	COM	ist eine Grundversion, die oft benötigte Systemteile enthält. Diese ist notwendig, da FORTH-Systeme allgemein nicht über einen Linker verfügen, sondern ausgehend vom Systemkern die zur Problemlösung notwendigen Einzelprogramme schrittweise hinzukompiliert werden.
KERNEL	COM	ist eine Grundversion, die nur den Sprachkern enthält. Damit können Sie eigene FORTH-Versionen mit z.B. einem veränderten Editor zusammenstellen und dann mit **SAVESYSTEM <name>** als fertiges System abspeichern. In der gleichen Art können Sie auch fertige Applikationen herstellen, denen man ihre FORTH-Abstammung nicht mehr ansieht.
KERNEL	SCR	enthält die Quelltexte des Sprachkerns. Eben dieser Quelltext ist mit einem Target-Compiler kompiliert worden und entspricht exakt dem KERNEL.COM. Sie können sich also den Compiler ansehen, wenn Sie wissen wollen, wie das volksFORTH83 funktioniert !
VOLKS4TH	SYS	enthält einen Ladeblock (Block 1), der alle Teile kompiliert, die zu Ihrem Arbeitssystem gehören. Mit diesem Loadscreen ist aus KERNEL.COM das File VOLKS4TH.COM zusammengestellt worden.
EXTEND	SCR	enthält Erweiterungen des Systems. Hier tragen Sie auch persönliche Erweiterungen ein.

CED	SCR	enthält den Quelltext des Kommandozeilen Editors, mit dem die Kommandozeile des Interpreters editiert werden kann. Soll dieser CED ins System eingefügt werden, so ist diese Datei mit
		`include ced.scr`
		`savesystem volks4TH.com`
		ins volksFORTH einzukompilieren.

HISTORY		wird von CED angelegt und enthält die zuletzt eingegebenen Kommandos.
STREAM	SCR	enthält zwei oft gewünschte Dienstprogramme: Die Umwandlung von Text-Dateien (stream files) in Block-Dateien (block files) und zurück.

DISASM SCR enthält den Dis-Assembler, der - wie beim CED beschrieben - ins System eingebaut werden kann.

2.2 Die Oberfläche

Wenn Sie VOLKS4TH von der DOS-Ebene starten, meldet sich volksFORTH83 mit einer Einschaltmeldung, die die Versionsnummer rev. <xxxx> enthält.

Was Sie nun von volksFORTH sehen, ist die Oberfläche des Interpreters. FORTH-Systeme und damit auch volksFORTH sind fast immer interaktive Systeme, in denen Sie einen gerade entwickelten Gedankengang sofort überprüfen und verwirklichen können. Das Auffälligste an der volksFORTH-Oberfläche ist die inverse Statuszeile in der unteren Bildschirmzeile, die sich mit **status off** aus- und mit **status on** wieder einschalten läßt.

Diese Statuszeile zeigt von links nach rechts folgende Informationen, wobei ¦ für "oder" steht:

<2¦8¦10¦16> die zur Zeit gültige Zahlenbasis (dezimal)

s <xx> nennt die Anzahl der Zahlenwerte, die zum Verarbeiten bereitliegen

Dic <xxxx> nennt den freien Speicherplatz

Scr <xx> ist die Nummer des aktuellen Quelltextblocks

A:¦C: gibt das aktuelle Laufwerk an

<name>.<ext> zeigt den Namen der Date, die gerade bearbeitet wird. Dateien haben im MSDOS sowohl einen Namen <name> als auch eine dreibuchstabige Kennung, die Extension <ext>, wobei auch Dateien ohne Extension angelegt werden können.

FORTH FORTH FORTH zeigt die aktuelle Suchreihenfolge gemäß dem Vokabularkonzept.
Ein Beispiel dafür sind die Assembler-Befehle:
Diese befinden sich in einem Vokabular namens **ASSEMBLER** und **assembler words** zeigt Ihnen den Befehlsvorrat des Assemblers an. Achten Sie bitte auf die rechte Seite der Statuszeile, wo jetzt

assembler forth forth

zu sehen ist. Da Sie aber jetzt - noch - keine Assembler-Befehle einsetzen wollen, schalten Sie bitte mit **forth** die Suchlaufpriorität wieder um. Die Statuszeile zeigt wieder das gewohnte

forth forth forth.

Zur Orientierung im Arbeitssystem stellt das volksFORTH einige standardkonforme Wörter zur Verfügung:

words zeigt Ihnen die Befehlsliste von FORTH, die verfügbaren Wörter. Diese Liste stoppt bei einem Tastendruck mit der Ausgabe oder bricht bei einem <ESC> ab.

files zeigt alle im System angelegten logischen Datei-Variablen, die zugehörigen handle Nummern, Datum und Uhrzeit des letzten Zugriffs und ihre entsprechenden physikalischen DOS-Dateien. Eine solche FORTH-Datei wird allein durch die Nennung ihres Namens angemeldet. Die MSDOS-Dateien im Directory werden mit **dir** angezeigt.

path informiert über eine vollständige Pfadunterstützung nach dem MSDOS-Prinzip, allerdings vollkommen unabhängig davon.
Ist kein Suchpfad gesetzt, so gibt path nichts aus.

order beschreibt die Suchreihenfolge in den Befehlsverzeichnissen (Vokabular).

vocs nennt alle diese Unterverzeichnisse (vocabularies).

2.3 Arbeiten mit Programm- und Datenfiles

Um überhaupt Programmtexte (Quelltexte) schreiben zu können, brauchen Sie eine Datei, die diese Programmtexte aufnimmt. Diese Datei muß zur Bearbeitung angemeldet werden.

volksFORTH geht nach dem Systemstart erst einmal von der Datei VOLKS4TH.SYS als aktueller Datei aus. VOLKS4TH.SYS ist aber die Steuerdatei, aus der Ihr FORTH-System aufgebaut wurde; deshalb deklarieren Sie die Datei, die Sie bearbeiten wollen, mit dieser Befehlsfolge:

```
use <name>.<ext>
```

Als Beispiel wird die Datei test.scr mit `use test.scr` angemeldet.
Möchten Sie allerdings eine vollkommen neue Datei für Ihre Programme benutzen, so überlegen Sie sich einen Namen <name>, eine Kennung <extension> und eine vernünftige Größe in KByte. Anschließend geben Sie ein:

```
makefile <name>.<ext>  <Größenangabe> more
```

Daraufhin wird die Datei auf dem Laufwerk angelegt und zum Bearbeiten angemeldet. Ein **use** ist danach nicht mehr notwendig.

2.4 Der Editor

Zum Bearbeiten von Quelltext-Blöcken enthalten ältere FORTH-Systeme meist Editoren, die diesen Namen höchstens zu einer Zeit verdient haben, als man noch die Bits einzeln mit Lochzange und Streifenleser an die Hardware übermitteln mußte. Demgegenüber bot ein Editor, mit dem man jeweils eine ganze Zeile bearbeiten kann, sicher schon einigen Komfort. Da man jedoch mit solch einem Zeileneditor heute keinen Staat mehr machen und erst recht nicht mit anderen Sprachen konkurrieren kann, können Sie im volksFORTH selbstverständlich mit einem komfortablen Fullscreen-Editor arbeiten.

In diesem Editor kann man zwei Dateien gleichzeitig bearbeiten:
Eine Vordergrund-Datei, das aktuelle isfile und ein Hintergrundfile fromfile . Daher werden im Editor zwei Dateinamen angezeigt. Das Wort use meldet eine Datei automatisch sowohl als isfile als auch als fromfile an, so daß sich Verschiebe- und Kopieroperationen nur auf diese eine Datei beziehen.

Im Editor wird immer ein FORTH-Screen - also 1Ø24 Bytes - in der üblichen Aufteilung in 16 Zeilen mit je 64 Spalten dargestellt.
Es gibt einen Zeichen- und einen Zeilenspeicher. Damit lassen sich Zeichen bzw. Zeilen innerhalb eines Screens oder auch zwischen zwei Screens bewegen oder kopieren. Dabei wird verhindert, daß versehentlich Text verloren geht, indem Funktionen nicht ausgeführt werden, wenn dadurch Zeichen nach unten oder zur Seite aus dem Bildschirm geschoben würden.

2.4.1 HELP und VIEW

Der Editor unterstützt das 'Shadow-Konzept'. Zu jedem Quelltext-Screen gibt es einen Kommentar-Screen. Dieser erhöht die Lesbarkeit von FORTH-Programmen erheblich, Sie wissen ja, guter FORTH-Stil ist selbstdokumentierend ! Auf den Tastendruck CRTL-F9 stellt der Editor den Kommentar-Screen zur Verfügung. So können Kommentare 'deckungsgleich' zu den Quelltexten angefertigt werden. Dieses shadow Konzept wird auch bei dem Wort **help** ausgenutzt, das zu einem Wort einen erklärenden Text ausgibt. Dieses Wort wird so eingesetzt:

```
help <name>
```

HELP zeigt natürlich nur dann korrekt eine Erklärung an, wenn ein entsprechender Text auf einem shadow screen vorhanden ist.

Häufig möchte man sich auch die Definition eines Wortes ansehen, um z.B. den Stackkommentar oder die genaue Arbeitsweise nachzulesen. Dafür gibt es das Kommando

```
view <name>
```

Damit wird der Screen - und natürlich auch das File - aufgerufen, auf dem <name> definiert wurde. Dieses Verfahren ersetzt (fast) einen Decompiler, weil es

natürlich sehr viel bequemer ist und Ihnen ja auch sämtliche Quelltexte des Systems zur Verfügung stehen.

Werfen Sie doch bitte mit view u? einen Blick auf das Wort, das Ihnen den Inhalt einer Variablen ausgibt. Benutzen Sie

```
fix <name>
```

(engl. = reparieren), so wird zugleich der Editor zugeschaltet, so daß Sie sofort gezielt Änderungen im Quelltext vornehmen können. Im Editor werden Sie zuerst nach einer Eingabe gefragt "Enter your id:", die Sie einfach mit dem Druck auf die <cr>-Taste beantworten.

Dann befinden Sie sich im Editor, mit dem man die Quelltextblöcke bearbeitet. Der Cursor steht hinter dem gefundenen Wort u? . Nun können Sie mit PgUp oder PgDn in den Screens blättern oder mit <ESC> aus dem Editor zum FORTH zurückkehren.

Natürlich müssen für **view** , **fix** und **help** die entsprechenden Files auf den Laufwerken 'griffbereit' sein, sonst erscheint eine Fehlermeldung. Die VIEW-Funktion steht auch innerhalb des Editors zur Verfügung, man kann dann mit dem Tastendruck CTRL-F das rechts vom Cursor stehende Wort anfordern. Dies ist insbesondere nützlich, wenn man eine Definition aus einem anderen File übernehmen möchte oder nicht mehr sicher ist, wie der Stackkommentar eines Wortes lautet.

2.4.2 Öffnen und Editieren eines Files

Nun aber zum Erstellen und Ändern von Quelltexten für Programme - eine schöne Definition von Editieren. Sie haben sich bestimmt eine Datei test.scr wie oben beschrieben mit

```
makefile test.scr  6 more
```

angelegt. Damit haben Sie ein File namens TEST.SCR mit einer Länge von 6 Blöcken (6144 Byte = 6KB), bestehend aus den Screens Ø bis 5 ; davon sind die Nummern Ø bis 2 für Quelltexte, die anderen für Kommentare bestimmt.

Um in den Editor zu gelangen gibt es drei Möglichkeiten:

```
            <screen#> edit
oder        <scr#> l
```

ruft den Screen mit der Nummer <scr#> auf. Hat man bereits editiert, ruft

```
v
```

den zuletzt bearbeiteten Screen wieder auf. Dies ist der zuletzt editierte oder aber, und das ist sehr hilfreich, derjenige, der einen Abbruch beim Kompilieren verursacht hat. volksFORTH bricht - wie die meisten modernen Systeme - bei einem Programmfehler während des Kompilierens ab und markiert die Stelle, wo der Fehler auftrat. Dann brauchen Sie nur **v** einzugeben. Der fehlerhafte Screen wird in den Editor geladen und der Cursor steht hinter dem Wort, das den Abbruch des Kompilierens verursachte.

Als Beispiel soll der Block #1 editiert werden:

```
1 edit
```

Sie werden nach der Eingabe des dreibuchstabigen Kürzels gefragt. Dieses Kürzel wird dann rechts oben in den Screen eingetragen und gibt die programmer's id wieder. Wenn Sie nichts eingeben möchten, drücken Sie bitte nur <cr>. Zum Benutzen dieser Blöcke gibt es noch einige Vereinbarungen, von denen ich hier zwei nennen möchte:

1. wird der Block Nr.Ø nie !! für Programmtexte benutzt - dort finden sich meist Erklärungen und Hinweise zum Programm und zum Autor.
2. In die Zeile Ø eines Blockes wird immer ein Kommentar eingetragen, der mit \ (skip line) eingeleitet wird. Dieser Backslash sorgt dafür, daß die nachfolgende Zeile als Kommentar überlesen wird. In diese Zeile Ø schreibt man oft die Namen der Wörter, die im Block definiert werden.

2.4.3 Tastenbelegung des Editors

Beim Editieren stehen Ihnen folgende Funktionen zur Verfügung:

F1	gibt Hilfestellung für den Editor
ESC	verläßt den Editor mit dem sofortigen Abspeichern der Änderungen.
CTRL-U	(undo) macht alle Änderungen rückgängig, die noch nicht auf Disk zurückgeschrieben wurden.
CTRL-E	verläßt den Editor ohne sofortiges Abspeichern.
CTRL-F	(fix) sucht das Wort rechts vom Cursor, ohne den Editor zu verlassen.
CTRL-L	(showload) lädt den Screen ab Cursorposition.
CTRL-N	fügt an Cursorposition eine neue Zeile ein.
CTRL-PgDn	splittet innerhalb einer Zeile diese Zeile.
CTRL-S	(Scr#) legt die Nummer des gerade editierten Screens auf dem Stack ab; z.B. für ein folgendes load oder plist.
CTRL-Y	löscht die Zeile an Cursorposition
CTRL-PgUp	fügt innerhalb einer Zeile den rechten Teil der unteren Zeile an die obere Zeile (join).
TAB	bewegt den Cursor einen großen TABulator vor.
SHIFT TAB	bewegt den Cursor einen kleinen TABulator zurück.
F2	(suchen/ersetzen) erwartet eine Zeichenkette, die gesucht werden soll und eine Zeichenkette, die statt dessen eingefügt werden soll. Wird eine Übereinstimmung gefunden, kann man mit R (replace) die gefundene Zeichenkette ersetzen, mit <cr> den Suchvorgang abbrechen oder mit jeder anderen Taste die nächste Übereinstimmung suchen.

Auf diese Weise kann man die Quelltexte auch nach einer Zeichenkette durchsuchen lassen. Als Ersatz-String wird dann <CR> eingegeben.

F3 bringt eine Zeile in den Zeilenpuffer und löscht sie im Screen.

F5 bringt die Kopie einer Zeile in den Zeilenpuffer.

F7 fügt die Zeile aus dem Zeilenpuffer in den Screen ein.

F4 wie F3, nur für ein einzelnes Zeichen.

F6 wie F5, jedoch für ein einzelnes Zeichen.

F8 entspricht F7, bezogen auf ein Zeichen.

F9 vertauscht die aktuelle Datei (isfile) mit der Hintergrunddatei (fromfile). Erneutes F9-Drücken vertauscht erneut und stellt damit den alten Zustand wieder her. Diese Funktion ist dann sinnvoll, wenn Sie eine Datei bearbeiten, sich zwischendurch aber mit CRTL-F ein Wort anzeigen lassen. Dann stellt F9 (=fswap) die alte Dateiverteilung wieder her, die sich durch das fix geändert hat.

SHIFT F9 schaltet auf die Kommentartexte (shadowscreens) um und beim nächsten Drücken wieder zurück.

F1Ø legt den aktuellen Screen kurz beiseite, um ihn dann mit einem Druck auf F9 wieder bearbeiten zu können. Sollte Ihnen das volksFORTH eine der Kopierfunktionen copy oder convey mit der Meldung TARGET BLOCK NOT EMPTY verweigern, weil isfile und fromfile unterschiedlich sind, so sorgt F1Ø wieder für klare Verhältnisse.

Noch ein Hinweis zum Editor:

Der Editor unterstützt nur das Kopieren von Zeilen. Man kann auf diese Art auch Screens kopieren, aber beim gelegentlich erforderlichen Einfügen von Screens in der Mitte eines Files ist das etwas mühselig. Zum Kopieren ganzer Screens innerhalb eines Files oder von einem File in ein anderes werden im volksFORTH83 die Worte COPY und CONVEY verwendet.

2.4.4 Beispiel: CLS

Für Ihr erstes Programm tragen Sie nun bitte den folgenden Quelltext in Screen 1 ein:

```
\ CLS löscht den gesamten Bildschirm
: cls   ( -- )  full page ;
```

Nun drücken Sie SHIFT-F9 und tragen einen Kommentar in diesen Screen ein, wobei die Erklärung zu CLS in der gleichen Zeile, wie die Definition des Quelltextes, z.B.:

```
\\ CLS

CLS  löscht den gesamten Bildschirm, indem es auf die Worte
     full (Bildschirmfenster auf volle Größe) und PAGE (Bild
     schirmfenster löschen)  zurückgreift.
```

Ein nochmaliges SHIFT-F9 bringt Sie wieder zum Quelltext (source code) zurück.. Damit bekommen Sie diesen erklärenden Text nach dem Kompilieren mit help cls angezeigt.

Wenn Sie einen Block vollgeschrieben haben, blättern Sie nur mit PgUp vor oder mit PgDn zurück zum nächsten Block. Sind Sie mit Ihrem Programm zufrieden, so drücken Sie ESC ; dann wird der geänderte Screen sofort abgespeichert.

Danach werden Sie wieder etwas bemerken: Der Bildschirm arbeitet nicht mehr korrekt, es bleiben oben Zeilen stehen, die nicht scrollen.
Das ist richtig, damit der zuletzt bearbeitete Quelltext nicht nach oben wegscrollt. Um nicht weitere zwei Zeilen des Bildschirms zu verlieren, hat das Fenster, in dem Sie gerade arbeiten, keinen Rahmen. Wie man mit Rahmen und Fenstern arbeitet, wird im Editor vorgeführt, der wie alle anderen Systemteile im Quelltext vorliegt.

Um sich in Zukunft schnell aus der mißlichen Lage mit der reduzierten Bildschirmgröße zu befreien, benutzen Sie Ihr erstes selbstgeschriebenes Programm cls . Denn die Eingabe von full stellt Ihnen wieder die gesamten Bildschirmfläche zur Verfügung, wogegen page nur das aktuelle Fenster löscht.
In volks4TH wird das Kompilieren - wie in den meisten FORTH-Systemen - über <scr#> load durchgeführt:

```
1 load
```

kompiliert Ihr Wort **CLS** mit für 'C'- oder Pascal-Programmierer unvorstellbarer Geschwindigkeit ins FORTH. Damit steht Ihr Mini-Programm jetzt zur Ausführung bereit. Zugleich erhalten Sie die Meldung, daß ein Wort namens **CLS** bereits besteht: "CLS exists" . Dies hat nur die Konsequenz, daß nach einer Redefinition das alte Wort gleichen Namens nicht mehr zugegriffen werden kann. Eine Möglichkeit, diese Namensgleichheit mit einem bereits existierenden Wort zu vermeiden, wäre der Einsatz eines Vokabulares.
Geben Sie einmal **words** ein, dann werden Sie feststellen, daß Ihr neues Wort **CLS** ganz oben im Dictionary steht (drücken Sie die <ESC>-Taste, um die Ausgabe von **words** abzubrechen oder irgendeine andere Taste, um sie anzuhalten).
Um das Ergebnis Ihres ersten Programmierversuchs zu überprüfen, geben Sie nun ein:

```
cls
```

Und siehe da, der gesamte Bildschirm wird dunkel. Schöner wäre es allerdings, wenn Ihr Programm seine Arbeit mit einer Meldung beenden würde. Um dies zu ändern, werfen Sie bitte erst einmal das alte Wort **CLS** weg:

```
forget cls
```

Bitte kontrollieren Sie mit **words** , ob **CLS** wirklich vergessen wurde. Rufen Sie dann erneut den Editor mit **v** auf. Nun benutzen Sie das Wort für den Beginn einer Zeichenkette **."** , das Wort für ihr Ende **"** und das Wort für einen Zeilenvorschub **cr** . Ihr Screen 1 sieht dann so aus :

```
\ CLS löscht den Bildschirm mit Meldung
```

```
: cls   ( -- )
   full page
   ." Bildschirm ordnungsgemäß gelöscht!" cr ;
```

Nach dem ." muß ein Leerzeichen stehen. Denn FORTH benutzt standardmäßig das Leerzeichen als Trennzeichen zwischen einzelnen Worten, so daß dies Leerzeichen nicht mit zur Zeichenkette (string) zählt. Dann verlassen Sie den Editor und kompilieren Sie Ihr Programm wie gehabt und starten es. Die Änderung erweist sich als erfolgreich, und Sie haben gelernt, wie einfach in FORTH das Schreiben, Austesten und Ändern von Programmteilen ist.
Eine große Hilfe sowohl für den Programmierer als auch für den späteren Benutzer sind Informationen darüber, was gerade geladen wird und was schon kompiliert wurde. Fügt man

```
cr  .( Funktion installiert )
```

ein, so werden während des Ladens diese Meldungen ausgegeben. Das Wort .(leitet einen Kommentar im Interpreter ein, die schließende Klammer) beendet ihn und cr ist natürlich für den Zeilenvorschub, das carriage return, verantwortlich.

2.4.5 Compilieren im Editor - Showload

Eine Besonderheit von volksFORTH ist, daß selbst im Editor Funktionen des Interpreters/Compilers zur Verfügung stehen. Dieses Interpretieren und Kompilieren im Editor nennt sich Showload.

CTRL-F	(fix) sucht und zeigt das Wort rechts vom Cursor, ohne den Editor zu verlassen.
CTRL-L	(showload) lädt den Screen ab Cursorposition.

Um zu sehen, was diese Showload-Funktion leistet, geben Sie nun bitte folgenden Screen ein:

```
\ Ein Test für das showload
  : inc  1+ ;
  : dec  1- ;
\\
  15 inc .
  15 inc dec .
  15 15 +  2 spaces .
```

Dieser Screen soll jetzt im Editor kompiliert und interpretiert werden !!
Dazu setzen Sie bitte den Cursor durch die Taste CTRL-home in die erste Zeile auf das Zeichen \ (skip line). Drücken Sie nun CTRL-L zum Laden des Screens. volksFORTH kompiliert nun bis zu der Zeile, die mit \\ (skip screen) beginnt. Die Wörter inc und dec sind dem System jetzt bekannt und können benutzt werden.
Anschließend bewegen Sie den Cursor hinter das \\ und drücken CTRL-L zum weiteren Interpretieren im Editor. Sofort sehen Sie die Ausgaben an der entsprechenden Stelle im Editor erscheinen. Dabei bleibt der Inhalt des Screens selbst-

verständlich unversehrt - verlassen Sie den Editor mit ESC und sehen Sie sich den Inhalt den Screens mit <scr#> list an; Sie sehen nur die Anweisungen, aber nicht mehr die Ausgaben vor sich.
Eine typische Anwendung dieses showload wäre das Neukompilieren eines Wortes nach einer kleinen Änderung oder das interaktive Hinzufügen von Wörtern, die man gerade mal braucht.

2.5 Erstellen einer Applikation

Sie wollen Ihr 'Programm' nun als eigenständige Anwendung abspeichern. Dazu erweitern Sie es zunächst ein klein wenig (Editor mit v aufrufen.) FÜgen Sie nun noch folgende Definition in einer neuen Zeile hinzu:

```
: runalone    cls bye  ;
```

RUNALONE führt zuerst CLS aus und kehrt dann zum Betriebssystem zurück. Kompilieren Sie nun erneut, wobei Sie die Meldung erhalten "CLS exists". Sie führen **RUNALONE** aber nicht aus, sonst würden Sie FORTH ja verlassen (bye).
Das Problem besteht vielmehr darin, das System so abzuspeichern, daß es gleich nach dem Laden RUNALONE ausführt und sonst gar nichts. volksFORTH83 ist an zwei Stellen für solche Zwecke vorbereitet. In den Worten COLD und RESTART befinden sich zwei 'deferred words' namens 'COLD bzw. 'RESTART , die im Normalfall nichts tun, vom Anwender aber nachträglich verändert werden können.
Sie benutzen hier 'COLD, um auch schon die Startmeldung zu unterbinden. Geben Sie also ein

```
' runalone Is 'cold
```

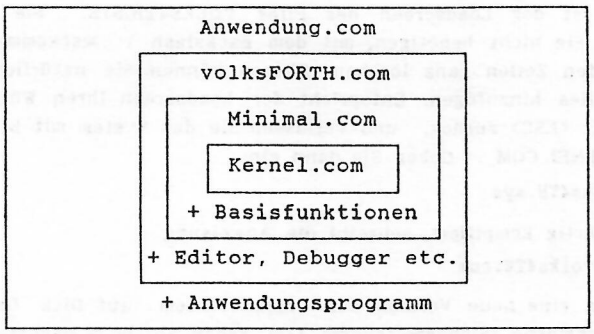
und speichern Sie das Ganze mit

```
savesystem cls.com
```

auf Disk zurück. Sie haben Ihre erste Applikation erstellt, die Sie von MSDOS-Ebene aus mit CLS aufrufen können !

Etwas enttäuschend ist es aber schon. Das angeblich so kompakte FORTH benötigt über 20KByte, um eine so lächerliche Funktion auszuführen ?? Da stimmt doch etwas nicht. Natürlich, es wurden ja eine Reihe von Systemteilen mit abgespeichert, vom Fileinterface über den Assembler, den Editor usw., die für unser Programm überhaupt nicht benötigt werden.

Um dieses und ähnliche Probleme zu lösen, gibt es das File KERNEL.COM. Dieses Programm enthält nur den Systemkern und das Fileinterface und entspricht damit der Laufzeit-Bilbliothek (runtime library) anderer Sprachen.



Laden Sie also KERNEL.COM und kompilieren Sie Ihre Applikation mit

```
include multi.vid
include test.scr
```

Das vorherige Laden von MULTI.VID ist nötig, weil FORTH-Systeme selten über Linker verfügen; wird MULTI.VID nicht vorher geladen, so ist dem FORTH-System das Wort full unbekannt. Dann wie gehabt RUNALONE in 'COLD eintragen und das System auf Disk zurückspeichern. Von der MS-DOS Ebene aus läßt sich diese Programmerstellung mit

```
kernel include mulit.vid include test.scr
```

durchführen, wobei diese beiden Zeilen

```
' runalone Is 'cold
savesystem dark.com
```

die letzten Anweisungen auf Ihrem Screen sind. Damit wird der FORTH-Interpreter angewiesen, die fertige Anwendung DARK.com auf Disk zu speichern.
Sie haben jetzt eine verhältnismäßig kompakte Version vorliegen. Natürlich ließe sich auch diese noch erheblich kürzen, aber dafür bräuchten Sie einen Target-Compiler, mit dem Sie nur noch die wirklich benötigten Systemteile selektiv aus dem Quelltext zusammenstellen könnten. Mit der beschriebenen Methode lassen sich aber auch größere Programme kompilieren und als Stand-alone-Applikationen abspeichern.

2.6 Das Erstellen eines eigenen FORTH-Systems

Das File VOLKS4TH.COM ist als Arbeitsversion gedacht.

Es enthält alle wichtigen Systemteile wie Editor, Printer-Interface, Tools, Decompiler, Tracer usw. Sollte Ihnen die Zusammenstellung nicht gefallen, können Sie sich jederzeit ein Ihren speziellen Wünschen angepaßtes System zusammenstellen.

Schlüssel dazu ist der Loadscreen des Files VOLKS4TH.SYS . Sie können dort Systemteile, die Sie nicht benötigen, mit dem Backslash \ wegkommentieren oder die entsprechenden Zeilen ganz löschen. Ebenso können Sie natürlich dem Loadscreen eigene Files hinzufügen. Entspricht der Loadscreen Ihren Wünschen, speichern Sie ihn mit <ESC> zurück, und verlassen Sie das System mit BYE. Laden Sie nun das File KERNEL.COM . Geben Sie dann ein:

```
include volks4TH.sys
```

Ist das System fertig kompiliert, schreibt die Anweisung

```
savesystem volks4TH.com
```

des Load-Screens eine neue Version von volks4TH.com auf Disk. Damit wird Ihr altes File überschrieben (Sicherheitskopie !!!), sodaß Sie beim nächsten Laden von volks4TH.com Ihr eigenes System erhalten.

Natürlich können Sie 'Ihr' System auch unter einem anderen Namen abspeichern. Ebenso können Sie Systemvoreinstellungen ändern. Unsere Arbeitsversion arbeitet - voreingestellt - neuerdings im dezimalen Zahlensystem. Natürlich können Sie mit hex auf Hexadezimalsystem umstellen; wir halten das für sehr viel sinnvoller, weil vor allem Speicheradressen im Dezimalsystem kaum etwas aussagen (oder wissen Sie, ob Speicherstelle 978584 im Bildschirmspeicher liegt oder nicht ?). Wollen Sie bereits unmittelbar nach dem Laden im Hexadezimalsystem arbeiten, können Sie sich dies mit SAVESYSTEM abspeichern, indem Sie von der FORTH-Kommandozeile aus savesystem volks4TH.com eingeben.

Im Übrigen empfehlen wir bei allen Zahlen über 9 dringend die Benutzung der sogenannten Präfixe:

```
$ für Hexadezimal-,
& für Dezimal- und
% für Binärzahlen.
```

Man vermeidet so, daß irgendwelche Files nicht - oder noch schlimmer, falsch - kompiliert werden, weil man gerade im anderen Zahlensystem ist. Außerdem ist es möglich, hexadezimale und dezimale Zahlen beliebig zu kombinieren, je nachdem, was gerade sinnvoller ist. In den Quelltexten finden Sie genug entsprechende Beispiele.

Besonders schnell und komfortabel arbeitet volksFORTH natürlich, wenn alle Teile des Systems auf einer Festplatte abgelegt sind. Sie sollten dafür ein eigenes Directory einrichten und PATH und DIR entsprechend einstellen.
Auch die Arbeit mit einer RAM-Disk ist prinzipiell möglich, allerdings nicht sehr zu empfehlen. FORTH ist sehr maschinennah und Systemabstürze daher vor allem zu Anfang nicht so ganz auszuschließen.
Das Ausdrucken der Quelltexte des Systems ist sicher sinnvoll, um Beispiele für den Umgang mit volksFORTH83 zu sehen. So stellen z.B. der Screen-Editor und der Kommando-Editor vollständige Anwendungen dar, die im Quelltext vorliegen.
Welche Files sich im Einzelnen auf Ihren Disketten befinden und ob sie Kommen-

tarscreens enthalten, steht im File README.DOC. Zunächst müssen Sie das Printer-Interface hinzuladen, falls es nicht schon vorhanden ist. Reagiert Ihr volksFORTH83 auf die Eingabe von PRINTER mit einem ? , so ist das Printerinterface nicht vorhanden.

2.7 Ausdrucken von Screens

Sollte in Ihrem System kein Druckerinterface vorhanden sein, so laden Sie es von der FORTH-Kommandozeile aus mit

```
include <Druckername>.prn
```

nach. Sollte Ihr Drucker in der Liste nicht erscheinen, so benutzen Sie statt dessen graphic.prn oder epson.prn . Die meisten Drucker können als IBM-Graphic-Printer oder als EPSON FX/LX-Drucker arbeiten.
Im Printer-Interface sind einige Worte zur Ausgabe eines formatierten Listings enthalten. PTHRU druckt einen Bereich von Screens, jeweils 6 Screens auf einer DIN A4 Seite in komprimierter Schrift. Ganz ähnlich arbeitet das Wort DOCUMENT , jedoch wird bei diesem Wort neben einen Quelltextscreen der zugehörige Shadowscreen gedruckt. LISTING druckt ein ganzes File so aus, man erhält so ein übersichtliches Listing eines Files mit ausführlichen Kommentaren.

Glossar

pthru (von bis --)
druckt die angegebenen Blöcke immer zu sechst auf einer Seite aus.

document (von bis --)
arbeitet wie pthru , druckt aber jeweils drei Quelltextblöcke und drei Kommentarblöcke auf einer Seite aus.

listing (--)
erstellt ein Listing der gesamten Datei, indem jeweils drei Quelltext-Blöcke und drei Kommentar-Blöcke auf einer Seite ausgedruckt werden.

plist (scr# --)
druckt einen angegebenen Block auf dem Drucker aus.

scr (-- addr)
ist eine Variable, die die Nummer des gerade editierten Screens enthält.
Vergl. r# , list, (error

r# (-- addr)
ist eine Variable, die den Abstand des gerade editierten Zeichens von Anfang des gerade editierten Screens enthält.

2.8 Druckeranpassung

Die Druckeranpassung das Arbeitssystems wird im File VOLKS4TH.SYS durch die Zeile

```
include <printer>.prn
```

vorgenommen. In dieser Anpassung sind - zusätzlich zu den reinen Ausgaberoutinen - eine Reihe nützlicher Worte enthalten, mit denen die Druckersteuerung sehr komfortabel vorgenommen werden kann.
Im Arbeitssystem ist das Printerinterface bereits enthalten. Müssen Sie Änderungen vornehmen, können Sie mit dem Editor den Loadscreen von VOLKS4TH.SYS ändern und sich ein neues Arbeitssystem zusammenstellen mit:

```
kernel include volks4TH.sys
```

Sie können natürlich auch den Loadscreen in seiner jetzigen Fassung benutzen und das Printer-Interface jedesmal 'von Hand' mit

```
include <printer>.prn
```

nachladen.

Leider sind im Moment im volksFORTH noch deutsche und englische Fehlermeldungen gemischt und die help Funktion zeigt Ihnen einen erklärenden Text nur, wenn dieser vorhanden ist.

Die wichtigsten Befehle noch einmal im Überblick:

status	steuert die Status-Zeile
words	zeigt die gerade verfügbaren Befehle an
files	zeigt die angemeldeten Dateien
path	verändert oder nennt den Datei-Suchpfad
order	listet die Suchreihenfolge der Befehlsgruppen auf
vocs	nennt alle verfügbaren Befehlsgruppen
view	zeigt und
fix	editiert den Quelltext eines bestimmten Wortes
help	zeigt - wenn vorhanden - den Kommentartext eines Wort
full	schaltet das Bildschirmfenster auf die volle Größe
page	löscht das aktuelle Fenster
index	- nicht resident - zeigt den Inhalt einer Block-Datei
list	zeigt den Inhalt eines Screens.

include lädt eine ganze Befehlsgruppe oder einen Programteil

Nun sollten Sie bereits zu einem begleitenden Buch zu greifen. Denn volksFORTH ist ein komplexes multitaskingfähiges Entwicklungssystem, das man nicht von heute auf morgen beherrscht.

[1] Leo Brodie — Programmieren in FORTH (Hanser Verlag)

[2] Leo Brodie — In FORTH denken (Hanser Verlag)

[3] R. Zech — FORTH 83 (Franzis Verlag)

[4] H.-W. Beilstein — Wie man in FORTH programmiert (Chip Wissen)

3. Arithmetik

3.1 Stacknotation

Im folgenden werden hauptsächlich Worte in ihrer Einzelfunktion beschrieben. In dieser Form der Beschreibung, die Sie bereits kennengelernt haben, wird die Wirkung eines Wortes auf den Stack in Klammern angegeben und zwar in folgender Form:

(vorher -- nachher)

vorher : Werte auf dem Stack vor Ausführung des Wortes
nachher : Werte auf dem Stack nach Ausführung des Wortes

In dieser Notation wird das oberste Element des Stacks (tos) immer ganz rechts geschrieben. Sofern nicht anders angegeben, beziehen sich alle Stacknotationen auf die spätere Ausführung des Wortes. Bei immediate Worten wird auch die Auswirkung des Wortes auf den Stack während der Kompilierung angegeben. Worte werden ferner durch folgende Symbole gekennzeichnet:

C	Dieses Wort kann nur während der Kompilation einer :-Definition benutzt werden.
I	Dieses Wort ist ein immediate Wort, das auch im kompilierenden Zustand ausgeführt wird.
83	Dieses Wort wird im 83-Standard definiert und muß auf allen Standardsystemen äquivalent funktionieren.
U	Kennzeichnet eine Uservariable.

Weicht die Aussprache eines Wortes von der natürlichen englischen Aussprache ab, so wird sie in Anführungszeichen angegeben. Gelegentlich folgt auch eine deutsche Übersetzung.
Die Namen der Stackparameter folgen, sofern nicht suggestive Bezeichnungen gewählt wurden, dem nachstehendem Schema. Die Bezeichnungen können mit einer nachfolgenden Ziffer versehen sein.

Stack-notation	Zahlentyp	Wertebereich in Dezimal	minimale Feldbreite
flag	logischer Wert	Ø=falsch, sonst=true	16 Bit
true (tf)	logischer Wert	-1 (als Ergebnis)	16 Bit
false (ff)	logischer Wert	Ø	16 Bit
b	Bit	Ø..1	1 Bit
char	Zeichen	Ø..127 (Ø..256)	7(8Bit)

8b	8 beliebige Bits	nicht anwendbar	8
16b	16 beliebige Bits	nicht anwendbar	16
n	Zahl,bewertete Bits	-32768..32767	16
+n	positive Zahl	Ø..32767	16
u	vorzeichenlose Zahl	Ø..65535	16
w	Zahl, n oder u	-32768..65535	16
addr	Adresse, wie u	Ø..65535	16
32b	32 beliebige Bits	nicht anwendbar	32
d	doppelt genaue Zahl	-2,147,483,648... 2,147,483,647	32
+d	pos. doppelte Zahl	Ø..2,147,483,647	32
ud	vorzeichenlose doppelt genaue Zahl	Ø..4,294,967,295	32
sys	systemabhängige Werte	nicht anwendbar	nicht anwendbar

3.2 Arithmetische Funktionen

-1 (-- -1)
Ø (-- Ø)
1 (-- 1)
2 (-- 2)
3 (-- 3)
4 (-- 4)

Oft benutzte Zahlenwerte wurden zu Konstanten gemacht. Definiert in der Form :

```
n Constant n
```

Dadurch wird Speicherplatz eingespart und die Ausführungszeit verkürzt.

1+ (w1 -- w2) 83 "one-plus"

w2 ist das Ergebnis von Eins plus w1. Die Operation 1 + wirkt genauso.

1- (w1 -- w2) 83 "one-minus"

w2 ist das Ergebnis von w1 minus Eins. Die Operation 1 - wirkt genauso.

2+ (w1 -- w2) 83 "two-plus"

w2 ist das Ergebnis von w1 plus Zwei. Die Operation 2 + wirkt genauso.

2- (w1 -- w2) 83 "two-minus"

w2 ist das Ergebnis von w1 minus Zwei. Die Operation 2 - wirkt genauso.

2* (w1 -- w2) "two-times"
w1 wird um ein Bit nach links geschoben und das ergibt w2. In das niederwertigste Bit wird eine Null geschrieben. Die Operation 2 * wirkt genauso.

2/ (n1 -- n2) 83 "two-divide"
n1 wird um ein Bit nach rechts verschoben und das ergibt n2. Das Vorzeichen wird berücksichtigt und bleibt unverändert. Die Operation 2 / wirkt genauso.

3+ (w1 -- w2) "three-plus"
w2 ist das Ergebnis von w1 plus Drei. Die Operation 3 + wirkt genauso.

abs (n -- u) 83 "absolute"
u ist der Betrag von n. Wenn n gleich -32768 ist, hat u denselben Wert wie n. Vergleiche auch Zweierkomplement.

not (w1 -- w2) 83
Jedes Bit von w1 wird einzeln invertiert und das ergibt w2.

negate (n1 -- n2) 83
n2 hat den gleichen Betrag, aber das umgekehrte Vorzeichen von n1. n2 ist gleich der Differenz von Null minus n1.

even (u1 -- u2)
ist im 8086-FORTH ein noop-Befehl ohne Funktion.

max (n1 n2 -- n3) 83 "maximum"
n3 ist die Größere der beiden Werte n1 und n2. Benutzt die Operation > . Die größte Zahl für n1 oder n2 ist 32767.

min (n1 n2 -- n3) 83 "minimum"
n3 ist die Kleinere der beiden Werte n1 und n2. Benutzt die Operation < . Die kleinste Zahl für n1 oder n2 ist -32768.

+ (w1 w2 -- w3) 83 "plus"
w1 und w2 addiert ergibt w3.

- (w1 w2 -- w3) 83 "minus"
w2 von w1 subtrahiert ergibt w3.

***** (w1 w2 -- w3) 83 "times"

Der Wert w1 wird mit w2 multipliziert. w3 sind die niederwertigen 16 Bits des Produktes. Ein Überlauf wird nicht angezeigt.

/ (n1 n2 -- n3) 83 "divide"

n3 ist der Quotient aus der Division von n1 durch den Divisor n2. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768...32767) liegt.

mod (n1 n2 -- n3) 83 "mod"

n3 ist der Rest der Division von n1 durch den Divisor n2. n3 hat daßelbe Vorzeichen wie n2 oder ist Null. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768..32767) liegt.

/mod (n1 n2 -- n3 n4) 83 "divide-mod"

n3 ist der Rest und n4 der Quotient aus der Division von n1 durch den Divisor n2. n3 hat dasselbe Vorzeichen wie n2 oder ist Null. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768..32767) liegt.

***/** (n1 n2 n3 -- n4) 83 "times-divide"

Zuerst wird n1 mit n2 multipliziert und ein 32-bit Zwischenergebnis erzeugt. n4 ist der Quotient aus dem 32-bit Zwischenergebnis und dem Divisior n3. Das Produkt von n1 mal n2 wird als 32-bit Zwischenergebnis dargestellt, um eine größere Genauigkeit gegenüber dem sonst gleichwertigen Ausdruck n1 n2 * n3 / zu erhalten. Eine Fehlerbedingung besteht, wenn der Divisor Null ist, oder der Quotient außerhalb des Intervalls (-32768.. 32767) liegt.

***/mod** (n1 n2 n3 -- n4 n5) 83 "times-divide-mod"

Zuerst wird n1 mit n2 multipliziert und ein 32-bit Zwischenergebnis erzeugt. n4 ist der Rest und n5 der Quotient aus dem 32-bit-Zwischenergebnis und dem Divisor n3. n4 hat das gleiche Vorzeichen wie n3 oder ist Null. Das Produkt von n1 mal n2 wird als 32-bit Zwischenergebnis dargestellt, um eine größere Genauigkeit gegenüber dem sonst gleichwertigen Ausdruck n1 n2 * n3 /mod zu erhalten. Eine Fehlerbedingung besteht, falls der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768...32767) liegt.

u/mod (u1 u2 -- u3 u4) "u-divide-mod"

u3 ist der Rest und u4 der Quotient aus der Division von u1 durch den Divisor u2. Die Zahlen u sind vorzeichenlose 16-Bit Werte (unsigned integer). Eine Fehlerbedingung besteht, wenn der Divisor Null ist.

umax (u1 u2 -- u3) "u-maximum"
u3 ist der Größere der beiden Werte u1 und u2. Benutzt die U> Operation. Die größte Zahl für u1 oder u2 ist 65535.

umin (u1 u2 -- u3) "u-minimum"
u3 ist der Kleinere der beiden Werte u1 und u2. Benutzt die U< Operation. Die kleinste Zahl für u1 oder u2 ist Null.

3.3 Logik und Vergleiche

true (-- -1)
hinterläßt -1 als Zeichen für logisch wahr auf dem Stack.

false (-- Ø)
Hinterläßt Null als Zeichen für logisch-falsch auf dem Stack.

Ø= (w -- flag) 83 "zero-equals"
Wenn w gleich Null ist, ist flag wahr.

Ø<> (n -- flag)
Wenn n verschieden von Null ist, ist flag wahr.

Ø< (n -- flag) 83 "zero-less"
Wenn n kleiner als Null (negativ) ist, ist flag wahr. Dies ist immer dann der Fall, wenn das höchstwertige Bit von n gesetzt ist. Deswegen kann dieser Operator zum Testen dieses Bits benutzt werden.

Ø> (n -- flag) 83 "zero-greater"
Wenn n größer als Null ist, ist flag wahr.

= (w1 w2 -- flag) 83 "equals"
Wenn w1 gleich w2 ist, ist flag wahr.

< (n1 n2 -- flag) 83 "less-than"
Wenn n1 kleiner als n2 ist, ist flag wahr. z.B. -32768 32767 < ist wahr. -32768 Ø < ist wahr.

> (n1 n2 -- flag) 83 "greater-than"
Wenn n1 größer als n2 ist, ist flag wahr.z.B. -32768 32767 > ist falsch. -32768 Ø > ist falsch.

u< (u1 u2 -- flag) 83 "u-less-than"
Wenn u1 kleiner als u2 ist, ist flag wahr. Die Zahlen u sind vorzeichenlose 16-Bit Werte. Wenn Adressen verglichen werden sollen, muß U< benutzt werden, sonst passieren oberhalb von 32K seltsame Dinge !

u> (u1 u2 -- flag) 83 "u-greater-than"
Wenn u1 größer als u2 ist, ist flag wahr. Ansonsten gilt das gleiche wie für U< .

and (w1 w2 -- w3) 83
w1 wird mit w2 bitweise logisch UND verknüpft und das ergibt w3.

or (w1 w2 -- w3) 83
w1 wird mit w2 logisch ODER verknüpft und das ergibt w3.

xor (w1 w2 -- w3) 83 "x-or"
w1 wird mit w2 bitweise logisch EXKLUSIV ODER verknüpft und das ergibt w3.

uwithin (u u1 u2 -- flag)
Wenn u1 kleiner oder gleich u und u kleiner u2 ist (u1<=u<u2), ist flag wahr. Benutzt die U< Operation.

case? (16b1 16b2 -- 16b1 false | true) "case-question"
Vergleicht die beiden Werte 16b1 und 16b2 miteinander. Sind sie gleich, verbleibt TRUE auf dem Stack. Sind sie verschieden, verbleibt FALSE und der darunterliegende Wert 16b1 auf dem Stack. Wird z.B. in der folgenden Form benutzt :

```
key
Ascii a case? IF ... exit  THEN
Ascii b case? IF ... exit  THEN
drop
```

Entspricht dem Ausdruck over = dup IF nip THEN .

3.4 32Bit-Worte

extend (n -- d)
Der Wert n wird auf den doppelt genauen Wert d vorzeichenrichtig erweitert. Benutze für das in der Literatur oft auftretende s>d

```
' extend Alias s>d
```

dabs (d -- ud) 83 "d-absolut"
ud ist der Betrag von d. Wenn d gleich -2.147.483.648 ist, hat ud den selben Wert wie d.

dnegate (d1 -- d2) 83 "d-negate"
d2 hat den gleichen Betrag aber ein anderes Vorzeichen als d1.

d+ (d1 d2 -- d3) 83 "d-plus"
d1 und d2 addiert ergibt d3.

d- (d1 d2 -- d3) "d-minus"
d2 minus d1 ergibt d3.

d* (d1 d2 -- d3) "d-times"
d1 multipliziert mit d2 ergibt d3.

d= (d1 d2 -- flag) "d-equal"
Wenn d1 gleich d2 ist, ist flag wahr.

d< (d1 d2 -- flag) 83 "d-less-than"
Wenn d1 kleiner als d2 ist, ist flag wahr.

dØ= (d -- flag) 83 "d-zero-equals"
Wenn d gleich Null ist, ist flag wahr.

m* (n1 n2 -- d) "m-times"
Der Wert von n1 wird mit n2 multiplizert und d ist das doppelt genaue Produkt.

um* (u1 u2 -- ud) 83 "u-m-times"
Die Werte u1 und u2 werden mulitpliziert und das ergibt das doppelt genaue Produkt ud. UM* ist die anderen multiplizierenden Worten zugrundeliegende Routine.

m/mod (d n1 -- n2 n3) "m-divide-mod"
n2 ist der Rest und n3 der Quotient aus der Division der doppelt genauen Zahl d durch den Divisor n1. Der Rest n2 hat dasselbe Vorzeichen wie n1 oder ist Null. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768..32767) liegt.

ud/mod (ud1 u1 -- u2 ud2) "u-d-divide-mod"
u2 ist der Rest und ud2 der doppelt genaue Quotient aus der Divison der doppelt genauen Zahl ud1 durch den Divisor u1. Die Zahlen u sind vorzeichenlose 16-Bit Werte (unsigned integer). Eine Fehlerbedingung besteht, wenn der Divisor Null ist.

um/mod (ud u1 -- u2 u3) 83 "u-m-divide-mod"
u2 ist der Rest und u3 der Quotient aus der Division von ud durch den Divisor u1. Die Zahlen u sind vorzeichenlose Zahlen. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (Ø..65535) liegt.

3.5 Stack-Operationen

Herkömmliche Programmiersprachen enthalten mehr oder weniger ausgeprägt das Konzept der PROZEDUREN:
Für bestimmte Programmfunktionen notwendige Operatoren werden in benannten Programmteilen zusammengefaßt, um diese Programmfunktionen an mehreren Stellen innerhalb eines Programmes über ihren Namen aktivieren zu können. Da FORTH ohne jede Einschränkung prozedural ist, macht FORTH auch keinen Unterschied zwischen Prozeduren und Funktionen oder seinen Operatoren. Alles wird als ein WORT bezeichnet.
FORTH als Programmier-SPRACHE besteht also aus Wörtern.
Somit können FORTH-Wörter sein:

1. Datenbereiche
2. Algorithmen (Befehle)
3. Programme

Um Prozeduren sinnvoll benutzen zu können, kennen die meisten Sprachen auch PARAMETER:
Dies sind Daten, die einer Prozedur bei ihrem Aufruf zur Bearbeitung übergeben werden. Daten, die ausschließlich innerhalb einer Prozedur benötigt werden, heißen LOKAL zu dieser Prozedur; im Gegensatz dazu nennt man Daten, die außerhalb von bestimmten Prozeduren zur Verfügung stehen und auf die von allen Prozeduren aus mit allen Operatoren zugegriffen werden kann, GLOBAL.

Die erste Möglichkeit der Parameterübergabe zwischen Prozeduren ist die Vereinbarung von benannten GLOBALEN Variablen. Diese globalen Variablen sind für die gesamte Laufzeit des Programmes statisch existent und können von allen Prozeduren manipuliert werden.

Eine andere Möglichkeit der Parameterübergabe besteht im Einrichten eines Speicherbereiches, in dem während das Aufrufes eines Wortes namentlich benannte Parameter dynamisch verwaltetet werden.
Diesen Mechanismus für benannte lokale Variable stellt Standard-FORTH nicht zur

Verfügung, weil die Organisation dieser lokalen Variablen mit einem Verlust sowohl der lokalen Daten nach der Ausführung des Wortes als auch einem Verlust in der Ausführungsgeschwindigkeit des Wortes verbunden sind. Für das volks4TH wurde in der VD 1/88 eine Implementierung benannter lokaler Variabler vorgestellt.

FORTH benutzt zur gegenseitigen Übergabe von Parametern an Wörter hauptsächlich den STACK, einen bestimmten Speicherbereich, in dem die Wörter ihre Parameter erwarten. Diese Parameter erhalten keine Namen, sondern ihre Interpretation ergibt sich aus der Position innerhalb des Stack-Speicherbereiches. Daraus resultiert die Vielzahl von Operatoren zur Änderung der Stack-Position eines Wertes, für die FORTH berühmt/berüchtigt ist.

Damit deutlich wird, welche und wieviele Parameter ein Wort benötigt, werden allgemein STACK-KOMMENTARE verwendet:

Der öffnenden runden Klammer folgt eine Aufzählung der Parameter. Dabei steht der Parameter, der als oberstes Stack-Element erwartet wird, ganz rechts. Dann folgt ein " -- ", das die Ausführung des Wortes symbolisieren soll. Anschließend wird der Zustand des Stacks nach der Ausführung des Wortes dargestellt, wobei das oberste Stackelement wieder ganz rechts steht. Die schließende runde Klammer beendet den Stack-Kommentar.

Ein Wort **SQRT**, das die Quadratwurzel einer Integerzahl liefert, würde in FORTH so benannt und beschrieben:

```
sqrt  ( number -- sqrt )
```

Wird dieses neue Wort aufgerufen, so werden alle darin enthaltenen Wörter ausgeführt, eventuell bereitgestellte Parameter bearbeitet und daraus resultierende Ergebnisse auf dem Stack übergeben.
Der Aufruf von Prozeduren erfolgt in FORTH implizit durch die Nennung des Namens, ebenso wie auch die Datenübergabe zwischen Wörtern meist implizit erfolgt.

3.5.1 Datenstack-Operationen

drop (16b --) 83
Der Wert 16b wird vom Stack entfernt.

2drop (32b --) 83 "two-drop"
Der Wert 32b wird vom Stack entfernt.

dup (16b -- 16b 16b) 83
Der Wert 16b wird dupliziert.

?dup (16b -- 16b 16b ¦ Ø) 83 "question-dup"
Nur wenn der Wert 16b von Null verschieden ist, wird er verdoppelt.

2dup (32b -- 32b 32b) 83 "two-dup"
Der Wert 32b wird dupliziert.

swap (16b1 16b2 -- 16b2 16b1) 83
Die beiden obersten 16-Bit Werte werden vertauscht.

2swap (32b1 32b2 -- 32b2 32b1) 83 "two-swap"
Die beiden obersten 32-Bit Werte 32b1 und 32b2 werden vertauscht.

nip (16b1 16b2 -- 16b2)
Der Wert 16b1, der unter 16b2 auf dem Stack liegt, wird vom Stack entfernt.

over (16b1 16b2 -- 16b1 16b2 16b1) 83
Der Wert 16b1 wird über 16b2 herüberkopiert.

2over (32b1 32b2 -- 32b1 32b2 31b1) "two-over"
Der Wert 32b1 wird über den Wert 32b2 herüber kopiert.

under (16b1 16b2 -- 16b2 16b1 16b2)
Eine Kopie des obersten Wertes auf dem Stack wird unter dem zweiten Wert eingefügt.

rot (16b1 16b2 16b3 -- 16b2 16b3 16b1) 83
Die drei obersten Werte auf dem Stack werden rotiert, sodaß der unterste zum obersten wird.

-rot (16b1 16b2 16b3 -- 16b3 16b1 16b2) "minus-rot"
Die drei obersten 16b Werte werden rotiert, sodaß der oberste Wert zum Untersten wird. Hebt rot auf.

roll (16bn 16bm..16bØ +n -- 16bm..16bØ 16bn) 83
Das +n-te Glied einer Kette von n Werten wird nach oben auf den Stack gerollt. Dabei wird +n selbst nicht mitgezählt.

-roll (16bn .. 16b1 16bØ +n -- 16bØ 16bn .. 16b1)
Das oberste Glied einer Kette von +n Werten wird an die n-te Position gerollt. Dabei wird +n selbst nicht mitgezählt.
2 -roll wirkt wie -rot , Ø -roll verändert nichts.

pick (16bn..16bØ +n -- 16bn..16bØ 16bn) 83

Der +n-te Wert auf dem Stack wird nach oben auf den Stack kopiert. Dabei wird +n selbst nicht mitgezählt.
Ø pick wirkt wie dup , 1 pick wie over.

.s (--) "dot-s"

Gibt alle Werte, die auf dem Stack liegen aus, ohne den Stack zu verändern. Oft benutzt, um neue Worte auszutesten. Die Ausgabe der Werte erfolgt von links nach rechts, der oberste Stackwert zuerst, so daß der top of stack (tos) ganz links steht!

clearstack (... --)

Löscht den Datenstack. Alle Werte, die sich vorher auf dem Stack befanden, sind verloren.

depth (-- n)

n ist die Anzahl der Werte, die auf dem Stack lagen, bevor DEPTH ausgeführt wurde.

sØ (-- addr) "s-zero"

addr ist die Adresse einer Uservariablen, in der die Startadresse des Stacks steht. Der Ausdruck sØ @ sp! wirkt wie clearstack und leert den Stack.

sp! (addr --) "s-p-store"

Setzt den Stackzeiger (stack pointer) auf die Adresse addr. Der oberste Wert auf dem Stack ist dann der, welcher in der Adresse addr steht.

sp@ (-- addr) "s-p-fetch"

Holt die Adresse addr aus dem Stackzeiger. Der oberste Wert im Stack stand in der Speicherstelle bei addr, bevor sp@ ausgeführt wurde.

3.5.2 Returnstack-Operationen

rdepth (-- n) "r-depth"

n ist die Anzahl der Werte, die auf dem Returnstack liegen.

>r (16b --) C,83 "to-r"

Der Wert 16b wird auf den Returnstack gelegt. Siehe auch R> .

r> (-- 16b) C,83 "r-from"

Der Wert 16b wird vom Returnstack geholt. Vergleiche R>.

r@ (-- 16b) C,83 "r-fetch"
Der Wert 16b ist eine Kopie des obersten Wertes auf dem Returnstack.

rdrop (--) C "r-drop"
Der oberste Wert wird vom Returnstack entfernt. Der Datenstack wird nicht verändert. Entspricht der Operation **r> drop** .

rØ (-- addr) U "r-zero"
addr ist die Adresse einer Uservariablen, in der die Startadresse des Returnstacks steht.

rp@ (-- addr) "r-p-fetch"
Holt die Adresse addr aus dem Returnstackzeiger. Der oberste Wert im Returnstack steht in der Speicherstelle bei addr.

rp! (addr --) "r-p-store"
Setzt den Returnstackzeiger (return stack pointer) auf die Adresse addr. Der oberste Wert im Returnstack ist nun der, welcher in der Speicherstelle bei addr steht.

push (addr --)
Der Inhalt aus der Adresse addr wird bis zum nächsten **EXIT** oder **;** auf dem Returnstack verwahrt und sodann nach addr zurückgeschrieben. Dies ermöglicht die lokale Verwendung von Variablen innerhalb einer :-Definition. Wird z.B. benutzt in der Form :

```
: hex.  ( n -- ) base push hex . ;
```

Hier wird innerhalb von **HEX.** in der Zahlenbasis **HEX** gearbeitet, um eine Zahl auszugeben. Nachdem **HEX.** ausgeführt worden ist, besteht die gleiche Zahlenbasis wie vorher, durch **HEX** wird sie also nur innerhalb von **HEX.** verändert.

4. Kontrollstrukturen

4.1 Programm-Strukturen

Wil Baden, auf den Sie in der englischsprachigen Literatur oft stoßen, hat in seinem Beitrag ESCAPING FORTH folgendes dargelegt:
Es gibt vier Arten von Steueranweisungen :

- die Abfolge von Anweisungen,
- die Auswahl von Programmteilen,
- die Wiederholung von Anweisungen und Programmteilen
- den Abbruch.

Die ersten drei Möglichkeiten sind zwingend notwendig und in den älteren Sprachen wie PASCAL ausschließlich vorhanden. Entsprechend steht im volksFORTH eine Anweisung für die Auswahl von Programmteilen zur Verfügung, wobei die Ausführung vom Resultat eines logischen Ausdrucks abhängig gemacht wird:

```
flag IF ‹Anweisungen› THEN
flag IF ‹Anweisungen› ELSE ‹Anweisungen› THEN
```

Soll dagegen im Programm ein Rücksprung erfolgen, um Anweisungen wiederholt auszuführen, wird bei einer gegebenen Anzahl von Durchläufen diese Anweisung eingesetzt, wobei der aktuelle Index über I und J zur Verfügung steht:

```
‹Grenzen› DO / ?DO ‹Anweisungen› LOOP
‹Grenzen› DO / ?DO ‹Anweisungen› ‹Schrittweite› +LOOP
```

Wenn eine Wiederholung von Anweisungen ausgeführt werden soll, ohne daß die Anzahl der Durchläufe bekannt ist, so ist eine Indexvariable mitzuführen oder sonstwie zum Resultat eines logischen Ausdrucks zu kommen. Die folgende Konstruktion ermöglicht eine Endlos-Schleife:

```
BEGIN ‹Anweisungen› REPEAT
```

Die Wiederholungsanweisungen sind insoweit symmetrisch, daß eine Anweisung solange (while) ausgeführt wird, wie ein Ausdruck wahr ist, oder eine Anweisung wiederholt wird, bis (until) ein Ausdruck wahr wird.

```
BEGIN ‹Anweisungen› flag UNTIL
BEGIN ‹Anweisungen› flag WHILE ‹Anweisungen› REPEAT
```

Beide Möglichkeiten lassen sich in volksFORTH auch kombinieren, wobei auch mehrere (multiple) WHILE in einer Steueranweisung auftreten dürfen.

```
BEGIN ‹Anweisungen›  flag WHILE ‹Anweisungen›  flag UNTIL
```

Nun tritt in Anwendungen häufig der Fall auf, daß eine Steueranweisung verlassen werden soll, weil sich etwas ereignet hat.

Dann ist die vierte Situation, der Abbruch, gegeben. C stellt dafür die Funktionen: break, continue, return und exit zur Verfügung; volksFORTH bietet hier exit **leave endloop quit abort abort"** und **abort(** an.
In FORTH wird EXIT dazu benutzt, um die Definition zu verlassen, in der es erscheint; **LEAVE** dagegen verläßt die kleinste umschließende DO...LOOP-Schleife.

Glossar

Ab der Version 3.81.3 verfügt volksFORTH über eine zusätzliche Steueranweisung für den Compiler, die bedingte Kompilierung in der Form:

```
have <word> not .IF <action1> .ELSE <action2> .THEN
```

Diese Worte werden außerhalb von Colon-Definitionen eingesetzt und ersetzen das **\needs** früherer Versionen.

have (-- flag)
prüft, ob ein Wort im Wörterbuch in der Suchreihenfolge existiert und hinterläßt ein entsprechendes Flag. In der Literatur und in Quelltexten findet man auch **exists?** als Synonym.

exit (--)
ist ein Synonym für **unnest** .
Dieses wird von ";" (Semicolon) als Abschluß einer Colondefinition compiliert. Ebenso dient **EXIT** als Austritt aus einem Wort, wobei das Programm in der aufrufenden Ebene fortgesetzt wird (return to caller).
Ein **EXIT** ist innerhalb von DO..LOOP-Strukturen nicht ohne weiteres möglich (siehe **endloop**).
Diese Steueranweisung ist nicht umkehrbar, der Sprung in eine hierarchisch niedrigere Ebene ist nicht erlaubt.
Typisch: **flag IF exit THEN**

?exit (flag --) "question-exit"
führt **EXIT** aus, falls das Flag wahr ist. Ist das Flag falsch, so geschieht nichts. Hierbei soll daran erinnert werden, daß jede Zahl ungleich NULL als wahr interpretiert wird.
Entspricht: **true IF exit THEN**

Ø=exit (flag --) "zero-equals-exit"
führt **EXIT** aus, falls das Flag falsch ist. Ist das Flag wahr, so geschieht nichts. Es entspricht **Ø= IF exit THEN** und wird typisch so eingesetzt:
key #cr - Ø=exit

IF (flag --) 83,I,C
(-- sys) compiling

wird in der folgenden Art benutzt:

```
flag IF ... ELSE ... THEN
```

oder:

```
flag IF ... THEN
```

Ist das Flag wahr, so werden die Worte zwischen IF und ELSE ausgeführt und die Worte zwischen ELSE und THEN ignoriert.
Der ELSE-Teil ist optional. Ist das Flag falsch, so werden die Worte zwischen IF und ELSE (bzw. zwischen IF und THEN , falls ELSE nicht vorhanden ist) ignoriert.

.IF ist das IF für den Interpretermodus.

THEN (--) 83,I,C
(sys --) compiling

wird in der folgenden Art benutzt:

```
IF (...ELSE) ... THEN
```

Hinter THEN ist die Programmverzweigung zuende.
Weil viele FORTH-Freunde die "alte" Schreibweise vor der Festlegung des 83er Standards besser und deutlicher finden, sei hier die Definition von ENDIF gezeigt:

```
' THEN Alias ENDIF  immediate restrict
```

.THEN ist das THEN für den Interpretermodus.

ELSE (--) 83,I,C
(sys1 -- sys2) compiling

wird in der folgenden Art benutzt:

```
flag IF ... ELSE ... THEN
```

ELSE wird unmittelbar nach dem Wahr-Teil, der auf IF folgt, ausgeführt. ELSE setzt die Ausführung unmittelbar hinter THEN fort.

.ELSE ist das ELSE für den Interpretermodus.

DO (w1 w2 --) 83,I,C
(sys --) compiling

beginnt eine Schleife und entspricht somit ?DO , jedoch wird der Schleifenrumpf mindestens einmal durchlaufen. Der Schleifenindex beginnt mit w2, Grenze ist w1 .
Ist w1=w2 , so wird der Schleifenrumpf 65536-mal durchlaufen.

?DO (w1 w2 --) 83,I,C "question-do"
(-- sys) compiling

wird in der folgenden Art benutzt:

?DO ... LOOP bzw. ?DO ... +LOOP

Beginnt eine Schleife. Der Schleifenindex beginnt mit w2, Limit ist w1. . Ist w2=w1, so wird der Schleifenrumpf überhaupt nicht durchlaufen.
Für Details über die Beendigung von Schleifen siehe +LOOP

LOOP (--) 83,I,C
(-- sys) compiling

entspricht +LOOP, jedoch mit einer festen Schrittweite von 1 .

+LOOP (n --) 83,I,C "plus-loop"
(sys --) compiling

Die Schrittweite n wird zum Loopindex addiert. Falls durch die Addition die Grenze zwischen limit-1 und limit überschritten wurde, so wird die Schleife beendet und die Loop-Parameter werden entfernt. Wurde die Schleife nicht beendet, so wird sie hinter dem korrespondierenden DO bzw. ?DO fortgesetzt.

I (-- w) 83,C

wird zwischen DO und LOOP benutzt, um eine Kopie des Schleifenindex auf den Stack zu holen.

J (-- w) 83,C

wird in zwei geschachtelten DO...LOOP-Schleifen zwischen DO .. DO und LOOP .. LOOP benutzt, um eine Kopie des Schleifenindex der äusseren Schleife auf den Stack zu holen.

leave (--) 83,C

beendet die zugehörige Schleife und setzt die Ausführung des Programmes hinter dem nächsten LOOP oder +LOOP fort. Mehr als ein LEAVE pro Schleife ist möglich, ferner kann LEAVE zwischen anderen Kontrollstrukturen auftreten. Der FORTH83-Standard schreibt abweichend vom volksFORTH vor, daß LEAVE ein immediate Wort ist.

endloop (--)

ermöglicht ein EXIT innerhalb einer DO...LOOP. Es wird so eingesetzt:

```
<Grenzen>DO
        <Anweisungen>
        flag IF endloop exit THEN
       LOOP
```

Damit kann exit auch in einer DO...LOOP-Schleife verwendet werden, vorausgesetzt, Sie halten sich an zwei Regeln:

1. Sie dürfen das System nicht mit Returnstack-Manipulationen aus dem Gleichgewicht gebracht haben.
2. Bei geschachtelten DO...LOOP-Schleife muß für jede Schleifenebene ein endloop dem abbrechenden exit vorangehen.

In der Literatur findet man auch UNDO als Synonym für ENDLOOP.

bounds (start count -- limit start)

dient dazu, ein Intervall, das durch Anfangswert und Länge gegeben ist, in ein Intervall umzurechnen, das durch Anfangswert und Endwert+1 beschrieben wird. Beispiel:

```
10 3 bounds DO...LOOP
```

führt dazu, das I die Werte 10 11 12 annimmt.

BEGIN (--) 83,I,C

(sys ---) compiling

wird in der folgenden Art benutzt:

```
BEGIN ( ...flag WHILE ) ... flag UNTIL
```

oder:

```
BEGIN ( ...flag WHILE ) ... REPEAT
```

BEGIN markiert den Anfang einer Schleife. Der ()-Ausdruck ist optional und kann beliebig oft auftreten. Die Schleife wird wiederholt, bis das Flag vor UNTIL wahr oder oder das Flag vor WHILE falsch ist. REPEAT setzt die Schleife immer fort.

Die Schleife `BEGIN <Anweisungen> flag UNTIL` wird immer mindestens einmal durchlaufen, da die Abbruchbedingung erst nach dem ersten Durchlauf geprüft wird.

Um dazu vollständige Symmetrie zu erreichen, kann im Ausdruck

```
BEGIN <action> flag WHILE.<action> REPEAT
```

der Anweisungsteil vor WHILE entfallen:

```
BEGIN flag WHILE <Anweisungen> REPEAT
```

prüft die Abbruchbedingung vor dem Eintritt in die Schleife.

REPEAT (--) 83,I,C
(-- sys) compiling

wird in der folgenden Form benutzt:

```
BEGIN (.. WHILE) .. REPEAT
```

REPEAT setzt die Ausführung der Schleife unmittelbar hinter BEGIN fort. Der ()-Ausdruck ist optional und kann beliebig oft auftreten. Deshalb gibt es kein AGAIN , benutzen Sie statt dessen REPEAT oder definieren: `' REPEAT Alias AGAIN immediate restrict`

UNTIL (flag --) 83,I,C
(sys --) compiling

wird in der folgenden Art benutzt:

```
BEGIN (... flag WHILE) ... flag UNTIL
```

Markiert das Ende einer Schleife, deren Abbruch durch flag herbeigeführt wird. Ist das Flag vor UNTIL wahr, so wird die Schleife beendet, ist es falsch, so wird die Schleife unmittelbar hinter BEGIN fortgesetzt.

WHILE (flag --) 83,I,C
(sys1 -- sys2) compiling

wird in der folgenden Art benutzt:

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

oder:

```
BEGIN .. flag WHILE .. flag UNTIL
```

Ist das Flag vor WHILE wahr, so wird die Ausführung der Schleife bis UNTIL oder REPEAT fortgesetzt, ist es falsch, so wird die Schleife beendet und das Programm hinter UNTIL bzw. REPEAT fortgesetzt. Es können mehrere WHILE in einer Schleife verwendet werden.

execute (addr --) 83

Das Wort, dessen Kompilationsadresse addr ist, wird ausgeführt.

perform (addr --)

addr ist eine Adresse, unter der sich ein Zeiger auf die Kompilationsadresse eines Wortes befindet. Dieses Wort wird ausgeführt. Entspricht der Sequenz @ execute .

case? (16b1 16b2 -- 16b1 false | true) "case-question"

vergleicht die beiden Werte 16b1 und 16b2 miteinander.
Sind sie gleich, verbleibt TRUE auf dem Stack. Sind sie verschieden, verbleibt FALSE und der darunterliegende Wert 16b1 auf dem Stack.
Wird z.B. in der folgenden Form benutzt :

```
key
    Ascii a case? IF ... exit  THEN
    Ascii b case? IF ... exit  THEN
drop
```

stop? (-- flag) "stop-question"

ist ein komfortables Wort, das es dem Benutzer gestattet, einen Programmablauf anzuhalten oder zu beenden.
Steht vom Eingabegerät ein Zeichen zur Verfügung, so wird es eingelesen. Ist es #ESC oder CTRL-C , so ist flag TRUE, sonst wird auf das nächste Zeichen gewartet. Ist dieses jetzt #ESC oder CTRL-C , so wird STOP? mit TRUE verlassen, sonst mit FALSE.
Steht kein Zeichen zur Verfügung, so ist das Flag FALSE . STOP? prüft also einen Tastendruck auf #ESC oder CTRL-C .

4.2 Worte zur Fehlerbehandlung

Diese arbeiten auch wie Steueranweisungen, wie die Definitionen von ARGUMENTS und IS-DEPTH zeigen:

```
: is-depth  ( n -- )
   depth 1-  - abort" falsche Parameterzahl!" ;
```

IS-DEPTH überprüft den Stack auf eine gegebene Anzahl Stackelemente (depth) hin.

abort (--) 83,I

leert den Stack, führt END-TRACE 'ABORT STANDARDI/O und QUIT aus.

'abort (--) "tick-abort"

ist ein deferred Wort, das mit NOOP vorbesetzt ist. Es wird in ABORT ausgeführt, bevor QUIT aufgerufen wird.

abort" (flag) 83,I,C "abort-quote"
(--) compiling

wird in der folgenden Form benutzt:

```
flag Abort" ccc"
```

Wird ABORT" später ausgeführt, so geschieht nichts, wenn das Flag falsch ist. Ist das Flag wahr, so wird der Stack geleert und der Inhalt von ERRORHANDLER ausgeführt. Beachten Sie bitte, daß im Gegensatz zu ABORT kein END-TRACE ausgeführt wird.

error" (flag) I,C "error-quote"
(--) compiling

Dieses Wort entspricht ABORT" , jedoch mit dem Unterschied, daß der Stack nicht geleert wird.

errorhandler (-- addr)

addr ist die Adresse einer Uservariablen, deren Inhalt die Kompilationsadresse eines Wortes ist. Dieses Wort wird ausgeführt, wenn das Flag, das ABORT" bzw. ERROR" verbrauchen, wahr ist. Der Inhalt von ERRORHANDLER ist normalerweise (ERROR.

(error (string --) "paren-error"

Dieses Wort steht normalerweise in der Variablen ERRORHANDLER und wird daher bei ABORT" und ERROR" ausgeführt. string ist dann die Adresse des auf ABORT" bzw. ERROR" folgenden Strings. (ERROR gibt das letzte Wort des Quelltextes gefolgt von dem String auf dem Bildschirm aus. Die Position des letzten Wortes im Quelltext, bei dem der Fehler auftrat, wird in SCR und R# abgelegt.

r# (-- addr) "r-sharp"

addr ist die Adresse einer Variablen, die den Abstand des gerade editierten Zeichens vom Anfang des gerade editierten Screens enthält.
Vergleiche (ERROR und SCR .

scr (-- addr) 83 "s-c-r"

addr ist die Adresse einer Variablen, die die Nummer des gerade editierten Screens enthält.
Vergleiche R# , (ERROR und LIST .

quit (--) "quit"

entleert den Returnstack und schaltet den interpretierenden Zustand ein.

?pairs (n1 n2 --) "question-pairs"

Ist n1 <> n2 , so wird die Fehlermeldung "unstructured" ausgegeben. Dieses Wort wird benutzt, um die korrekte Schachtelung der Kontrollstrukturen zu überprüfen.

4.3 Fallunterscheidung in FORTH

4.3.1 Strukturierung mit IF ELSE THEN / ENDIF

An dieser Stelle soll kurz die vielfältigen Möglichkeiten gezeigt werden, mit denen eine Fallunterscheidung in FORTH getroffen werden kann. Kennzeichnend für eine solche Programmsituation ist, daß von verschiedenen Möglichkeiten des Programmflusses genau eine ausgesucht werden soll.

Ausgehend von einer übersichtlichen Problemstellung, einem Spiel, werden die notwendigen Grundlagendefinitionen und die Entwicklung der oben beschriebenen Kontrollstruktur beschrieben.

Als Beispiel dient ein Spiel mit einfachen Regeln:

Bei diesem Trinkspiel, das nach [1] auch CRAPS genannt wird, geht es darum, einen Vorrat von gefüllten Gläsern unter den Mitspielern mit Hilfe des Würfels zu verteilen und leerzutrinken:

- Bei einer EINS wurde ein Glas aus dem Vorrat in der Tischmitte genommen und vor sich gestellt.
- Bei einer ZWEI oder einer DREI bekam der Nachbar/ die Nachbarin links ein Glas des eigenen Vorrates zugeschoben.
- Bei einer VIER oder einer FÜNF wurde dem Nachbarn/ der Nachbarin rechts ein Glas des eigenen Vorrates vorgesetzt.
- Bei einer SECHS wurden alle Gläser, die der Spieler/ die Spielerin vor sich stehen hatte, leergetrunken.

Zuordnung ist also: 1=nehmen, 2/3=links, 4/5=rechts, 6=trinken und entsprechend der Augenzahl des Würfels soll eine der 6 möglichen Aktionen ausgeführt werden. Das Programm soll sich darauf beschränken, das Ergebnis dieses Würfelns einzulesen und auszuwerten. Daraufhin wird eine Meldung ausgegeben, welche der sechs Handlungen auszuführen ist.

Für ein solches Programm ist eine Zahleneingabe notwendig. Diese wurde hier mit dem Wort F83-NUMBER? realisiert:

```
: F83-number?   ( string -- d f )
  number? ?dup
  IF
     0<  IF extend THEN
     true exit
  THEN
  drop  0 0  false  ;

: input#   ( <string> -- n )
  pad c/l 1- >expect
  pad F83-number?  2drop ;
```

Die Definition der Wörter, die sechs oben genannten Aktionen symbolisch ausführen sollen, richtet sich nach den Spielregeln, die für jedes Würfelergebis genau eine Handlung vorschreiben:

```
\ nehmen  trinken  links  rechts  schieben

: nehmen   bright ." ein Glas nehmen"          normal  2 spaces ;
: trinken  bright ." alle Gläser austrinken"   normal  2 spaces ;
: links    bright ." ein Glas nach LINKS"      normal  2 spaces ;
: rechts   bright ." ein Glas nach RECHTS"     normal  2 spaces ;

: schieben  ;
```

SCHIEBEN ist eine Dummyprozedur, ein Füllsel, dessen Notwendigkeit sich erst sehr spät ergibt.
Für den Dialog mit dem Anwender wird definiert:

```
: Anfrage      cr ." Sollen Sie nehmen, trinken oder schieben? "
               cr ." Bitte Ihre Augenzahl und <cr> : "   ;

: Glückwunsch  cr ." Viel Glück beim nächsten Wurf ... " ;
```

Das Wort AUSWERTUNG soll entsprechend einem Selektor genau eine von 6 möglichen Prozeduren ausführen. Also wird man prüfen, ob diese oder diese oder ... der Möglichkeiten in Frage kommt. Hinzu kommt noch die Prüfung, ob der übergebene Parameter zwischen (between) 1 und 6 lag.
Die Definition von BETWEEN ist volksFORTH-gemäß recht kurz:

```
( Wert Untergrenze Obergrenze -- false  oder )
(                             -- true   wenn Untergrenze≤Wert≤Obergrenze )
: between   1+ uwithin ;

: Auswertung.1   ( Wurfergebnis -- )
   dup 1 =   IF   nehmen  ELSE
             dup 2 = IF links schieben ELSE
             dup 3 = IF links schieben ELSE
                     dup 4 = IF rechts schieben ELSE
                     dup 5 = IF rechts schieben ELSE
                             dup 6 = IF trinken THEN
                             THEN
                             THEN
                     THEN
                     THEN
             THEN
      1 6 between not IF invers ." Betrug!" normal THEN ;
```

Ein Verzicht auf den ELSE-Teil führt schon zu einer übersichtlicheren Form:

```
: Auswertung.2  ( Wurfergebnis --)
     dup 1 = IF nehmen          THEN
     dup 2 = IF links  schieben THEN
     dup 3 = IF links  schieben THEN
     dup 4 = IF rechts schieben THEN
     dup 5 = IF rechts schieben THEN
     dup 6 = IF trinken         THEN

     1 6 between not IF inverse ." Betrug!" normal THEN ;
```

Da eine solche Prüfung auf Gleichheit in der Programmierpraxis oft vorkommt, stellt das volks4TH dafür das Wort case? zur Verfügung. case? vergleicht die obersten beiden Stackwerte miteinander. Bei Ungleichheit bleibt der Testwert (Selektor) erhalten, so daß die Worte DUP und = dadurch ersetzt werden.

```
: Auswertung.3   ( Wurfergebnis -- )

      1 case? IF nehmen          exit THEN
      2 case? IF links  schieben exit THEN
      3 case? IF links  schieben exit THEN
      4 case? IF rechts schieben exit THEN
      5 case? IF rechts schieben exit THEN
```

```
    6 case? IF trinken          exit THEN

    drop    invers ." Betrug!" normal    ;
```

Bei dieser Auswertung wird aus dem Quelltext zu wenig deutlich, daß bei ZWEI und DREI dieselbe Handlung ausgeführt wird, wie auch VIER und FÜNF die gleichen Aktionen zur Folge haben.

=OR prüft deshalb einen Testwert n2 auf Gleichheit mit einer unter einem Flag f1 liegenden Zahl n2. Das Ergebnis dieses Tests wird mit dem bereits vorliegenden Flag OR-verknüpft. Dieses neue Flag f2 und der Testwert n1 werden übergeben:

```
code =or        ( n1 f1 n2 -- n1 f2 )
   A D xchg  D pop
   S W mov
     W ) A cmp
      0= ?[ -1 # D mov ]?
  next
end-code

\ : =or  ( n1 f1 n2 -- n1 f2 )  2 pick = or  ;
```

Dieses Wortes bringt im Quelltext eine deutliche Verbesserung:

```
: Auswertung.4  ( Wurfergebnis --)
    dup
    1 6 between IF
                 dup 1 =       IF  nehmen           THEN
                 dup 2 = 3 =or IF  links schieben   THEN
                 dup 4 = 5 =or IF  rechts schieben  THEN
                 dup 6 =       IF  trinken          THEN
                ELSE
                 invers ." Betrug!" normal
                THEN
    drop   ;
```

Damit wurde ohne eine CASE-Anweisung eine sehr übersichtliche Steuerung des Programm-Flusses geschaffen.
Die Plausibilitätsprüfung, ob die eingegeben Zahl zwischen 1 und 6 lag, ist hier an den Anfang gerückt und wird in einem einzigen ELSE-Zweig abgearbeitet.

4.3.2 Behandlung einer CASE - Situation

4.3.2.1 Strukturelles CASE

Viele Programmiersprachen eine CASE-Anweisung zur Verfügung, die wie in PASCAL mit Hilfe eines Fall-Indices eine Liste von Fall-Konstanten auswertet und eine entsprechende Anweisung ausführt.
Obwohl ein solches CASE-Konstrukt - wie oben gezeigt - nicht notwendig ist, macht es Programme besser lesbar und liegt bei Problemstellungen wie der Auswer-

tung eines gegebenen Index eigentlich näher.
Dies ist in [1] ausführlich diskutiert worden, wobei aber der ältere Eaker-CASE [2] von Dr. Charles Eaker sicherlich der bekanntere ist, der auch in der Literatur und in Quelltexten häufig Erwähnung und Verwendung findet.

Herr H. Schnitter hat diesen Eaker-CASE für das volks4TH implementiert und dabei Veränderungen in der Struktur und Verbesserungen in der Anwendung vorgenommen.

```
\ caselist  initlist  >marklist  >resolvelist

! variable caselist

! : initlist     ( list -- addr )
                 dup @ swap off
  ;

! : >marklist    ( list -- )
                 here over @ , swap !
  ;

! : >resolvelist ( addr list -- )
                 BEGIN dup @
                 WHILE dup dup @ dup @ rot ! >resolve
                 REPEAT !
  ;

\ case  elsecase  endcase

: CASE           caselist initlist 4
; immediate restrict

: ELSECASE       4 ?pairs
                 compile drop 6
; immediate restrict

: ENDCASE        dup 4 =
                 IF   drop compile drop
                 ELSE 6 ?pairs
                 THEN caselist >resolvelist
; immediate restrict

\ of  endof

: OF             4 ?pairs
                 compile over
                 compile =
                 compile ?branch
                 >mark compile drop 5
; immediate restrict

: ENDOF          5 ?pairs
                 compile branch
                 caselist >marklist
                 >resolve 4
; immediate restrict
```

Diese Implementierung des Eaker-CASE stellt eine Verbesserung gegenüber dem Original dar, indem Herr Schnitter die Kontrollstruktur um **ELSECASE** erweitert hat. Selbstverständlich ist die neue Version vollkommen aufwärtskompatibel mit der Original-version.

Verbesserung:

In der Originalversion der CASE-Struktur ist es nicht möglich, zwischen dem letzten **ENDOF** und **ENDCASE** einen Wert oder ein Flag auf den Stapel zu legen, da **ENDCASE** grundsätzlich den "Top of Stack" entfernte.
In der verbesserten Version bereinigt **ELSECASE** den Stapel. **ELSECASE** muß jedoch nicht aufgerufen werden; in diesem Fall kompiliert **ENDCASE** wie bisher ein DROP. Es ist jetzt möglich, zwischen den Worten **ELSECASE** und **ENDCASE** - wie auch zwischen **OF** und **ENDOF** - einen Wert auf den Stapel zu legen und diesen außerhalb der CASE-Kontrollstruktur zu verwenden.

Änderung:

Die Vorwärtsreferenzen werden nicht über den Stack aufgelöst, sondern über eine verkettete Liste.
Die Variable **caselist** enthält die Startadresse für noch nicht bekannte Sprungadressen. Die Schachtelungstiefe mehrerer CASE-Konstruktionen ist beliebig und wird durch initlist gelöst. **>marklist** füllt zur Kompilierzeit die Liste der Vorwärtsreferenzen und **>resolvelist** löst sie wieder auf.

Anwendungshinweis:

Wenn diese Definitionen außerhalb der Zusammenstellung des Arbeitssystems zugeladen werden, sollten nach dem Compilieren die Namen der mit ¦ als headerless markierten Worte mit **clear** entfernt werden.

Das Beispiel einer Tastatuabfrage auf CTRL-Tasten zeigt , wie dieses CASE-Konstrukt einzusetzen ist. Wichtig ist hierbei, daß das **OF** selbst die Gleichheit der beiden vorliegenden Werte prüft und in diesem Fall die Anweisungen zwischen OF und ENDOF ausführt.

```
: Control       bl word 1+ c@  $BF and  state @
                IF  [compile] Literal  THEN
; immediate

: Tastaturabfrage
  ." exit mit ctrl x" cr
  BEGIN key
        CASE control A OF ." action ^a " cr false ENDOF
             control B OF ." action ^b " cr false ENDOF
             control C OF ." action ^c " cr false ENDOF
             control D OF ." action ^d " cr false ENDOF
             control X OF ." exit "         true  ENDOF
        ELSECASE
                   ." befehl unbekannt " cr false
        ENDCASE
  UNTIL   ;
```

Mit dieser CASE-Anweisung läßt sich die Zuordnung der sechs Möglichkeiten zu den sechs Anweisungen ähnlich wie in PASCAL schreiben, lediglich Bereiche wie Ø..255 als Fall-Konstanten sind nicht erlaubt.

```
: Auswertung.5  ( Augenzahl -- )
      CASE
          1 OF  nehmen           ENDOF
          2 OF  links schieben   ENDOF
          3 OF  links schieben   ENDOF
          4 OF  rechts schieben  ENDOF
          5 OF  rechts schieben  ENDOF
          6 OF  trinken  ENDOF
      ELSECASE
          invers ." Betrug!" normal
      ENDCASE
;
```

Das vollständige Programm kann so geschrieben werden, wobei die typische Dreiteilung Eingabe-Verarbeitung-Ausgabe deutlich wird:

```
: craps   ( -- )

    cr Anfrage cr
       input#
       Auswertung
    cr Glückwunsch
;
```

Wil Baden hat in [1] ausgeführt, das eine CASE-Anweisung nur syntaktischer Zucker für ein Programm ist und letztendlich nichts weiter ist, als das Compilieren einer verschachtelten IF...THEN-Anweisung.
Eine solche Implemetierung für das volksFORTH83 wurde von Herrn K. Schleisiek-Kern geschrieben:

```
\ CASE   OF  ENDOF   ENDCASE     BREAK

: CASE      ( n1 -- n1 n1 )  dup ;

: OF        [compile] IF compile drop ;  immediate restrict

: ENDOF     [compile] ELSE 4+ ;  immediate restrict

: ENDCASE   compile drop
            BEGIN
              3 case?
            WHILE
              >resolve
            REPEAT ;  immediate restrict
```

Wil Badens Implementierung hält sich sehr eng an die logischen Grundlagen, wobei der Unterschied zum EAKER-CASE hauptsächlich darin besteht, daß hier jedes TRUE-Flag den Anweisungsteil zwischen OF und ENDOF ausführt; das OF nimmt keine Prüfung auf Gleichheit vor, sondern beliebige Ausdrücke können zu einem Flag führen, das dann von OF ausgewertet wird. So ist das Auswerten des Fall-Index variabler als beim EAKER-CASE:

```
: Auswertung.6  ( Augenzahl -- )
   dup
      1 6 between  not
            IF invers ." Betrug!" normal  drop exit  THEN
   CASE 1 =  OF nehmen          ENDOF
   CASE 6 =  OF trinken         ENDOF
   CASE 4 <  OF links schieben  ENDOF
   CASE 3 >  OF rechts schieben ENDOF
   ENDCASE  ;
```

Hier bei dieser Konstruktion steht die Plausibilitätsprüfung ganz vorn, um den ELSECASE-Fall durch ein EXIT aus dem Wort zu erreichen. Wird keines der Worte aus der Auswahl-Liste ausgeführt, läßt sich mit BREAK eine andere Lösung erreichen:

```
: BREAK     compile exit
          [compile] THEN ;  immediate restrict
```

Dadurch, daß BREAK ein EXIT aus dem Wort darstellt, wird ein (implizites) ELSECASE erreichen, indem man die Anweisungen der Auswahl-Liste mit OF und BREAK klammert und die Anweisungen für den ELSE-Fall nach ENDCASE aufführt:

```
: Auswertung.7  ( Augenzahl -- )

   CASE 1 =         OF  nehmen           BREAK
   CASE 2 = 3 =or   OF  links schieben   BREAK
   CASE 4 = 5 =or   OF  rechts schieben  BREAK
   CASE 6 =         OF  trinken          BREAK
   ENDCASE
   invers ." Betrug!" normal  ;
```

4.3.2.2 Positionelles CASE

Eine ganz anderen Lösungsansatz bietet ein positioneller CASE Konstrukt, bei dem die Fallunterscheidung durch den Fall-Index tabellarisch vorgenommen wird.

Bei den bisherigen Lösungen wurden immer eine Reihe von Vergleichen zwischen einem Fall-Index und einer Liste von Fall-Konstanten vorgenommen; nun wird der Fall-Index selbst benutzt, die gewünschte Prozedur auszuwählen. Die Verwendung des Fall-Index als Selektor bringt auch Vorteile in der Laufzeit, da die Vergleiche entfallen.

Wenn FORTH-Worte in Tabellen abgelegt werden sollen, stellt sich das Problem, daß ein FORTH-Wort bei seinem Aufruf normalerweise die eincompilierten Worte ausführt.
Bei einer Tabelle ist das nicht erwünscht; dort ist sinnvollerweise gefordert, daß die Startadresse der Tabelle übergeben wird, um der Fall-Index als Offset in diese Tabelle zu nutzen.

Dies läßt sich in volksFORTH entweder auf die traditionelle Weise mit] und [oder dem volksFORTH-spezifischen Create: lösen:

```
Create  Glas
      ]  nehmen  links schieben
         rechts schieben  trinken  [

Create: Glas
        nehmen
        links schieben
        rechts schieben
        trinken ;
```

Diese Tabelle Glas macht auch deutlich, welche Funktion das Dummy-Wort **schieben** außer einer besseren Lesbarkeit noch hat: Es löst die Schwierigkeit, daß 6 möglichen Wurfergebnissen nur 4 mögliche Aktionen gegenüberstehen.

Die Art und Weise des Zugriffs in **BEWEGEN** entspricht dem Zugriff auf eine Zahl in einem eindimensionalen Feld, einem Vektor:

```
: bewegen  ( addr n -- cfa )
    2* +  perform ;

: richtig  ( n  --  0<= n <= 3 )
    swap
    1 max  6 min
    3 case? IF  2  1- exit  THEN
    5 case? IF  4  1- exit  THEN
    1-  ;
```

Dieses Wort **RICHTIG** läßt zwar Werte kleiner als 1 und größer als 6 zu, justiert sie aber auf den Bereich zwischen 1 und 6 . Auch hier müßte eine Möglichkeit geschaffen werden, ein Wurfergebnis außerhalb der 6 Möglichkeiten als Betrugsversuch zurückzuweisen !
Die Verbindung von Tabelle und Zugriffsprozedur wird von dem Wort :Does> vorgenommen:

```
\ :Does>  für  Create <name> :Does> <action> ;   ks 25 aug 88

! : (does>   here >r [compile] Does> ;

  : :Does>    last @ 0= Abort" without reference"
     (does>  current @  context !   hide 0 ] ;
```

Dieses Wort :DOES> weist dem letzten über Create definierten Wort einen Laufzeit-Teil zu. Dieses Wort wurde von Herrn K. Schleisiek-Kern programmiert ; auch hier gilt der Hinweis, nach dem Compilieren das mit ! als headerless deklarierte Wort durch clear zu löschen.

```
Create: Auswertung.8
          nehmen
           links schieben
           rechts schieben
          trinken ;
       :Does>
          richtig bewegen ;
```

Ohne :DOES> sind die Tabelle und die Zugriffsprozeduren voneinander unabhängige Worte:

```
: CRAPS1
     cr  Anfrage cr
           input#
           Glas richtig bewegen
     cr  Glückwunsch  ;
```

Entschließt man sich dagegen, sowohl Tabelle als auch Zugriffsprozedur in einem Wort zu definieren, so ergibt sich das gewohnte Erscheinungsbild:

```
: CRAPS
     cr  Anfrage cr
           input#
           Auswertung
     cr  Glückwunsch  ;
```

Bei häufigerem Einsatz solcher Tabellen bietet sich der Einsatz von positional CASE defining words an. Auch hier wiederum zuerst die volks4TH-gemäße Lösung, danach die traditionelle Variante:

```
: Case:   ( -- )
   Create:  Does>   ( pfa -- )  swap 2* +  perform ;

\ alternative Definition für CASE:
: Case:
        :  Does>   ( pfa -- )  swap 2* +  perform ;
```

Eine sehr elegante Möglichkeit, die Fehlerbehandlung im Falle eines unglaubwürdigen Fall-Indexes zu handhaben, bietet das Wort **Associative:** .
Dieses Wort **Associative:** durchsucht eine Tabelle nach einer Übereinstimmung zwischen einem Zahlenwert auf dem Stack und den Zahlenwerten in der Tabelle und liefert den Index der gefundenen Zahl (match) zurück. Im Falle eines Mißerfolgs (mismatch) wird der größtmögliche Index +1 (out of range = maxIndex +1) übergeben:

```
: Associative:           ( n -- )
   Constant   Does>      ( n - index )
   dup @ -rot
   dup @ 0
   DO  2+  2dup  @ =
       IF  2drop drop  I  0 0  LEAVE  THEN
   LOOP 2drop ;


6 Associative:  Auswerten

                  1 ,
                2 , 3 ,
                4 , 5 ,
                  6 ,

         Case:  Handeln          \ besteht aus :

                 nehmen
```

```
            links  links
           rechts  rechts
              trinken
        schimpfen ;
```

Statt der Primitivabsicherung über **MIN** und **MAX** wird eine out of range Fehlerbehandlung namens **schimpfen** an der Tabellenposition maxIndex +1 durchgeführt.

4.3.2.3 Einsatzmöglichkeiten

Dieser letzte Teil der Ausführungen über die Möglichkeiten, eine CASE-Situation zu handhaben, greift Anregungen aus der Literatur [5],[6] auf.

Dazu werden zwei Worte definiert:

CLS löscht den gesamten Bildschirm und

CELLS macht die Berechnung des Tabellenzugriffs deutlicher:

```
: cls   full page ;
: cells  2* ;
```

Das Inhaltliche und die tabellarische Struktur bleiben unverändert, lediglich die Behandlung einer out of range Situation wird diesmal mit **min** und **max** und zweimaligem Eintragen der Fehler-Routine **schimpfen** verwirklicht.

```
Create: Handlung
          schimpfen   nehmen links links
                      rechts rechts trinken   schimpfen  ;

\ Die Ausführung einer Liste  nach Floegel 7/86
: auswählen   ( addr n -- *cfa ) 2 arguments
     swap  0 max                \ out of range MIN
           7 min                \ out of range MAX
     cells +  ;

: auswerten   ( n -- ) 1 arguments
     Handlung auswählen  perform ;

: .all  ( -- )
     8 0  DO  cr I dup . auswerten  2 spaces  LOOP ;
```

AUSWÄHLEN übergibt bei gegebenem Vektor und gegebenem Index einen Zeiger auf die code field address des entsprechenden Wortes. **AUSWERTEN** führt das so ausgewählte Wort aus und **.ALL** diente nur zur Kontrolle. Solch ein Wort, das angelegte Datenstrukturen auf dem Bildschirm darstellt, sollte in der Entwicklungsphasen eines Programmes immer dabei sein.

Eine weitere Möglichkeit, Werte in einen Vektor einzutragen, hat Herr Floegel in seinem Buch [4] dargestellt:

```
Create Tabelle  8 cells allot
       :Does> ( i -- addr ) swap cells + ;

 ' schimpfen  0 Tabelle !
 ' nehmen     1 Tabelle !
```

```
' links dup  2 Tabelle !
             3 Tabelle !
' rechts dup 4 Tabelle !
             5 Tabelle !
' trinken    6 Tabelle !
' schimpfen  7 Tabelle !

: auswerten  ( i -- )  0 max  7 min  Tabelle perform ;

: .action    ( i -- )
    Tabelle @  >name  bright .name normal ;
: .Tabelle  ( -- )  cr  8 0 DO  cr I .action  LOOP ;
```

Hier besteht mit .ACTION und .TABELLE die Möglichkeit, sich den Vektor darstellen zu lassen. In ähnlicher Weise werden auch im Kommandozeilen-Editor CED die neuen Aktionen in die Eingabe-Vektoren eingetragen.

Eine geringfügige Modifikation von [5] soll die Verknüpfung eines Vektors von Worten und einer Menü-Option zeigen:

```
Create function
        ] noop noop noop noop
          noop noop noop noop [
       :Does> ( i -- addr )
          swap  0 max 7 min   cells + ;
```

function ist ein execution vector , der mit NOOP vorbesetzt ist. Zur Laufzeit liefert er die Adresse des indizierten Elementes zurück.

```
: .action  ( i addr -- )
    @   >name  bright .name normal ;
```

.WORD gibt den Namen eines Wortes aus, dessen CFA in eine Adresse eingetragen wurde.

```
: option   ( i -- )
    r>
    dup 2+ >r                  \ i *w.addr
    @                          \ i w.addr
    stash swap  function !     \ i  w.addr i addr
    function .action ;         \ i addr
```

option holt die Adresse des auf **option** folgenden Wortes. Das Wort soll nicht ausgeführt werden, sondern das nachfolgende. Nur der Pointer auf das Wort soll ausgewertet werden. Nach dem übergebenen Index wird der Pointer in **function** eingetragen. Der Name des so eingetragenen Wortes wird angezeigt !

```
\ Menü        jrg 06feb89
: Menü
   0 option schimpfen
     1 option nehmen
     2 option links
     3 option links
     4 option rechts
     5 option rechts
     6 option trinken
   7 option schimpfen  ;
```

Wenn das Wort **MENÜ** aufgerufen wird, werden nicht nur die Optionen in die Tabelle eingetragen, sondern auch namentlich auf dem Bildschirm dargestellt. Diese Technik bietet sich für eine Menüzeile an fester Bildschirmposition an, ähnlich der Statuszeile des volksFORTH. Zum Ändern solcher Menüpunkte bieten sich die Funktionstasten an:

```
: fkey   ( -- )
   key &58 + abs  function perform ;
```

FKEY liefert beim Druck einer Funktionstaste einen Wert von -59 bis -68 zurück. Dieser wird für 1Ø Funktionstasten in den Bereich von -1 bis -1Ø skaliert und der Absolutwert gebildet.

[1] Wil Baden — Ultimate CASE-Statement (VD2/87 S.4Ø ff.)

[2] Dr. Charles Eaker — Just in CASE (FORTH DIM II/3)

[3] R. Zech — FORTH 83 (S.98ff/S.318f.)

[4] E. Floegel — FORTH Handbuch (S.1Ø9)

[5] W. Wejgaard — Menus in FORTH Elektroniker 9/88 (S.1Ø9 ff.)

4.4 Rekursion

Bevor die Technik der Rekursion für das volks4TH dargestellt wird, soll ein anderes Wort **.LASTNAME** zeigen, daß das Wort **LAST** mit dem in der Literatur oft anzutreffenden **LATEST** identisch ist:
Beide Worte liefern die name field address des zuletzt definierten Wortes im **CURRENT**-Vokabular. Das Wort **LAST'** dagegen liefert die cfa des zuletzt definierten Wortes.

```
: .lastname   last @  .name ;
```

Die Rekursion ist eine Technik, bei der ein Wort sich immer wieder selbst aufruft. Eines der bekannten Beispiele dafür ist die Berechnung der Fakultät einer positiven ganzen Zahl. Hierbei ergibt sich n! aus dem Produkt aller ihrer Vorgänger.

Im volks4TH ist der Selbstaufruf eines Wortes durch **RECURSIVE** gekennzeichnet, so daß sich ein Programm zur Fakultätsberechnung wie folgt präsentiert:

```
: fakultät  ( +n -- n! )
    recursive
    dup  Ø< IF  drop ." keine negativen Argumente! " exit
           THEN
    ?dup Ø= IF  1  \ Spezialfall: Ø
           ELSE  dup 1-  fakultät *
```

```
                THEN  ;

    cr  4 fakultät .
    cr  5 fakultät .
    cr  6 fakultät .
```

Allerdings findet sich - vor allem in der figFORTH-Literatur - ein Wort **MYSELF** , das mit dem in FORTH83-Umgebungen anzutreffenden **RECURSE** identisch ist. Da auch diese Konstruktion, bei der **MYSELF/RECURSE** als Platzhalter für den Wortnamen dienen, gerne eingesetzt wird, werden die möglichen Definitionen und eine weitere Form von **FAKULTÄT** gezeigt:

```
    : myself   last @  name> , ;  immediate

    : myself   last' , ;  immediate

    : recurse   [compile] myself ;  immediate
    ' myself  Alias  recurse   immediate

    : fakultät  ( +n -- n! )
       dup 0< IF  ." keine negativen Argumente erlaubt! "
              ELSE  ?dup 0= IF  1
                            ELSE  dup 1-  myself *
                            THEN
              THEN  ;
```

Bei der Verwendung von **RECURSE** wird lediglich **MYSELF** dadurch ersetzt:

```
    ...
                    ?dup 0= IF  1          \ Spezialfall: 0
                            ELSE  dup 1-  recurse *
                            THEN
    ...
```

5. Ein- / Ausgabe im volksFORTH

5.1 Ein- / Ausgabebefehle im volksFORTH

Alle Eingabe- und Ausgabeworte (**KEY EXPECT EMIT TYPE** etc.) sind im volksFORTH vektorisiert, d.h. bei ihrem Aufruf wird die Codefeldadresse des zugehörigen Befehls aus einer Tabelle entnommen und ausgeführt. So ist im System eine Tabelle mit Namen **DISPLAY** enthalten, die für die Ausgabe auf dem Bildschirmterminal sorgt.

Dieses Verfahren der Vektorisierung bietet entscheidende Vorteile:

- Mit der Input-Vektorisierung kann man z.B. mit einem Schlag die Eingabe von der Tastatur auf ein Modem umschalten.
- Durch die Output-Vektorisierung können mit einer neuen Tabelle alle Ausgaben auf ein anderes Gerät (z.B. einenDrucker) geleitet werden, ohne die Ausgabebefehle selbst ändern zu müssen.
- Mit einem Wort (**DISPLAY, PRINT**) kann das gesamte Ausgabeverhalten geändert werden. Gibt man z.B. ein:

  ```
  print 1 list display
  ```

  wird Screen 1 auf einen Drucker ausgegeben und anschließend wieder auf den Bildschirm zurückgeschaltet. Man braucht also kein neues Wort, etwa **PRINTERLIST**, zu definieren.

Eine neue Tabelle wird mit dem Wort **OUTPUT:** erzeugt. Die Definition können Sie mit **view output:** nachsehen. **OUTPUT:** erwartet eine Liste von Ausgabeworten, die mit **;** abgeschlossen werden muß.

Beispiel:

```
Output: >PRINTER
   pemit pcr ptype pdel ppage pat pat? ;
```

Damit wird eine neue Tabelle mit dem Namen **>PRINTER** angelegt. Beim späteren Aufruf von **>PRINTER** wird die Adresse dieser Tabelle in die Uservariable **OUTPUT** geschrieben. Ab sofort führt **EMIT** ein **PEMIT** aus, **TYPE** ein **PTYPE** usw.

Die Reihenfolge der Worte nach OUTPUT:

```
userEMIT userCR userTYPE userDEL userPAGE userAT userAT?
```

muß unbedingt eingehalten werden.

Entsprechend wird die Input-Vektorisierung gehandhabt.

5.2 Ein- / Ausgaben über Terminal

Das volks4th verfügt über eine Reihe von Konstanten, die der besseren Lesbarkeit dienen:

c/row (-- Anzahl)
ist die Konstante, die die Anzahl der Zeichen pro Zeile (&80) angibt.

c/col (-- Anzahl)
ist die Konstante, die die Anzahl der Zeichen pro Spalte (&25) angibt.

c/dis (-- Anzahl.Cells)
ist die Konstante, die die Größe des Speichers für einen Ausgabeschirm angibt.

c/l (-- +n) "characters-per-line"
+n ist die Anzahl der Zeichen pro Screenzeile. Aus historischen Gründen ist dieser Wert &64 bzw. $40 .

l/s (-- +n) "lines-per-screen"
+n ist die Anzahl der Zeilen pro Screen.

bl (-- n) "b-l"
n ist der ASCII-Wert für ein Leerzeichen.

#esc (-- n) "number-escape"
n ist der ASCII-Wert für Escape.

#cr (-- n) "number-c-r"
n ist der Wert, den man durch KEY erhält, wenn die Return-Taste <cr> gedrückt wird.

#lf (-- n) "number-linefeed"
n ist der ASCII-Wert für Linefeed.

#bel (-- n)
n ist der ASCII-Wert für BELL.

#bs (-- n) "number-b-s"
n ist der Wert, den man durch KEY erhält, wenn die Backspace-Taste gedrückt wird.

standardi/o (--) "standard-i-o"
stellt sicher, daß die beim letzten save bestimmten Ein- und Ausgabegeräte wieder eingestellt sind.

inputkol und outputkol
sind beides definierende Wörter, die eine festgelegte Anzahl von Zeigern auf Prozeduren erwarten.

area (-- addr)
ist eine Uservariable und zeigt auf den Zustandsvektor des aktiven Windows.

areakol
ist ein definierendes Wort, das zur Erzeugung eines Windows benutzt wird.

terminal
ist ein Window, das sich von Zeile Ø - 23 erstreckt für das Terminal Output Window.

window (topline bottomline --)
setzt die oberste und unterste Zeile des aktuellen Windows neu.

full (--)
setzt das aktuelle Window über den ganzen Bildschirm von der obersten bis zur untersten Zeile. Somit entspricht

```
: cls   full page ;
```

dem Befehl, der den gesamten Bildschirm löscht.

curat? (-- row col)
liefert die aktuelle Cursorposition im aktiven Fenster.

cur! (--)
setzt den Cursor in das Window, auf das AREA zeigt.

setpage (n --)
setzt aktive Bildschirmseite.

video@ (-- seg)
liefert das Segment zurück, in dem der Speicher der Videokarte liegt.

savevideo (-- seg ¦ ff)
rettet den Speicherbereich der Videoausgabe; wird bei dem Wort **MSDOS** eingesetzt.

restorevideo (seg --)
restauriert die Bildschirmausgabe auf dem angegebenen Segment; wird bei dem Wort **MSDOS** eingesetzt.

catt (-- addr)
ist eine Variable, die das Attribut zur Zeichendarstellung enthält, z.B. das Zeichenattribut **INVERS** .

list (u --) 83
zeigt den Inhalt des Screens u auf dem aktuellen Ausgabegerät an. **SCR** wird auf u gesetzt.
Siehe **BLOCK** .

(page (--) "(page"
löscht den Bildschirm und positioniert den Cursor in die linke obere Ecke.
Vergleiche **PAGE** .

page (--)
bewirkt, daß der Cursor des Ausgabegerätes auf eine leere neue Seite bewegt wird. Eines der OUTPUT-Worte.

(del (--) "(del"
löscht ein Zeichen links vom Cursor.
Vergleiche **DEL** .

del (--)
löscht das zuletzt ausgegebene Zeichen. Bei Druckern ist die korrekte Funktion nicht immer garantiert. Eines der OUTPUT-Worte

(cr (--) "(c-r"
setzt den Cursor in die erste Spalte der nächsten Zeile. Ein **PAUSE** wird ausgeführt.

cr (--) 83 "c-r"
bewirkt, daß die Schreibstelle des Ausgabegerätes an den Anfang der nächsten Zeile verlegt wird. Eines der OUTPUT-Worte.

?cr (--) "question-c-r"
prüft, ob in der aktuellen Zeile mehr als C/L Zeichen ausgegeben wurden und führt in diesem Fall CR aus.

(at (row col --) "(at"
setzt die aktuelle Cursorposition. (AT positioniert den Cursor in der Zeile row und der Spalte col . Ein Fehler liegt vor, wenn row > $18 (&24) oder col > $5Ø (&8Ø) ist. Die fehlerhafte Ausgabe wird nicht unterdrückt!
Vergleiche AT .

(at? (-- row col) "(at-question"
liefert die aktuelle Cursorposition. row ist die aktuelle Zeilennummer des Cursors, col die aktuelle Spaltennummer.
Vergleiche AT? .

at (row col --)
positioniert die Schreibstelle des Ausgabegerätes in die Zeile row und die Spalte col. AT ist eines der über OUTPUT vektorisierten Worte.
Siehe AT? .

.i.at? (-- row col) "at-question"
ermittelt die aktuelle Position der Schreibstelle des Ausgabegerätes und legt Zeilen- und Spaltennummer auf den Stack. Eines der OUTPUT-Worte.

col (-- #col) "col"
#col ist die Spalte, in der sich die Schreibstelle des Ausgabegerätes gerade befindet.
Vergleiche ROW und AT? .

row (-- #row) "row"
#row ist die Zeile, in der sich die Schreibstelle des Ausgabegerätes befindet.die Benutzung beschreiben-
Vergleiche COL und AT? .

curoff (--)
schaltet den Cursor aus.

curon (--)
schaltet den Cursor ein.

curshape (topline bottomline --)
bestimmt das Aussehen des Cursors.

5.3 Drucker-Ausgaben

printer (--)
ist das Vokabular mit den Worten zur Druckersteuerung.

print (--)
schaltet die Ausgabe auf den Drucker um

+print (--)
schaltet den Drucker zusätzlich zu.

lst! (8b --)
gibt ein Byte zum Drucker aus.

5.4 Ein- / Ausgabe von Zahlen

Die Eingabe von Zahlen erfolgt im interpretativen Modus über die Tastatur, wobei grundlegend Eingabeworte mit **number number?** und den verwandten Worten definiert werden.
Bei der Ausgabe von Zahlen ist wieder die fehlende Typisierung von FORTH zu beachten - für ein bestimmtes Datenformat (integer, unsigned, double) ist jeweils der geeignete Operator auszuwählen.

. (n --)
gibt den obersten Stack-Werte als Zahl (integer) aus.
Soll die Ausgabe des nachfolgenden Leerzeichens unterdrückt werden, so sind die Befehle für rechtsbündige Zahlenausgabe **.r u.r d.r** mit einer Feldlänge von Ø zu benutzen.

u. (u --)
gibt den obersten Stack-Wert als vorzeichenlose 16Bit-Zahl (unsigned) aus.

d. (d --)
gibt die obersten beiden Stack-Werte als 32Bit-Zahl (double) aus.

.r (n Feldlänge --)
druckt eine 16Bit-Zahl in einem Feld mit angegebener Länge rechtsbündig aus.

u.r (u Feldlänge --)
gibt den obersten Stack-Wert als vorzeichenlose 16Bit-Zahl in einem Feld rechtsbündig aus.

d.r (d Feldlänge --)
gibt eine 32Bit-Zahl in einem Feld rechtsbündig aus.

5.5 Ein- / Ausgabe über einen Port

pc@ (port.addr -- 8b)
holt ein Byte von port.addr aus einem Peripheriebaustein des 8Ø86-Systems auf den Stack.

pc! (8b port.addr --)
speichert ein Byte in einen Peripheriebaustein des 8Ø86-Systems bei port.addr.

5.6 Eingabe von Zeichen

In FORTH wird man immer einen Speicherbereich benennen, in dem Zeichen und Zeichenketten verarbeitet werden. Hierfür verwendet man meistens einen kleinen, 8Ø Zeichen langen Speicherbereich namens PAD . Dieser Notizblock - so die deutsche Übersetzung von pad - belegt keinen festen Speicherbereich und steht sowohl dem FORTH-System als auch dem zur Verfügung.

Dann möchte ich Ihnen mit dem Texteingabe-Puffer tib einen weiteren wichtigen Speicherbereich vorstellen,der den vernünftigen Umgang mit den angeschlossenen Geräten sicherstellt. Weil die Texteingabe über die Tastatur relativ langsam vor sichgeht, werden die Zeichen hier erst in einem freien Speicherbereich, dem Pufferspeicher tib , gesammelt und dann abgearbeitet.

tib (-- addr) 83
liefert die Adresse des Text-Eingabe-Puffers. Hier wird die Eingabe-Befehlszeile des aktuellen Eingabegerätes (meist KEYBOARD) gespeichert.
Siehe >TIB .

#tib (-- addr) 83 "number-t-i-b"
addr ist die Adresse einer Variablen, die die Länge des aktuellen Textes (die Anzahl der Zeichen) im Text-Eingabe-Puffer enthält.
Vergleiche TIB .

>tib (-- addr) "to-tib"

addr ist die Adresse eines Zeigers auf den Text-Eingabe-Puffer.
Siehe **TIB** .

>in (-- addr)

ist eine Variable, die den Offset auf das gegenwärtige Zeichen im Quelltext enthält.
>in indiziert relativ zum Beginn des Blockes, der durch die Variable **blk** gekennzeichnet wird. Ist blk = Ø , so wird als Quelle der **tib** angenommen.
Vergleiche **word** .

pad (-- addr)

liefert die Adresse des temporären Speichers **PAD** . Die Adresse von **PAD** ändert sich, wenn der Dictionarypointer **DP** verändert wird, z.B. durch **ALLOT** oder , (Komma).

input (-- addr) U

addr ist die Adresse einer Uservariablen, die einen Zeiger auf ein Feld von vier Kompilationsadressen enthält, die für ein Eingabegerät die Funktionen **KEY KEY? DECODE** und **EXPECT** realisieren.
Vergleiche die Beschreibung der INPUT- und OUTPUT-Struktur.

keyboard (--)

ein mit **INPUT:** definiertes Wort, das die Tastatur als Eingabegerät setzt. Die Worte **KEY KEY? DECODE** und **EXPECT** beziehen sich auf die Tastatur.
Siehe **(KEY (KEY? (DECODE** und **(EXPECT** .

empty-keys (--)

löscht den Tastaturpuffer über den Int.21h, Fct.ØC, AL=Ø.

(key? (-- flag) "(key-question"

Das Flag ist **TRUE** , wenn eine Taste gedrückt wurde, sonst **FALSE** .
Vergleiche **KEY?** .

key? (-- flag) "key-question"

Das Flag ist **TRUE** , falls ein Zeichen zur Eingabe bereitsteht, sonst ist flag **FALSE** . Eines der INPUT-Worte.

(key (-- 16b) "(key"

wartet auf einen Tastendruck. Während der Wartezeit wird **PAUSE** ausgeführt.
Zeichen des erweiterten ASCII-Zeichensatzes werden in den unteren 8 Bit von 16b übergeben. Funktionstasten liefern negative Werte.Steuerzeichen werden nicht ausgewertet, sondern unverändert abgeliefert.
Vergleiche **KEY** .

key (-- 16b) 83

empfängt ein Zeichen vom Eingabegerät. Es wird kein Echo ausgesandt. **KEY** wartet, bis tatsächlich ein Zeichen empfangen wurde.
Die niederwertigen 8 Bit enthalten den ASCII-Code des zuletzt empfangenen Zeichens. Alle gültigen ASCII-Codes können empfangen werden.
Die oberen 8 Bit enthalten systemspezifische Informationen, den Tastatur-Scancode.
Die Funktionstasten werden als negative Zahlenwerte zurückgegeben, so daß beispielsweise auf einer PC/XT-Tastatur folgende Tasten entsprechende 16Bit-Werte liefern :

Tasten F1 - F10:	-59 - -68
Cursorblock:	-71 (home) - -81 (pgdn)
Zahlen Ø - 9 :	48 (Ø) - 57 (9)

KEY ist eines der INPUT-Worte.

(decode (addr posØ key -- addr pos1) "(decode"

wertet **key** aus. **key** wird in der Speicherstelle addr+pos1 abgelegt und als Echo auf dem Bildschirm ausgegeben. Die Variable **SPAN** und pos werden inkrementiert. Backspace löscht das Zeichen links vom Cursor und dekrementiert pos1 und **SPAN** .
Vergleiche **INPUT:** und **(expect** .

(expect (addr len --) "(expect"

erwartet len Zeichen vom Eingabegerät, die ab addr im Speicher abgelegt werden. Ein Echo der Zeichen wird ausgegeben. Return beendet die Eingabe vorzeitig. Ein abschließendes Leerzeichen wird statt des <CR> ausgegeben. Die Länge der Zeichenkette wird in der Variablen **SPAN** übergeben.
Vergleiche **EXPECT** .

expect (Zieladresse maxAnzahl --) 83

empfängt maxAnzahl Zeichen und speichert sie ab der Zieladresse im Speicher, ohne ein count byte einzubauen. Der count wird statt dessen in der Variablen **span** abgelegt. Ist maxAnzahl = Ø , so werden keine Zeichen übertragen.
Die Übertragung wird beendet, wenn ein **#CR** erkannt oder maxAnzahl Zeichen übertragen wurden. Das **#CR** wird nicht mit abgespeichert.
Alle Zeichen werden als Echo, statt des **#CR** wird ein Leerzeichen ausgegeben.
Dies ist das **expect** aus der Literatur, mit dem man an beliebigen Stellen im Speicher Eingabepuffer jeder Länge anlegen kann. Eines der INPUT-Worte.
Vergleiche **SPAN** .

span (-- addr) 83

Der Inhalt der Variablen **SPAN** gibt an, wieviele Zeichen vom letzten **EXPECT** übertragen wurden; in ihr ist die Anzahl der tatsächlich eingegebenen Zeichen des zuletzt ausgeführten **expect** abgelegt.
span wird ausgelesen, um zu sehen, ob die erwartete Zeichenzahl empfangen oder die Eingabe vorher mit <CR> abgebrochen wurde. Bei **span** ist zu beachten, daß es direkt nach einem **expect** benutzt werden muß, weil auch einige FORTH-Systemwörter **span** benutzen.

>expect (Zieladresse maxAnzahl.Zeichen --)

legt ebenfalls einen zu erwartenden String an einer Zieladresse ab, allerdings wird jetzt das count-byte eingebaut.

nullstring? (addr -- addr ff ¦ addr tf)

prüft, ob der counted String an der gegebenen Adresse die Länge NULL hat. Wenn dies der Fall ist, wird **true** hinterlegt, ansonsten bleibt addr erhalten und **false** wird übergeben.

stop? (-- flag) "stop-question"

Ein komfortables Wort, das es dem Benutzer gestattet, einen Programmablauf anzuhalten oder zu beenden.
Steht vom Eingabegerät ein Zeichen zur Verfügung, so wird es eingelesen. Ist es **#ESC** oder CTRL-C , so ist Flag **TRUE** , sonst wird auf das nächste Zeichen gewartet. Ist dieses jetzt **#ESC** oder CTRL-C , so wird **STOP?** mit **TRUE** verlassen, sonst mit **FALSE** .
Steht kein Zeichen zur Verfügung, so ist Flag **FALSE** .

source (-- addr len)

übergibt Adresse und Länge des Quelltextes; wenn die Blocknummer = Ø ist, entsprechen diese Angaben denen des Text-Eingabe-Puffers TIB ; das ist der Grund, warum Block Ø niemals geladen werden kann. Ist die Blocknummer nicht NULL, beziehen sich die Angaben auf den Massenspeicher.

word (delim -- addr)

durchsucht den Quelltext (siehe SOURCE) nach einem frei wählbaren Begrenzer und legt den gefundenen String in der counted Form im Speicher an der Adresse here ab.
Führende Zeichen vom Typ delim werden ignoriert. Deshalb erwartet word immer einen Delimiter und übernimmt nicht automatisch den üblichen Begrenzer, das Leerzeichen. Daher typisch:

```
... bl word ...
```

parse (delim -- addr len)

erwartet ebenfalls ein Zeichen als Begrenzer, liefert die nächste delim begrenzte Zeichenkette des Quelltextes aber schon in der üblichen Form addr count passend für type.
Die Länge ist Null, falls der Quelltext erschöpft oder das erste Zeichen der delimiter ist. parse verändert die Variable >in .
Typisches Auftreten:

```
: .(  ASCII ) parse  type ;
```

name (-- string)

holt den nächsten durch Leerzeichen abgeschlossenen String aus dem Quelltext, wandelt ihn in Großbuchstaben um und hinterläßt die Adresse, ab der der String im Speicher steht.
Siehe word .

find (string -- string ff ¦ cfa tf)

erwartet die Adresse eines counted Strings. Dieser String wird in der aktuellen Suchreihenfolge gesucht und das Resultat dieser Suche übergeben.
Wird das Wort gefunden, so liegt die cfa und ein Pseudoflag tf bereit:
- Ist das Wort immediate, so ist das flag positiv, sonst negativ.
- Ist das Wort restrict, so hat das flag den Betrag 2, sonst Betrag 1.

Typisch:

```
... (name) find ...
... " <String>" find ...
... name  find ...
```

execute (addr --)

Das durch addr gekennzeichnete Wort wird aufgerufen bzw. ausgeführt. Dabei kann es sich um ein beliebiges Wort handeln. Eine Fehlerbedingung existiert, falls addr nicht cfa eines Wortes ist.

Typisch: ' <name> execute

perform (addr --)

ist ein **execute** für einen Vektor und entspricht **@ execute** . Typisch beim Einsatz einer Prozedurvariable:

```
Variable <vector>
' noop <vector> !
(vector) perform
```

Hier wird zuerst eine Variable angelegt und anschließend dieser Variablen mit Hilfe des ' ("Tick") die CFA eines Wortes zugewiesen. Dieses Wort kann nun über die Adresse der Prozedurvariablen und **PERFORM** ausgeführt werden. Allgemein wird diese Variable mit **NOOP** abgesichert.

query (--) 83

ist die allgemeine Texteingabe-Prozedur für den Texteingabe-Puffer. **query** stoppt die Programmausführung und liest eine Textzeile von max. 8Ø Zeichen mit Hilfe von **expect** ein.

Die Zeichen werden von einem Eingabegerät geholt und in den Text-Eingabe-Puffer übertragen, der bei **TIB** beginnt. Die Übertragung endet beim Empfang von **#CR** oder wenn die Länge des Text-Eingabe-Puffers erreicht wird. Der Inhalt von **>IN** und **BLK** wird zu Ø gesetzt und **#TIB** enthält die Zahl der empfangenen Zeichen.

Zu beachten ist, daß der Inhalt von **TIB** überschrieben wird. Damit werden auch alle Worte im **TIB** , die noch nicht ausgeführt wurden, ebenfalls überschrieben. Um Text aus dem Puffer zu lesen, kann **WORD** benutzt werden.

Siehe **EXPECT** .

interpret (--)

ist der allgemeine Text-Interpreter des FORTH-Systems und beginnt die Interpretation des Quelltextes bei dem Zeichen, das durch den Inhalt der Variablen **>in** bezeichnet wird. **>in** indiziert relativ zum Beginn des Blockes, der durch die Variable **blk** gekennzeichnet wird. Ist blk = Ø , so interpretiert **interpret** die max. 8Ø Zeichen im **tib** .

5.7 Ausgabe von Zeichen

output (-- addr) U

addr ist die Adresse einer Uservariablen, die einen Zeiger auf sieben Kompilationsadressen enthält, die für ein Ausgabegerät die Funktionen **EMIT CR TYPE DEL PAGE AT** und **AT?** realisieren.
Vergleiche die gesonderte Beschreibung der Output-Struktur.

display (--)

ist ein mit **OUTPUT:** definiertes Wort, das den Bildschirm als Ausgabegerät setzt, wenn es ausgeführt wird. Die Worte **EMIT CR TYPE DEL PAGE AT** und **AT?** beziehen sich dann auf den Bildschirm.

(emit (8b --) "(emit"

gibt 8b auf dem Bildschirm aus. Ein **PAUSE** wird ausgeführt. Alle Werte werden als Zeichen ausgegeben, Steuercodes sind nicht möglich, d.h. alle Werte < $2Ø werden als PC-spezifische Zeichen ausgegeben.
Vergleiche **EMIT** .

emit (16b --) 83

gibt die unteren 8 Bit an das Ausgabegerät. Eines der OUTPUT-Worte.

charout (8b --)

gibt 8b auf Standard-I/O-Gerät aus. Dazu wird die Fct.Ø6 des Int21h benutzt. ASCII-Werte < $2Ø werden als Steuercodes interpretiert. **CHAROUT** ist ein Primitiv für die Ausgabe-routinen und sollte vermieden werden, da die Ausgabe dann nicht über die Videotreiber läuft.

tipp (addr count --)

ist die Primitiv-Implementierung für **TYPE** über den MSDOS Character-Output.
Siehe auch **DISPLAY** in KERNEL.SCR.

(type (addr len --)

gibt den String, der im Speicher bei addr beginnt und die Länge len hat, auf dem Bildschirm aus. Ein **PAUSE** wird ausgeführt.
Vergleiche **TYPE OUTPUT:** und **(EMIT** .

type (addr +n --) 83

sendet +n Zeichen, die ab addr im Speicher abgelegt sind, an ein Ausgabegerät. Ist +n = Ø , so wird nichts ausgegeben.

ltype (seg:addr Länge --)
ist ein segmentiertes TYPE .

space (--)
gibt ein Leerzeichen aus.

spaces (Anzahl --)
gibt eine bestimmte Anzahl von Leerzeichen aus.

6. Strings

Hier befinden sich grundlegende Routinen zur Stringverarbeitung.
Vor allem wurden auch Worte aufgenommen, die den Umgang mit den vom Betriebssystem geforderten Ø-terminated Strings ermöglichen. FORTH hat hier gegenüber C den Nachteil, daß FORTH-Strings standardmäßig mit einem Count-Byte beginnen, das die Länge des Strings enthält. Ein abschließendes Zeichen (z.B. ein Null-Byte) ist daher unnötig. Da das Betriebssystem aber in C geschrieben wurde, müssen Strings entsprechend umgewandelt werden.
Standardmäßig arbeitet FORTH mit counted Strings, die lediglich durch eine Adresse gekennzeichnet werden. Das Byte an dieser Adresse enthält die Angabe, wie lang die Zeichenkette ist. Auf dieses count byte folgt dann die Zeichenkette selbst. Dadurch ist die Länge eines Standard-Strings in FORTH auf 255 Zeichen begrenzt.
Die kürzeste Zeichenkette ist ein String der Länge NULL, für dessen Überprüfung der Befehl **NULLSTRING?** zur Verfügung steht.

addr

So sieht der String FORTH an der Adresse addr im Speicher unter FORTH aus.

." (--)
ist nur während des Compilierens in Wörtern zu verwenden. Die gewünschte Zeichenkette wird von ." und " eingeschlossen, wobei das Leerzeichen nach ." nicht mitzählt. Wenn das Wort abgearbeitet wird, so wird der String zwischen ." und " ausgegeben:

```
: Hallo ." Ich mache FORTH" ;
```

" (-- addr) C,I "string"
(--) compiling
liest den Text bis zum nächsten " und legt ihn als counted string im Dictionary ab. Kann nur während der Kompilation ver-wendet werden. Zur Laufzeit wird die Startadresse des counted string auf den Stack gelegt. Das Leerzeichen, das auf das erste Anführungszeichen folgt, ist nicht Bestandteil des Strings. Wird in der folgenden Form benutzt:

```
: Hallo " Ich mache FORTH" count type ;
```

," (--) "string literal"
speichert einen counted String ab HERE . Wird in der folgenden Form benutzt :
```
Create Hallo ," Ich mache FORTH"
Hallo count type
```

nullstring? (addr -- addr false ¦ true)
prüft, ob der counted String an der Adresse addr ein String der Länge NULL ist. Wenn dies zutrifft, wird TRUE übergeben. Anderenfalls bleibt addr erhalten und FALSE wird übergeben.

"lit (-- addr)
wird in compilierenden Worten benutzt, die auf INLINE-Strings zugreifen. Siehe auch ?" und " .

.((--)
druckt den nachfolgenden String immer sofort aus. Der String wird von) beendet.
Anwendung: .(Ich mache FORTH)

((--)
leitet einen Kommentar ein, d.h. Text, der vom Compiler ignoriert wird.

) beendet einen Kommentar. Dieses Wort ist aber selbst kein FORTH-Befehl, muß deshalb auch nicht mit Leerzeichen abgetrennt sein.

6.1 String-Manipulationen

Hier im Glossar bezeichnet der Stackkommentar (string --) die Adresse eines counted Strings, dagegen (addr len --) die Charakterisierung durch die Anfangsadresse der Zeichenkette und ihre Länge.

Keine Stringvariable - Benutze:
```
: String:  Create dup , 0 c, allot  Does> 1+ count ;
```

caps (-- addr)
liefert die Adresse einer Variablen, die angibt, ob beim Stringvergleich auf Groß- und Kleinschreibung geachtet werden soll.

capital (char -- char')
Die Zeichen im Bereich a bis z werden in die Großbuchstaben A bis Z umgewandelt, ebenso die Umlaute äöü . Andere Zeichen werden nicht verändert.

upper (addr len --)
wandelt einen String in Großbuchstaben um. Es werden keine Parameter zurückgegeben.

capitalize (string -- string)
ist durch UPPER ersetzt worden. Bei Bedarf kann es folgendermaßen definiert werden:

```
: capitalize  ( string -- string )  dup count upper ;
```

/string (addrØ len index -- addr1 restlen) "cut-string"
verkürzt den durch addrØ und len gekennzeichneten String an der mit index angegebenen Stelle und hinterläßt die Parameter des rechten Teilstrings.

-trailing (addrØ lenØ -- addr1 len1) "minus-trailing"
kürzt einen String um die Leerzeichen, die sich am Ende des Strings befinden. Anschließend liegen die neueen Parameter auf dem Stack, passend z.B. für TYPE .

scan (addrØ len char -- addr1 restlen)
durchsucht einen mit addr und len angegebenen String nach einem Zeichen char.
Wird das Zeichen gefunden, so wird die Adresse der Fundstelle und die Restlänge einschließlich des gefundenen Zeichens übergeben.
Wird char nicht gefunden, so ist addr1 die Adresse des ersten Bytes hinter dem String und die Restlänge ist NULL.

skip (addrØ len char -- addr1 restlen)
durchsucht einen mit addrØ und len angegebenen String nach einer Abweichung von dem gegebenen Zeichen char.
Beim ersten abweichenden Buchstaben wird die Adresse der Fundstelle und die Restlänge ohne das abweichende Zeichen übergeben.
Besteht der gesamte String aus dem Zeichen char, so ist addr1 die Adresse des Bytes hinter dem String und die Restlänge ist NULL.

?" (--)
gibt die Position eines Zeichen in einem String an. Dieses ?" wird in Definitionen benutzt, um z.B. bei

```
: vocal?   ( char -- index )   capital ?" aeiou"  ;
Ascii ( char)  vocal?
```

festzustellen, ob es sich bei dem gegebenen Zeichen um einen Vokal handelt (index ungleich NULL) oder um welchen Vokal es sich handelt (a=1, e=2, i=3 etc.) .

count (string -- addr+1 len)
wandelt eine String-Addresse in die Form addr len um, die für z.B. TYPE notwendig ist.

bounds (addr len -- limit start)
ist definiert als

```
: bounds   over + swap ;
```

und kann z.B. dazu benutzt werden, die Parameter einer Zeichenkette in DO..LOOP-gemäße Argumente umzurechnen:

```
: upper ( addr len -- )
    bounds ?DO
        I c@ capital I c!
    LOOP ;
```

type (addr len --)
gibt eine Zeichenkette aus, die an addr beginnt und die angebene Länge hat.

>type (addr len --)
ist ein **TYPE** , das zuerst den String nach **PAD** kopiert. Dies ist in Multitasking-Systemen wichtig und wird folgendermaßen definiert:

```
: >type   ( addr count -- )
    pad place
    pad count type ;
```

place (addrØ len dest.addr --)
legt eine Zeichenkette, die durch ihre Anfangsadresse und ihre Länge gekennzeichnet ist, an der Zieladresse als counted String ab.
PLACE wird in der Regel benutzt, um Text einer bestimmten Länge als counted String abzuspeichern; dabei darf dest.addr gleich, größer oder kleiner als addrØ sein.
Die Wirkungsweise von **PLACE** wird in der Definition von **>EXPECT** deutlich, wo die Anzahl der von **EXPECT** eingelesenen Zeichen mit **SPAN @** ausgelesen und von **PLACE** an der Zieladresse eingetragen wird:

```
: >expect ( addr len --)
    stash expect  span @  over place ;
```

attach (addr len string --)
fügt einen String an den durch addr und len gekennzeichneten String an, wobei der Count addiert wird.

append (char string --)
fügt ein Zeichen an den angegebenen String an und in-crementiert den Count.

detract (string -- char)
zieht das erste Zeichen aus einer Zeichenkette und de-crementiert den Count.

6.2 Suche nach Strings

6.2.1 In normalem Fließtext:

match (buf.addr buf.len string -- match.addr buf.rest)
Der Pufferbereich an buf.addr mit der Länge buf.len wird nach einer vollständigen Übereinstimmung mit dem counted String überprüft. Die Adresse der Übereinstimmung und die Restlänge des Puffers wird übergeben. **MATCH** befindet sich im Vokabular **EDITOR** .

search (buf len string -- offset flag)
durchsucht einen Speicherbereich mit angegebener Länge nach einem counted string.
Der Abstand vom Anfang des Speicherbereiches und ein Flag werden übergeben . Ist das Flag wahr, wurde der String gefunden, sonst nicht. Der Quelltext von **SEARCH** befindet sich im Editor und ist mit dessen Suchfunktion leicht aufzufinden.

6.2.2 Im Dictionary:

(find (string thread -- string ff ¦ nfa tf)
erwartet die Adresse eines counted string und eine Suchreihe. Wird das Wort in dieser Reihe gefunden, so wird ein true flag und die nfa übergeben; wird es nicht gefunden, wird ein false flag übergeben und die Stringadresse bleibt erhalten.
Typische Anwendung:

```
... last @  current @  (find ...
```

find (string -- string.addr ff ¦ cfa tf)

erwartet die Adresse eines counted Strings. Dieser String wird in der aktuellen Suchreihenfolge gesucht und das Resultat dieser Suche übergeben.
Wird das Wort gefunden, so liegt die cfa und ein Pseudoflag tf bereit:
- Ist das Wort immediate, so ist das Flag positiv, sonst negativ.
- Ist das Wort restrict, so hat das Flag den Betrag 2, sonst Betrag 1.

Typische Anwendung:

```
... ( name ) find ...
... " <string>" find ...
... name find ...
```

6.3 Ø-terminated Strings

Es gibt noch eine andere Darstellungsform für Strings, die beispielsweise für MS-DOS geeigneter ist. Diese Strings werden zwar ebenfalls durch eine Adresse gekennzeichnet; diese Adresse enthält aber kein count byte . Statt dessen werden diese Zeichenketten mit einem Nullbyte abgeschlossen.

asciz (-- asciz)
(--) compiling

holt das nächste Wort des Quelltextes (TIB oder BLOCK) in den Speicher und legt es als nullterminierten String (mit abschließendem Null-Byte) im Dictionary ab. Diese Adresse asciz im Dictionary wird zurückgeliefert.
Wird in der folgenden Form benutzt :

```
asciz <string>
```

Vergleiche ," .

>asciz (string addr.dest -- asciz)

wandelt den counted String an der Adresse STRING um in einen nullterminierten String, der an der gegebenen Ziel-Adresse ADDR abgelegt wird. Die Länge des Strings im Speicher bleibt gleich.
An der zurückgelieferten Adresse asciz liegt der neue String, wobei die übergebene Zieladresse und die zurückgelieferte ASCIZ-Adresse nicht gleich sein müssen.
>ASCIZ ist das grundlegende Wort, um FORTH-Strings in MS-DOS-Strings umzuwandeln.

counted (asciz -- addr len)

wird benutzt, um die Länge eines mit einem Nullbyte terminierten Strings zu bestimmen. asciz ist die Anfangsadresse dieses Strings, addr und len sind die Parameter, die z.B. von TYPE erwartet werden.

6.4 Konvertierungen: Strings -- Zahlen

6.4.1 String in Zahlen wandeln

digit? (char -- digit true)
(char -- false)

prüft, ob das Zeichen char eine gültige Ziffer entsprechend der aktuellen Zahlenbasis in **BASE** ist. Ist das der Fall, so wird der Zahlenwert der Ziffer und **TRUE** auf den Stack gelegt. Ist char keine gültige Ziffer, wird **FALSE** übergeben.

accumulate (+dØ addr char -- +d1 addr)

dient der Umwandlung von Ziffern in Zahlen.
Multipliziert die Zahl +dØ mit **BASE** , um sie eine Stelle in der aktuellen Zahlenbasis nach links zu rücken, und addiert den Zahlenwert von char. char muß eine in der Zahlenbasis gültige Ziffer darstellen, addr wird nicht verändert.
Wird z.B. in **CONVERT** benutzt. .

convert (+dØ addrØ -- +d1 addr1)

wandelt den Ascii-Text ab addrØ +1 in eine Zahl entsprechend der Zahlenbasis **BASE** um. Der enstehende Zahlenwert und die Adresse des ersten nicht wandelbaren Zeichens im Text werden hinterlassen.

number? (addr -- d Ø> ¦ n Ø< ¦ addr false)

wandelt den counted String bei der Adresse addr in eine Zahl n um. Die Umwandlung erfolgt entsprechend der Zahlenbasis in **BASE** oder wird vom ersten Zeichen im String bestimmt.
Enthält der String zwischen den Ziffer auch die Asciizeichen für Punkt oder Komma, so wird er als doppelt genaue Zahl interpretiert und Ø> gibt die Zahl der Ziffern hinter dem Punkt einschließlich an.
Sonst wird der String in eine einfach genaue Zahl n umgewandelt und eine Zahl kleiner als Null hinterlassen.
Wenn die Ziffern des Strings nicht in eine Zahl umgewandelt werden können, bleibt die Adresse des Strings erhalten und **FALSE** wird auf den Stack gelegt.

Die Zeichen, die zur Bestimmung der Zahlenbasis dem Ziffern-string vorangestellt werden können, sind:

```
%   ( Basis 2    "binär")
&   ( Basis 10   "dezimal")
$   ( Basis 16   "hexadezimal")
h   ( Basis 16   "hexadezimal")
```

Der Wert in **BASE** wird dadurch nicht verändert.

number (addr -- d)

wandelt den counted String bei der Adresse addr in eine Zahl d um. Die Umwandlung erfolgt entsprechend der Zahlenbasis in **BASE** . Eine Fehlerbedingung besteht, wenn die Ziffern des Strings nicht in eine Zahl verwandelt werden können. Durch Angabe eines Präfix (siehe **NUMBER?**) kann die Basis für diese Zahl modifiziert werden.

dpl (-- addr) "decimal point location"

ist eine Variable, die die Stellung des Dezimalpunktes angibt.

Ein Beispiel der Umwandlung von Zeichen in Zahlen:

In FORTH wird die Eingabe von Zahlen oft mit der allgemeinen Texteingabe und über die Befehle zur Umwandlung von Strings in Zahlen realisiert. In der Literatur wird dazu oft diese Lösung mit **QUERY** angeboten:

```
: in#   ( <string> -- d tf  n tf  addr ff )
    query  bl word  number?  ;
```

Diese Lösung ist ungünstig, da QUERY den TIB löscht. Zugleich stellt die Definition von **NUMBER?** eine unglückliche Stelle im volksFORTH dar.
Es gibt im Laxen&Perry-F83 ein Wort gleichen Namens, das ganz anders (besser!) mit den Parametern umgeht. Hier folgt die Definition des F83-NUMBER?, aus dem volksFORTH **NUMBER?** aufgebaut:

```
: F83-NUMBER?  ( string -- d f )
   number?   ?dup IF  0< IF  extend  THEN  true exit  THEN
   drop 0 0 false ;
```

Damit stellt das Wort **INPUT#** eine wenig aufwendige Zahleneingabemöglichkeit für 16/32Bit-Zahlen dar:

```
\ input#                                        jrg 29jul88
: input#  ( <string> -- d f )
   pad  c/l 1-  >expect          \ get 63 char maximal
   pad  F83-number?   ;          \ converts string > number
```

So kann der Anwender das übergebene Flag auswerten und die doppelt-genaue Zahl entsprechend seinen Vorstellungen einsetzen, im einfachsten Fall mit **DROP** zu einer einfach-genauen Zahl machen.

6.4.2 Zahlen in Strings wandeln

(+dØ -- +d1) 83 "sharp"

Der Rest von +dØ geteilt durch den Wert in **BASE** wird in ein Ascii-Zeichen umgewandelt und dem Ausgabestring in Richtung absteigender Adressen hinzugefügt. +d1 ist der Quotient und verbleibt auf dem Stack zur weiteren Bearbeitung. Üblicherweise zwischen **<#** und **#>** benutzt.

#s (+d -- Ø Ø) 83 "sharp-s"

+d wird mit **#** umgewandelt, bis der Quotient zu Null geworden ist. Dabei wird jedes Zwischenergebnis in ein Ascii-Zeichen umgewandelt und dem String für die strukturierte Zahlenausgabe angefügt. Wenn +d von vornherein den Wert Null hatte, wird eine einzelne Null in den String gegeben. Wird üblichwerweise zwischen **<#** und **#>** benutzt.

hold (char --) 83

Das Zeichen char wird in den Ausgabestring für die Zahlenausgabe eingefügt. Wird üblicherweise zwischen **<#** und **#>** benutzt.

sign (n --) 83

Wenn n negativ ist, wird ein Minuszeichen in den Ausgabestring für die Zahlenausgabe eingefügt. Wird üblicherweise zwischen **<#** und **#>** benutzt.

#> (32b -- addr len) 83 "sharp-greater"

Am Ende der strukturierten Zahlenausgabe wird der 32b Wert vom Stack entfernt. Hinterlegt werden die Adresse des erzeugten Ausgabestrings und eine positive Zahl als die Anzahl der Zeichen im Ausgabestring, passend z.B. für **TYPE** .

7. Umgang mit Dateien

Das File-Interface wurde grundlegend überarbeitet.
Auf der Benutzerebene stehen die gleichen Worte wie im volksFORTH 3.8Ø für den ATARI und für CP/M zur Verfügung; die darunterliegende Implementation wurde jedoch grundlegend geändert, so daß jetzt in FORTH auch sequentielle Files, die nicht die starre BLOCK-Struktur haben, manipuliert werden können.

Damit ist es endlich möglich, auch volksFORTH für kleine Hilfsprogramme zu verwenden, die mit anderen Programmen erstellte Files "bearbeiten" und durch den Befehl **SAVESYSTEM** als "standalone"-Programm abgespreichert wurden.
Besonders weitreichende Möglichkeiten erschließen sich dadurch, daß beim Aufruf von volksFORTH auf der Betriebssystemebene noch eine ganze Kommandozeile mit übergeben werden kann, die dann unmittelbar nach dem Booten von FORTH ausgeführt wird. Durch die Systemvariable **RETURN_CODE** kann nach Verlassen des FORTHProgramms ein Wert an MS-DOS zurückgegeben werden, der mit dem Batch-Befehl ERRORLEVEL abgefragt werden kann.

Darüberhinaus ist es auch möglich, mit dem Befehl **MSDOS** aus dem FORTH heraus eine weitere COMMAND.COM shell aufzurufen und später mit EXIT wieder ins FORTH zurückzukehren, wobei der Bildschirm, der zum Zeitpunkt des Aufrufs bestand, wiederhergestellt wird.

Selbstverständlich kann neben MSDOS selber auch jedes andere beliebige Anwendungsprogramm aufgerufen werden - auch eine weitere Inkarnation des FORTH-Systems - so daß sich mit diesen Möglichkeiten die Begrenzungen überwinden lassen, die in dem beschränkten Adreßraum von 64k liegen. Auch komplizierte Overlaystrukturen sind nicht mehr notwendig, es werden einfach aus einem zentralen "Verwaltungsprogramm" heraus spezielle FORTH-Anwendungsprogramme aufgerufen.

Das Fileinterface des volksFORTH benutzt die Dateien des MSDOS und dessen Subdirectories.

Dateien bestehen aus einem FORTH-Namen und einem MSDOS-Namen, die nicht übereinstimmen müssen.
Ist das FORTH-Wort, unter dem ein File zugreifbar ist, gemeint, so wird im folgenden vom (logischen) FORTH-File gesprochen. Dies entspricht einer Dateivariablen in den PASCAL-ähnlichen Sprachen. Der Zugriff auf eine Datei erfolgt daher in volksFORTH über den Namen eines Files ; dieser Namen ermöglicht dann den Zugriff entweder auf den Datei-Steuerblock (FCB) oder die MSDOS-Handle-Nummer.
Ist das File auf der Diskette gemeint, das vom MSDOS verwaltet wird, so wird vom (physikalischen) DOS-File gesprochen.