

Durch das Nennen des FORTH-Namens wird das FORTH-File (und das zugeordnete DOS-File) zum aktuellen File, auf das sich alle Operationen wie LIST, LOAD, CONVEY usw. beziehen.

Beim Öffnen eines Files wird die mit PATH angegebene Folge von Pfadnamen durchsucht. Dabei ist PATH ein von MSDOS völlig unabhängiger Pfad, so daß verschiedene Projekte auch mit verschiedenen FORTH-Systemen über ihren eigenen Pfad verwaltet werden können.

Der gültige MSDOS-Pfad wird davon nicht beeinflusst. Ist eine Datei einmal geöffnet, so kann diese Folge beliebig geändert werden, ohne daß der Zugriff auf das File behindert wird. Aus Sicherheitsgründen empfiehlt es sich aber, Files so oft und so lange wie irgend möglich geschlossen zu halten. Dann kann eine Änderung der Folge von Pfadnamen in PATH dazu führen, daß ein File nicht mehr gefunden wird.

volksFORTH bearbeitet seine Quelltexte in sogenannten Screen-Files, die üblicherweise die Endung .SCR haben.

Dies sind Files, die in 1 KB große Screens aufgeteilt sind, jeweils in 16 Zeilen zu je 64 Zeichen strukturiert.

Üblicherweise enthält der Screen 0 eines jeden Files eine kurze Erklärung über den Inhalt des Files - dies ist auch deshalb sinnvoll, da der Screen 0 eines Files nie geladen werden kann.

Screen 1 enthält üblicherweise den sogenannten "loadscreen". Dieser steuert den Ladevorgang des gesamten Files. Ein Beispiel für einen solchen LOADSCREEN ist der Scr1 der Datei VOLKS4TH.SYS, der die aktuelle Arbeitsumgebung VOLKS4TH.COM zusammenstellt.

In Zeile 0 eines jeden Screens ist ein Kommentar über den Inhalt des Screens und das Datum der letzten Änderung enthalten.

Um - vor allem auf Festplatten - Directories anzusprechen, gibt es verschiedene Methoden. Um beispielsweise Files in einem Directory "WORK" zu suchen oder anzulegen, geben Sie ein :

```
cd work
```

Jetzt werden alle Files und Sub-Directories im Directory WORK gesucht oder neu geschaffen. Ist das Directory WORK noch nicht vorhanden, so erzeugen Sie es mit :

```
md work
```

Wenn Sie Dateien auf einem bestimmten Laufwerk, z.B. A: , ansprechen wollen, geben Sie ein:

```
A:
```

Hierbei wird A: zum aktuellen Laufwerk gemacht.

Im einfachsten Fall existiert die zu benutzende Datei auf dem Massenspeicher und der Datei-Name bereits im volksFORTH-Dictionary. Solche Files sprechen Sie an, indem Sie einfach den Filenamen eingeben.

Mit dem Wort `files` zeigt Ihnen volksFORTH eine Liste dieser bereits im FORTH-System vorhandenen Dateien und Dateinamen, zusammen mit den zugeordneten DOS-Dateien und den Handle-Nummern.

Melden Sie irrtümlich eine Datei an, deren Name nicht im FORTH-Wörterbuch gefunden wird, erscheint die übliche Fehlermeldung, das "?".

Um dagegen zu kontrollieren, ob eine Datei auf Disk existiert, benutzen Sie `dir`. Soll in einem solchen Fall ein File, das auf der Disk schon existiert, neu von volksFORTH bearbeitet werden, so wird es durch `use <filename>` zum sogenannten "aktuellen" File gemacht, in die Liste `FILES` und ins Wörterbuch eingetragen.

So macht `use test.scr` das MSDOS-File `TEST.SCR` zum aktuellen File, auf das sich solange alle weiteren File-Operationen beziehen, bis ein anderes File zum aktuellen File gemacht wird. Der Zugriff auf verschiedene Laufwerke ist für die Laufwerke A: bis H: vordefiniert.

Das Wort `use` erzeugt deshalb im FORTH-System das Wort `TEST.SCR`, falls es noch nicht vorhanden war. Wissen Sie also nicht mehr, ob Sie ein File schon benutzt haben, so können Sie das Wort `use` voranstellen, mit `files` nachsehen oder probierhalber den Datei-Namen eingeben.

Möchten Sie ein neue Datei oder ein eigenes Screenfile anlegen, so wählen Sie einen Namen und benutzen:

```
makefile <filename>
```

Danach erfolgen die Schreib/Lesezugriffe des DOS auf <filename>. Bei einem Blockfile schätzen Sie die notwendige Größe `nn` in Blocks ab und geben ein:

```
makefile <filename> nn more (z.B. 20 more)
```

Dabei ist der Name maximal 8 Buchstaben und die Extension maximal 3 Buchstaben lang. Diese Extension kann, muß aber nicht angegeben werden. Als Konvention wird vorgeschlagen, daß Files, die FORTH-Screens, also Quelltexte, enthalten, die Endung `.SCR` erhalten. Files, die Daten enthalten, die nicht unmittelbar lesbar sind, sollten auf `.BLK` enden.

Auf `makefile` hin legt das System eine neue Quelltext-Datei von beispielsweise 20 KB Größe an. Das System geht bei 20KB davon aus, daß 10 KB Programmtexte in den ersten zehn Blöcken (0 - 9) und 10 KB Kommentare in den Shadow-Screens 10 - 19 eingetragen werden. Das File wird nur logisch (!) in der Mitte geteilt. Wenn auf allen Screens nur Quelltexte eingetragen werden, so hat das keine negativen Konsequenzen, Sie bekommen dann nur im Editor mit `SHIFT-F9` keinen Kommentar angezeigt.

Sie haben nun eine Datei, das die Screens 0..19 enthält. Geben Sie jetzt ein:

```
1 1 oder 1 edit
```

Nach der Abfrage einer dreibuchstabigen Kennung, die Sie mit `<CR>` beantworten können, editieren Sie jetzt den Screen 1 Ihres neuen Files `test.scr`. Sie können, falls der Platz nicht ausreicht, Ihr File später einfach mit `more` verlängern. Ein



File kann leider nicht verkürzt werden.

Mit dem volksFORTH-Editor können Sie selbstverständlich auch stream files ohne die FORTH-typische Blockstruktur ansehen und editieren.

Drei wichtige Wörter für screen files sind:

Mit `<nn> list` wird Screen nn auf dem Bildschirm angezeigt - also zum Beispiel mit `1 list` der Screen 1 des Files TEST.SCR.

Mit `<nn> load` wird ein Screen nn geladen, d.h. durch den FORTH-Compiler in das Wörterbuch compiliert.

Mit `include <filename>` kann man unkompliziert ein ganzes Screenfile laden. Diese Operation ist der Sequenz `use <filename> 1 load` äquivalent.

Beim Vergessen eines FORTH-Files mit Hilfe von `FORGET`, `EMPTY` usw. werden automatisch alle Blockpuffer, die aus diesem File stammen, gelöscht und, wenn sie geändert waren, auf die Diskette zurückgeschrieben. Das File wird anschließend geschlossen. Bei Verwendung von `flush` werden alle Files geschlossen. `FLUSH` sollte vor jedem Diskettenwechsel ausgeführt werden, und zwar nicht nur, um die geänderten Blöcke zurückzuschreiben, sondern auch damit alle Files geschlossen werden. Sind nämlich Files gleichen Namens auf der neuen Diskette vorhanden, so wird eine abweichende Länge des neuen Files vom FORTH nicht erkannt. Bei Verwendung von `VIEW` wird automatisch das richtige File geöffnet.

Sind bereits sehr viele Files geöffnet, kommt es in der MSDOS-Grundeinstellung recht schnell zu der Fehlermeldung des Betriebssystems: `too many open files`. Dagegen hilft die Änderung der Datei `CONFIG.SYS`:

```
FILES=16
```

Eine Besonderheit volksFORTH ist der Direktzugriff auf den Massenspeicher. Das Wort `DIRECT` schaltet das FileInterface ab und der Massenspeicher wird direkt physikalisch angesprochen!

Wenn nach dem Befehl `EMPTY` die Meldung `F: direct` auftaucht, ist das Fileinterface abgeschaltet. Solch ein Direktzugriff kann wieder auf das Fileinterface umgeschaltet werden, indem man einen Dateinamen aufruft. Bei den Dateinamen in der Liste der `FILES` ist also kein `USE` notwendig.

Wenn Sie als Beispiel direkt auf Drive A: zugreifen wollen, so ist die Syntax:

```
direct a:
```

Nun kann mit `list`, `load`, aber auch mit `edit (!)` auf die Disk zugegriffen werden! Deshalb ist bei Festplatten besondere Vorsicht geboten, da der Schreib-/Lesezeiger direkt auf dem Bootsektor der Platte stehen kann. Sollte Ihnen dies passieren, verlassen Sie den Editor mit `CTRL-U` (`undo`) und `<ESC>`.

Beim Direktzugriff muß allerdings beachtet werden, daß es für volksFORTH nur noch 1 KB große Blöcke gibt, die hintereinander liegen und nur durch ihre Blocknummern identifiziert sind. Eine 360K DS/DD-Disk enthält dann für volksFORTH keine Files und kein Directory, sondern es steht lediglich Platz für 362 Blöcke zur Verfügung.

Es gibt zwar einige Leute, die diese direct-Funktion dazu benutzen, um mit dem FORTH-Editor die Systembereiche ihrer Festplatte zu editieren; aber normalerweise ermöglicht der Direct-Zugriff dem volksFORTH, ohne MSDOS auszukommen und den Massenspeicher sehr schnell und effizient zu bedienen.

#### Einige wichtige Worte:

call ( -- )

ermöglicht den Aufruf von \*.COM und \*.EXE-Files.

call \park würde im Root-Directory Ihrer Harddisk ein Programm namens PARK aufrufen und ausführen.

msdos ( -- )

verläßt das volksFORTH zur MSDOS-Ebene; mit exit kehren Sie ins FORTH zurück.

savesystem <name> ( -- )

schreibt eine bootbare Form des jetzt laufenden FORTH-Systems unter dem Namen <name> auf den Massenspeicher. Dabei muß <name> die Endung .COM haben, wenn dieses System später wieder unter MSDOS gestartet werden soll.

Dieses Wort wird benutzt, um fertige Applikationen zu erzeugen. In diesem Fall sollte das Wort, daß die Applikation ausführt, in 'COLD gespeichert. Ein Beispiel:

Sie haben ein Kopierprogramm geschrieben, das oberste ausführende Wort heißt COPYDISK . Mit der Befehlsfolge

```
' copydisk is 'cold
savesystem copy.com
```

erhalten Sie ein Programm namens COPY.COM , das sofort beim Start COPYDISK ausführt.

direct ( -- )

schaltet das MSDOS-Fileinterface aus und auf Direktzugriff um. Auf den Filezugriff schalten Sie durch das Nennen eines Filenamens um.

### 7.1 Pfad-Unterstützung

volksFORTH besitzt eine eigene, vollständig von MSDOS unabhängige Pfadunterstützung. Die Verwaltung von Directories entspricht in etwa der des MSDOS-Command-interpreters COMMAND.COM.

Beispiel: `PATH A:\;B:\COPYALL\DEMO;\AUTO\`

In diesem Beispiel wird zunächst die Diskstation A, dann das Directory COPYALL\DEMO auf Diskstation B und schließlich das Directory AUTO auf dem aktuellen Laufwerk gesucht.

Beachten Sie bitte das Zeichen "\" vor und hinter AUTO. Das vordere Zeichen "\" steht für das Hauptdirectory. Wird es weggelassen, so wird AUTO\ an das mit DIR gewählte Directory angehängt. Das hintere Zeichen "\" trennt den Filenamen vom Pfadnamen.

Durch die vollständige Unabhängigkeit kann z.B. volksFORTH mit dem Pfad C:\4TH;\4TH\WORK arbeiten, während MSDOS den Pfad C:\UTIL\TOOLS absucht.

`path` ( -- )

Dieses Wort gestattet es, mehrere Directories nach einem zu öffnenden File durchsuchen zu lassen. Das ist dann praktisch, wenn man mit mehreren Files arbeitet, die sich in verschiedenen Directories oder Diskstationen befinden. Es wird in den folgenden Formen benutzt:

`path dir1;\dir2\subdir2;\dir3`

Hierbei sind <dir1>, <dir2> etc. Pfadnamen. Wird ein File auf dem Massenspeicher gesucht, so wird zunächst <dir1>, dann <dir2> etc. durchsucht. Zum Schluß wird das File dann im aktuellen Directory (siehe DIR) gesucht. Beachten Sie bitte, daß keine Leerzeichen in der Reihe der Pfadnamen auftreten dürfen.

`path` ohne Argument zeigt die aktuelle Pfadangabe und druckt die Reihe der Pfadnamen aus. Dieses Wort entspricht dem Kommando PATH des Kommandointerpreters.

`path ;` löscht die aktuelle Pfadangabe und es wird nur noch das aktuelle Directory durchsucht.

### 7.2 DOS-Befehle

`dos`

Viele Worte des Fileinterfaces, die normalerweise nicht benötigt werden, sind in diesem Vokabular enthalten.

`dir`  
zeigt das Inhaltsverzeichnis des Massenspeichers. Ohne weitere Eingaben wird der aktuelle Suchpfad einschließlich des Laufwerks angezeigt.

`delete <filename>`  
löscht eine Datei `<filename>` im aktuellen Directory .

`ren <filename.alt> <filename.neu>`  
benennt eine Datei nach üblicher MSDOS-Syntax um.

`ftype <filename>`  
gibt eine Text-Datei aus. `FTYPE` wurde so genannt, weil es ein FORTH-Wort `type` gibt.

`md <dir>`  
legt ein neues Directory an.

`cd <dir>`  
wechselt Directory (mit Pfadangabe).

`rd <dir>`  
löscht ein Directory.

`dos: FORTH-Name DOS-Name "`  
DOS: erwartet nach seinem Aufruf zwei Namen: Zuerst den FORTH-Namen, den der Befehl haben soll, dann den Namen der DOS-Funktion, die ausgeführt werden soll. Diese wird von einem Leerzeichen und einem quote " abgeschlossen. Ein Beispiel ist:  
`dos: ftype type "`  
Danach wird beim Aufruf des FORTH-Wortes `FTYPE` der MSDOS-Befehl `TYPE` ausgeführt.

### 7.3 Block-orientierte Dateien

Damit volksFORTH effizient auf den Massenspeicher zugreifen kann, bietet es - wie die meisten anderen FORTH-Systeme auch - einen besonderen Puffermechanismus für den Diskzugriff. Dadurch wird der Geschwindigkeitsunterschied zwischen den langsamen Laufwerken und dem schnellen Hauptspeicher ausgeglichen. Entgegen der Literatur [Brodie] ist im volkFORTH das Ende eines Screens nicht durch ein Doppel-Nullbyte `00` gekennzeichnet. volksFORTH arbeitet statt dessen mit `SKIP` und `SCAN` und wertet die Länge aus, die in `>in` bereits abgearbeitet ist.

- view** ( <name> -- )  
wird in der Form `view <name>` benutzt.  
Wenn <name> im Wörterbuch gefunden wird, so wird das File geöffnet, in dem der Quelltext von <name> steht und der Block wird gelistet, auf dem <name> definiert ist.  
Siehe: LIST
- list** ( blk# -- )  
zeigt den block mit der angegebenen Nummer an.
- files** ( -- )  
zeigt alle FORTH-Dateien an. Diese müssen nicht mit den MSDOS-Files identisch sein. Dagegen zeigt `dir` die MSDOS-Files an.
- isfile** ( -- addr )  
ist die Adresse einer Variablen, die auf das aktuelle FORTH-File zeigt. Der Wert in dieser Variablen entspricht typisch der Adresse des File-ControlBlocks FCB. Ist der Wert in dieser Adresse gleich Null, so wird direkt ohne ein File auf den Massenspeicher zugegriffen.
- isfile@** ( -- fcb )  
holt den FCB des `isfile`. Die Sequenz  
`isfile@ f.handle @`  
liefert die DOS-Handlenummer.
- fromfile** ( -- addr )  
ist die Adresse einer Variablen, die auf das FORTH-File zeigt, aus dem `copy` und `convey` die Blöcke lesen.  
`fromfile @` liefert den FCB dieser Datei.
- fswap** ( -- )  
vertauscht `isfile` und `fromfile` .  
Vergleiche im Editor das Drücken der Taste F9.
- [FCB]** ( -- fcb )  
ist eine Konstante des Typs FCB und dient für Vergleiche, ob ein Wort einen FCB darstellt. Es ist definiert als:  
`' kernel.scr @`
- .file** ( fcb -- )                   DOS  
druckt den FORTH-Filenamen des Files aus, dessen FCB-Adresse auf dem Stack liegt.

**filename** ( -- addr ) DOS  
ist die Anfangsadresse eines 62-Byte großen Speicherbereichs, der zum Ablegen von Filenamen während DOS-File-Operationen dient.

**fnamelen** ( -- n ) DOS  
ist eine Konstante, die die maximale Länge von logischen Filenamen, bestehend aus Drive, Path und Name, die in den FCB's abgespeichert werden können, bestimmt. Wird dieser Wert verändert, so kann die neue Länge erst in den FCB's verwendet werden, die nach der Änderung angelegt werden.

**A: B: C: D: E: F: G: H: ( -- )**  
Entspricht MS-DOS, macht das dadurch bezeichnete logische Laufwerk zum aktuellen Laufwerk.

**DTA** ( -- \$80 )  
ist als Konstante die Start-Adresse der DiskTransferArea.

**capacities** ( -- addr )  
ist die Adresse eines Vektors, der die Kapazitäten der angeschlossenen logischen Laufwerken in 1kByte-Blöcken enthält. Dafür sind maximal 6 Einträge (siehe: #DRIVES ) vorgesehen. Mit dem Hilfsprogramm DISKS.CFG können die Kapazitäten für die Diskettenlaufwerke eingestellt werden.

**drv** ( -- #drive )  
übergibt die Nummer des aktuellen Laufwerks.

**drive** ( #drive -- )  
macht das Laufwerk mit der angegebenen Nummer zum aktuellen Laufwerk. Hierbei entspricht n=0 dem Laufwerk A, n=1 dem Laufwerk B usw. . Auf dem aktuellen Laufwerk werden Files und Directories erzeugt (und nach Durchsuchen von PATH gesucht).  
Ø block liefert die Adresse des ersten Blocks auf dem aktuellen Laufwerk.  
Vergleiche >drive und offset .

**file <name>**  
erzeugt ein FORTH-Wort mit dem gleichen Namen. Wird dieses Wort später aufgerufen, so vermerkt es sich als aktuelles File isfile und als Hintergrund-Datei fromfile . Diesem logischen FORTH-File kann man dann mit assign oder make eine MSDOS-Datei zuordnen.



make <name>

wird in der Form

make <Dateiname>

benutzt. Es erzeugt ein MSDOS-File im aktuellen Directory und ordnet es dem aktuellen FORTH-File zu. Es hat die Länge Null (siehe more ).

Beispiel: file test.scr test.scr make test.scr

makefile <name>

erzeugt eine FORTH-Datei mit diesem Namen und ebenfalls eine MSDOS-Datei mit dem gleichen Namen. Dieses File wird auch sofort geöffnet, hat aber noch die Länge 0, wenn keine Größe mit nn more angegeben wurde. makefile ist ein Ersatz für die Prozedur in dem Beispiel, das bei make angegeben wurde.

assign

( -- )

wird benutzt in der Form

assign <filename>

und weist dem aktuellen logischen FORTH-File eine physikalische DOS-Datei zu.

file?

( -- )

zeigt den FORTH-Filenamen des aktuellen Files an.

?file

( -- )

zeigt den MSDOS-Namen, das Handle sowie Datum des letzten Zugriffs der aktuellen Datei.

load

( blk# -- )

lädt/compiliert den block mit der angegebenen Nummer an.

blk

( -- addr )

ist die Adresse einer Variablen, die die Blocknummer des gerade interpretierten Blocks enthält. Wird TIB interpretiert, enthält BLK den Wert NULL ; d.h., auch wenn Sie den Inhalt interaktiv auslesen, so erhalten Sie immer den Wert 0 !

loadfrom ( blk# -- )  
wird in folgender Form benutzt:  
( block# ) loadfrom <Dateiname>  
Dies ist wichtig, wenn während des Compilierens Teile eines anderen  
Files geladen werden sollen. Damit kann die Funktion eines Linkers imi-  
tiert werden.  
Beispiel: 15 loadfrom tools.scr  
Dieses Wort benutzt use, um Files zu selektieren; das bedeutet, daß in  
diesem Beispiel automatisch eine Datei namens tools.scr erzeugt wird,  
falls dieses File noch nicht existierte.  
Dieses Wort hat nichts mit from oder fromfile zu tun, obwohl es  
ähnlich heißt.

include ( -- )  
wird in der Form benutzt:  
include <Dateiname>  
Hier wird ein File vollständig geladen. Voraussetzung ist, daß in Screen  
#1 Anweisungen stehen, die zum Laden aller Screens führen.

b/blk ( -- &1024 ) "bytes pro block"  
ist die Länge eines Blocks (bzw. Screens, immer 1 KByte) wird auf den  
Stack gelegt.  
Siehe B/BUF .

b/buf ( -- n ) "bytes pro buffer"  
n ist die Länge eines Blockpuffers incl. der Verwaltungs-informationen  
des Systems, in der Version 3.81 für den PC \$408 Byte, bestehend aus  
\$400 Byte Puffer und \$8 Byte Verwaltungsinformation.

blk/drv ( -- n ) "blocks pro drive"  
n ist die Anzahl der auf dem aktuellen Laufwerk verfügbaren Blöcke.

(more ( n -- )  
wie MORE , jedoch wird das File nicht geschlossen.

more ( n -- )  
vergrößert die aktuelle Datei (isfile) um eine bestimmte Anzahl von  
Blöcken, die hinter angehängt werden. Anschließend wird die Datei ge-  
schlossen.

capacity

( -- n )

n ist die Speicherkapazität einer Datei, wobei Block 0 mitgezählt wird. Deshalb ist für Quelltexte nur capacity 1- (d.h. minus Block 0) nutzbar. Wenn shadow screens verwendet werden, steht nur capacity 2/ 1- zur Verfügung.

block

( u -- addr )

83

addr ist die Adresse des ersten Bytes des Blocks u in dessen Blockpuffer. Der Block u stammt aus dem File in ISFILE .

BLOCK prüft den Pufferbereich auf die Existenz des Blocks Nummer u. Befindet sich der Block u in keinem der Blockpuffer, so wird er vom Massenspeicher in einen an ihn vergebenen Blockpuffer geladen. Falls der Block in diesem Puffer als UPDATED markiert ist, wird er auf den Massenspeicher gesichert, bevor der Blockpuffer an den Block u vergeben wird.

Nur die Daten im letzten Puffer, der über BLOCK oder BUFFER angesprochen wurde, sind sicher zugreifbar. Alle anderen Blockpuffer dürfen nicht mehr als gültig angenommen werden (möglicherweise existiert nur 1 Blockpuffer).

Vorsicht ist bei Benutzung des Multitaskers geboten, da eine andere Task block oder buffer ausführen kann. Der Inhalt eines Blockpuffers wird nur auf den Massenspeicher gesichert, wenn der Block mit update als verändert gekennzeichnet wurde.

(block

( blk file -- addr )

liest den Block blk aus dem File, dessen FCB bei der Adresse file beginnt und legt diesen in einen Puffer bei der Adresse addr ab.

buffer

( block# -- addr )

weist dem block mit der angegebenen Nummer einen Blockpuffer im Speicher zu. Die Zuweisung wird von der internen Logik vorgenommen - anschließend wird die Startadresse dieses Pufferbereiches übergeben. Dabei entspricht addr der Adresse des ersten Bytes des Blocks in seinem Puffer. Der große Unterschied zwischen block und buffer ist, daß block tatsächlich die Daten von Disk in den Puffer liest. buffer dagegen reserviert nur den Puffer, initialisiert ihn nicht und liest keine Daten vom Massenspeicher. Daher ist der Inhalt dieses Blockpuffers undefiniert.

(buffer

( blk file -- addr )

reserviert einen 1kByte großen Puffer im Adreßbereich des FORTH-Systems für den Block blk. file ist die Adresse des FCB's, in dem sich der Block befindet. Ist file = 0, dann handelt es sich um einen DIRECTen physikalischen Zugriff. addr ist die Anfangsadresse des Puffers.

offset ( -- addr )  
liefert die Adresse einer UserVariablen, deren Inhalt zu der Blocknummer addiert wird, die sich beim Aufruf von block , buffer usw. auf dem Stack befindet.

fblock@ ( addr blk fcb -- ) DOS  
1024 Bytes, die im File fcb in Block blk stehen, werden ab der Adresse addr im FORTH-Adressbereich abgelegt.

fblock! ( addr blk fcb -- ) DOS  
1024 Bytes, die ab der Adresse addr innerhalb des FORTH-Adressbereichs stehen, werden auf den Block blk innerhalb des Files geschrieben, das durch fcb charakterisiert ist.

\*block ( blk -- d ) DOS  
Die doppelgenaue Zahl d ist die Byteadresse des ersten Bytes im 1024-Byte großen Block blk.

/block ( d -- rest blk ) DOS  
Die doppelgenaue Zahl D wird umgerechnet in die REST-Anzahl von Bytes innerhalb des 1024-Byte großen Blocks blk.

r/w ( addr blk fcb r/w -- \*f )  
ist ein deferred Wort, bei dessen Aufruf das systemabhängige Wort für den blockorientierten Massenspeicherzugriff ausgeführt wird. Dabei ist addr die Anfangsadresse des Speicherbereiches für den Block block, fcb die Adresse des Files, in dem sich der Block befindet. r/w ist gleich Null, wenn vom Speicherbereich auf den Massenspeicher geschrieben werden soll oder r/w =1, wenn der Block gelesen werden soll. \*f ist nur im Fehlerfall wahr, sonst falsch.

(r/w ( addr blk fcb r/w -- \*f )  
Die Standardroutine für das deferred Wort R/W.

eof ( -- true )  
entspricht der Konstanten TRUE = -1 .

## 7.4 Verwaltung der Block-Puffer

- core?** ( blk# filename -- addr ! ff )  
 prüft, ob sich der Block mit der angegebenen Nummer aus der genannten Datei bereits in einem der Block-Puffer befindet. Wenn das der Fall ist, wird die Anfangsadresse des Puffers übergeben, anderenfalls ein false flag.  
 Vergleiche **BLOCK**, **BUFFER** und **ISFILE**.
- (core?** ( blk# filename -- addr ! false )  
 ist die Standardroutine für das Wort **CORE?**.
- update** ( -- )  
 markiert den zur Zeit gültigen Pufferspeicher als verändert. Von dieser update-Kennzeichnung wird dann die Sicherung der Daten abhängig gemacht, wenn dieser Blockpuffer für einen anderen Block benötigt oder wenn **SAVE-BUFFERS** ausgeführt wird.  
 Vergleiche **PREV**.
- save-buffers** ( -- )  
 Das Alias **sav** im Editor heißt normalerweise **save-buffers** und rettet alle als **UPDATED** markierten Pufferbereiche auf Disk und löscht die **UPDATE**-Kennzeichnungen. Hierbei bleibt die bestehende Zuweisung **blk -> buffer** erhalten, die Blöcke werden jedoch nicht verändert und bleiben für weitere Zugriffe im Speicher erhalten.
- flush** ( -- )  
 schreibt alle als **UPDATED** markierten Pufferbereiche auf Disk und löscht die **UPDATE**-Markierung. Allerdings wird jetzt die Zuweisung **BLK -> BUFFER** zerstört. Zugleich wird die Datei geschlossen und das Inhaltsverzeichnis aktualisiert.  
 Vergleiche **SAVE-BUFFERS** und **EMPTY-BUFFERS**.
- emptybuf** ( buffer.addr -- )
- empty-buffers** ( -- )  
 löscht den Inhalt aller Blockpuffer, ohne die Daten von als **UPDATED** markierten Blockpuffern auf den Massenspeicher zurückzuschreiben. Es zerstört die Zuweisung **BLK -> BUFFER** und löscht die **UPDATE**-Markierung.

- freebuffer** ( -- )  
Entfernt den Blockpuffer mit der niedrigsten Adresse aus der Liste der Blockpuffer und gibt den dadurch belegten Speicherbereich frei. **FIRST** wird entsprechend um **B/BUF** erhöht. Ist der Inhalt des Puffers als **UPDATED** markiert, so wird er zuvor auf den Massenspeicher gesichert. Gibt es im System nur noch einen Blockpuffer, so geschieht nichts. Vergleiche **ALLOTBUFFER** .
- allotbuffer** ( -- )  
Fügt der Liste der Blockpuffer noch einen weiteren hinzu, falls oberhalb vom Ende des Returnstacks dafür noch Platz ist. **FIRST** wird entsprechend geändert. Vergleiche **FREEBUFFER** und **ALL-BUFFERS** .
- all-buffers** ( -- )  
Belegt den gesamten Speicherbereich von **LIMIT** abwärts bis zum oberen Ende des Returnstacks mit Blockpuffern. Siehe **ALLOTBUFFER** .
- first** ( -- addr )  
ist eine Variable, die einen Zeiger auf den Blockpuffer mit der niedrigsten Adresse darstellt. So liefert Ihnen **FIRST @** die Startadresse des Blockpufferbereiches. Vergleiche **ALLOTBUFFER** .
- limit** ( -- addr )  
ist im Gegensatz zu **FIRST** keine Variable, sondern liefert direkt die **addr**, unterhalb der sich die Blockpuffer befinden. Das letzte Byte des obersten Blockpuffers befindet sich in **addr-1**. Vergleiche **ALL-BUFFERS** und **ALLOTBUFFER**.
- Die Anzahl der Blockpuffer im aktuellen System läßt sich so ausrechnen:  

$$: \#buf \ ( \text{-- Anzahl} )$$

$$\quad \text{limit first @} - \text{ b/buf } / ;$$
- prev** ( -- addr )  
**addr** ist die Adresse einer Variablen, deren Wert der Anfang der Liste aller Blockpuffer ist. Der erste Blockpuffer in der Liste ist der zuletzt durch **BLOCK** oder **BUFFER** vergebene.
- offset** ( -- addr )  
**addr** ist die Adresse einer Uservariablen, deren Inhalt zu der Blocknummer addiert wird, die sich bei Aufruf von **BLOCK** , **BUFFER** usw. auf dem Stack befindet. **OFFSET** wird durch **DRIVE** verändert.



`.status` ( -- )

ist ein defered Wort, das vor dem Laden eines Blockes oder vor dem Abarbeiten der Tastatureingabe ausgeführt wird. Normalerweise ist es mit NOOP vorbesetzt.

### 7.5 Index-, Verschiebe- und Kopierfunktionen für Block-Files

Die meisten FORTH-Systeme, die bevorzugt mit Block-Dateien arbeiten, verfügen über eine INDEX-Funktion, die die Zeile 0 eines jeden Screens anzeigt. Diese Zeile gibt üblicherweise den Inhalt des Screens an.

Allgemein ist die Syntax für Index: `<start> <end> index`

Das nachfolgend beschriebene INDEX erwartet dagegen nur die Angabe des Blockes, bei dem die INDEX-Anzeige beginnen soll ; wird kein Argument übergeben, so beginnt die INDEX-Anzeige mit Screen 1. Auch können nach dem Kompilieren die durch ! als headerless gekennzeichneten Worte mit `clear` gelöscht werden.

```
! : range ( from to -- to+1 from )
  2dup u> IF swap THEN 1+ swap ;

! : .fname isfile@ [ DOS ] .file ;

: index ( from -- )
  depth 0= IF 1 THEN capacity 1- range
  ." von " .fname
  ( range) DO cr I 4 .r space
    I block c/l -trailing type
    stop? IF leave THEN
  LOOP ;
```

`copy` ( u1 u2 -- )

Der Block u1 wird in den Block u2 kopiert.

Innerhalb einer Datei wird ein bereits beschriebener Zielblock überschrieben, der alte Inhalt des Blocks u2 ist verloren.

Vergleiche auch `CONVEY`.

Ist ein `FROMFILE` angemeldet, so arbeitet `COPY` immer in Bezug auf das `FROMFILE`. Der Befehl `3 4 copy` kopiert dann den Block# 3 aus der Hintergrunddatei `FROMFILE` in die aktuelle Datei `ISFILE` und nicht innerhalb des `ISFILE` !

from <name>

gibt an, aus welchem File bei Datei-Operationen herauskopiert werden soll. Vergleiche auch die Datei STREAM.SCR .

```
from abc.scr 1 10 copy
```

kopiert den Block 1 von File FROMFILE in den Block# 10 ins aktuelle ISFILE , wobei der Zielblock leer sein muß; soll nicht überschrieben, sondern eingefügt werden, wird

```
from abc.scr 1 10 1 convey
```

eingesetzt.

convey

```
( 1st.block last.block to.block -- )
```

Verschiebt die Blöcke von 1st.block bis einschließlich last.block nach to.block. Die Bereiche dürfen sich überlappen. Eine Fehlerbehandlung wird eingeleitet, wenn last.block kleiner als 1st.block ist.

Die Blöcke werden aus dem File in FROMFILE ausgelesen und in das File in ISFILE geschrieben. FROMFILE und ISFILE dürfen gleich sein. Das Beispiel

```
4 6 5 convey
```

kopiert die Blöcke 4,5,6 nach 5,6,7. Der alte Inhalt des Blockes 7 ist verloren, der Inhalt der Blöcke 4 und 5 ist gleich.

CONVEY wird auch benutzt, wenn man gerne einen freien Block in ein Blockfile einfügen möchte:

(INSERT

```
( start# -- )
```

fügt an der angegebenen Zielposition einen Block ein.

```
: (insert ( start# -- )
```

```
dup 1+ capacity 1- swap 1 more convey ;
```

INSERT

```
( -- )
```

fügt vor dem aktuellen Block einen Block ein.

```
: insert ( -- )
```

```
scr @ dup (insert
```

```
block b/blk blank update ;
```

## 7.6 FCB-orientierte Dateien

pushfile

```
( -- )
```

C

wird in :-Definitionen benutzt, um den aktuellen Zustand der Filevariablen ISFILE und FROMFILE nach dem Ende der Colon-Definition wiederherzustellen.

Vergleichen Sie bitte den Mechanismus der lokalen Variablen: PUSH

- open ( -- )  
öffnet das aktuelle File zur Bearbeitung; ist aber in den meisten Fällen überflüssig, weil Files beim Zugriff automatisch geöffnet werden.
- emptyfile ( -- )  
legt eine leere Datei zum aktuellen Dateinamen an. Wird z.B in **MAKE** benutzt.
- killfile ( -- )  
löscht ohne weitere Rückfrage des aktuelle MSDOS-File; das aktuelle FORTH-File wird dabei nicht gelöscht, denn das FORTH-File ist wie eine Dateivariablen in anderen Sprachen zu betrachten und dementsprechend als Variable mit `forget <filename>` zu löschen.
- close ( -- )  
schließt das aktuelle File und aktualisiert das Inhaltsverzeichnis des Massenspeichers.
- flush ( -- )  
schließt alle geöffneten Dateien und aktualisiert das Inhaltsverzeichnis des Massenspeichers. Zugleich werden alle Zwischenpuffer auf die Disk zurückgeschrieben.
- fopen ( fcb -- )  
ist im volks4TH nicht vorhanden, weil die erwartete Funktion von **FRESET** erfüllt wird. Deshalb kann man dies so definieren:  
`' FRESET Alias FOPEN`
- freset ( fcb -- ) DOS  
fcb ist die Adresse eines FileControlBlocks. Das dadurch charakterisierte File wird "zurückgesetzt", d.h. das File wird geöffnet (wenn es noch nicht geöffnet war) und der Schreib/Lesezeiger wird auf den Anfang des Files gesetzt.
- fclose ( fcb -- ) DOS  
Das File, dessen FCB-Adresse auf dem Stack liegt, wird geschlossen.
- fgetc ( fcb -- 8b | eof ) DOS  
Aus dem File, dessen FCB-Adresse auf dem Stack liegt, wird das nächste Byte gelesen und der Schreib/Lesezeiger um eine Position weitergerückt. Wenn das letzte Byte bereits gelesen war, wird die End-Of-File-Markierung -1 zurückgegeben.

- fputc** ( 8b fcb -- ) DOS  
Das Byte 8B wird an der aktuellen Position des Schreib/Lesezeigers in das File FCB geschrieben. Dabei wird der Zeiger um eine Position weitergerückt.
- file@** ( dfaddr fcb -- 8b ! eof ) DOS  
Das Byte an der 32-bit Position dfaddr im File, daß durch fcb charakterisiert ist, wird gelesen. Liegt dfaddr jenseits des letzten Bytes im File, so wird -1 zurückgegeben. Nach erfolgreichem Lesen steht der Lese-/Schreibzeiger hinter dem gelesenen Byte.
- file!** ( 8b dfaddr fcb -- ) DOS  
Das Byte 8B wird an die Position dfaddr des Files fcb geschrieben.
- fseek** ( dfaddr fcb -- ) DOS  
Der Schreib/Lesezeiger des Files, das durch fcb charakterisiert ist, wird auf die Position dfaddr gesetzt. Dabei ist dfaddr eine doppelgenaue Zahl, so daß maximal Files von 4-GByte Größe verwaltet werden können.
- savefile** ( addr len -- )  
wird in der Form:  
savefile <name>  
benutzt und schreibt die Anzahl von len Bytes ab der Adresse addr in das neu erzeugte File mit dem Namen <name>.
- lfsave** ( seg:addr quan string -- )  
erzeugt ein File mit dem Namen, der als gecounteter String an der Adresse string abgelegt ist und schreibt die Anzahl quan Bytes ab der erweiterten Adresse seg:addr in dieses neue File.
- lfputs** ( seg:addr quan fcb -- )  
quan Bytes ab der erweiterten Adresse seg:addr werden ab der aktuellen Position des Schreib/Lesezeigers in das File geschrieben, das durch fcb charakterisiert ist. Danach steht der Schreib/Lesezeiger hinter dem letzten geschriebenen Byte. Um z.B. den Bildschirmspeicher in seinem aktuellen Zustand in eine Datei zu schreiben, wird es in der Form benutzt:  
... open video@ 0 c/ds isfile@ lfputs close ...
- lfgets** ( seg:addr quan fcb -- #read ) DOS  
siehe: ~READ . Lediglich wird statt der Handlungnummer die Adresse des FCB's des gewünschten Files angegeben. Entsprechend dem Beispiel bei LFPUTS lassen sich mit diesem Wort Bildschirmmasken direkt in den Bildschirmspeicher laden.

`asciz` ( `-- asciz` )  
holt das nächste Wort im Quelltext in den Speicher und legt es als null-terminierten String bei der Adresse `asciz` ab.

`>asciz` ( `string addr -- asciz` )  
Mit diesem Operator wird der gecountete String an der Adresse `string` umgewandelt in einen nullterminierten String, der an der Adresse `addr` abgelegt wird. `asciz` ist die Adresse, an der der neue String liegt.

`counted` ( `asciz -- addr len` )  
wird benutzt, um die Länge eines mit einer Null terminierten Strings zu bestimmen. `asciz` ist die Anfangsadresse dieses Strings (MS-DOS verwaltet Strings so), `addr` und `len` sind die Stringparameter, die z.B. von `TYPE` verarbeitet werden würden.

`loadfile` ( `-- addr` )  
ist eine Variable als Pointer auf das File, das gerade geladen wird.

`file-link` ( `-- addr` )  
ist eine Variable, die den Anfang zur Verwaltung einer Liste der File-Control-Blöcke (FCB) enthält. Der Inhalt von `FILE-LINK` zeigt auf den Anfang des Parameterfeldes des zuletzt definierten FCB's - und an dieser Stelle steht dann die Adresse des davor definierten FCB's usw., so daß dadurch alle FCB's aufgefunden werden können. Diese Liste wird u.a. von `forget` und `files` benutzt.

#### 7.7 HANDLE-orientierte Dateien

`~creat` ( `asciz attribut -- handle ff ! err#` ) DOS  
Der MS-DOS Systemaufruf, um ein neues File zu erzeugen.

`~open` ( `asciz mode -- handle ff ! err#` ) DOS  
Der MS-DOS Systemaufruf für das Öffnen eines Files. `asciz` ist die Adresse des vollen Namensstrings und `mode` bezeichnet die Art des File-Attributes. Dabei sind mögliche Attribute :

- 0 Constant read-only
- 1 Constant write-only
- 2 Constant read-write

Bei Erfolg liegt eine HANDLE-Nummer unter einer Null auf dem Stack, ansonsten eine Fehlernummer.

- `~read` ( seg:addr quan handle -- #read ) DOS  
quan Bytes werden aus dem File gelesen, daß durch die Kennnummer handle charakterisiert ist. Sie werden im erweiterten Speicherbereich bei seg:addr abgelegt. Nach Ende der Leseoperation liegt die Anzahl der Bytes auf dem Stack, die tatsächlich bis zum Ende des Files gelesen werden konnten. Es können jedoch nur maximal 64kByte auf einmal gelesen werden.
- `~close` ( handle -- ) DOS  
Der MS-DOS Systemaufruf, um das File, das durch handle characterisiert ist, zu schließen.
- `~unlink` ( asciz -- err# ) DOS  
Der MS-DOS Systemaufruf, um einen Fileeintrag zu löschen.
- `~dir` ( addr drive -- err# ) DOS  
Der MS-DOS Systemaufruf, mit dem das aktuelle Directory an der Adresse addr als nullterminierter String abgelegt wird.
- `~select` ( n -- ) DOS  
Der MS-DOS Systemaufruf, mit dem das aktuelle Laufwerk selektiert wird.
- `~disk?` ( -- n ) DOS  
Der MS-DOS Systemaufruf, mit dem das aktuelle Laufwerk abgefragt wird.
- `attribut` ( -- addr ) DOS  
Eine Variable, die die File-Attribute enthält, die bei der Suche nach Files in einem Directory berücksichtigt werden. Standardmäßig mit 7 initialisiert, so daß in die Suche read-only, hidden und system -Files eingeschlossen sind.
- `(fsearch` ( string -- asciz \*f ) DOS  
Das File, dessen Name als String ab Adresse string steht, wird in der Directory gesucht. Enthält der Filename keine Suchpfadinformation, dann wird im aktuellen Directory gesucht. Bei Erfolg liegt eine Null auf dem Stack, sonst eine Fehlernummer.
- `fsearch` ( string -- asciz \*f ) DOS  
Ein deferred Wort. Es enthält die Suchstrategie (siehe: (FSEARCH ), die beim Öffnen eines Files verwendet wird, um das File auf der Disk zu lokalisieren.



`~first` ( `asciz attr -- err#` ) DOS  
 Der MS-DOS Systemaufruf, um erstmalig nach einem File zu suchen.

`~next` ( `-- err#` ) DOS  
 Der MS-DOS Systemaufruf, der nach `~FIRST` benutzt wird, um weitere passende Filenamen aufzufinden.

### 7.8 Direkt-Zugriff auf Disketten

`direct` ( `--` )  
 Die Filevariablen werden auf Null gesetzt und damit beziehen sich die Diskzugriffe durch `BLOCK` auf physikalische Blocks.

`#drives` ( `-- n` )  
 ist eine Konstante, die die mögliche Anzahl von logischen Laufwerken im System definiert. Diese Anzahl ist nur im `DIRECT`-Modus von Bedeutung. So, wie der Kern compiliert ist, sind maximal 6 Laufwerke zugelassen.

`/drive` ( `blk1 -- blk2 drive` )  
 für den `DIRECT`-Modus beim Diskzugriff. Aus der absoluten Blocknummer `blk1` wird (siehe: `CAPACITIES` ) die relative Blocknummer `blk2` auf Laufwerk `DRIVE` berechnet. Dabei entspricht Laufwerk A: dem Drive 0 etc. .

`>drive` ( `blk1 #drv -- blk2` ) "to-drive"  
 dient zum "Umrechnen" von Blocknummern im `DIRECT`-Modus.  
`blk2` ist die absolute Blocknummer, die dem relativen Block `blk1` auf Drive `#drv` entspricht. Beispiel  
`23 1 >drive block`  
 holt den Block mit der Nummer 21 vom Laufwerk 1, egal welches Laufwerk gerade das aktuelle ist.

`capacity` ( `-- n` )  
 gibt die Kapazität in 1024-Byte Blöcken des aktuellen Files bzw. des aktuellen Laufwerks bei `DIRECT`-Zugriff.

### 7.9 Fehlerbehandlung

`?diskerror` ( `--` )  
 Ein deferred Wort, daß die Fehlerbehandlungsroutine für Disk- und Filezugriffe enthält. Standardmäßig ist die Routine `(DISKERROR)` zugewiesen.

error# ( -- addr )  
Eine Variable, die die Fehlernummer des letzten Fehlers beim Zugriff auf ein File enthält.

(diskerror ( #err -- ) DOS  
Die Standard-System Fehlerbehandlungsroutine für Fehler beim Diskzugriff. Hiermit ist das deferred Wort ?DISKERROR initialisiert.

## 8. Speicheroperationen

Die INTEL-Prozessoren haben eine verkomplizierte Art, den Adreßraum jenseits von 64kBytes zu adressieren - nämlich mit sogenannten "Segmentregistern". Am besten kommt man damit noch zurecht, wenn man diese Prozessoren als 16-bit Prozessoren betrachtet, die in der Lage sind, mehrere Programme, die jeweils höchstens 64k Programmspeicherbereich haben, gleichzeitig im Speicher zu halten. Es ist deshalb auch unvernünftig, auf diesen Prozessoren ein FORTH-System mit 32-bit Adressen zu installieren - es handelt sich eben nicht um 32-bit Prozessoren.

Um in volksFORTH den gesamten 1 MB-Adreßraum zu nutzen, ist die Möglichkeit gegeben, aus dem FORTH heraus mit dem Wort `call` (im DOS-Vokabular) ein weiteres .COM- oder .EXE-Programm aufzurufen. Dies können natürlich ihrerseits FORTH-Programme sein, denen dann auch noch eine ganze Eingabezeile als Parameter mit "auf den Weg" gegeben werden kann.

Damit wäre es zum Beispiel möglich, den Full-Screen Editor aus dem System auszulagern und mit den Befehlen `FIX`, `EDIT`, `ED` usw. jeweils ein .COM-Programm aufzurufen, das den FORTH-Editor als "stand-alone" Programm enthält und damit keinen Adreßraum im Entwicklungssystem mehr verbraucht.

Über die Systemvariable `RETURN_CODE` ist es auch noch möglich, einen Fehlercode bei Beendigung des FORTH-Programms an MS-DOS zu übergeben, der dann in Batch-Files getestet werden kann.

Die Intel-Prozessoren setzen die Speicheradressen aus zwei Teilen zusammen, dem SEGMENT und dem OFFSET. Dies ist jedoch nicht mit "echten" 32Bit-Adressen zu verwechseln. Diese werden auf einem 16Bit-Stack in der Reihenfolge "low-word" unter dem "high-word" abgelegt. Überträgt man diese Philosophie auf die "seg:addr"-Adressen des 8086, dann blockiert dauernd die Segmentadresse den Stack.

Deshalb wird bei den Operatoren, die im erweiterten Adreßraum des 8086 operieren, die Segmentadresse UNTER der Offsetadresse auf den Stack gelegt. Der Stackkommentar dafür lautet "seg:addr". Den Operatoren, die als Adreßargument eine "erweiterte" Adresse benötigen, wird ein "1" im Namen vorangestellt.

### 8.1 Speicheroperationen im 16-Bit-Adressraum

```
@      ( addr -- 16b )      83      "fetch"
      Von der Adresse addr wird der Wert 16b aus dem Speicher geholt. Siehe
      auch ! .
      volksFORTH enthält nicht das Wort ? . Benutzen Sie:
      : ? ( addr -- ) @ . ;
```

! ( 16b addr -- ) 83 "store"  
 16b werden in den Speicher auf die Adresse addr geschrieben. In 8Bit-weise adressierten Speichern werden die zwei Bytes addr und addr+1 überschrieben.

+! ( w1 addr -- ) 83 "plus-store"  
 w1 wird zu dem Wert w in der Adresse addr addiert. Benutzt die plus-Operation. Die Summe wird in den Speicher in die Adresse addr geschrieben. Der alte Speicherinhalt wird überschrieben.

2! ( 32b addr -- ) 83 "two-store"  
 32b werden in den Speicher ab Adresse addr geschrieben.

2@ ( addr -- 32b ) 83 "two-fetch"  
 Von der Adresse addr wird der Wert 32b aus dem Speicher geholt.

c! ( 16b addr -- ) 83 "c-store"  
 Von 16b werden die niederwertigsten 8 Bit in den Speicher an der Adresse addr geschrieben.

c@ ( addr -- 8b ) 83 "c-fetch"  
 Von der Adresse addr wird der Wert 8b aus dem Speicher geholt.

move ( addr0 addr1 u -- )  
 Beginnend bei addr0 werden u Bytes nach addr1 kopiert. Dabei ist es ohne Bedeutung, ob überlappende Speicherbereiche aufwärts oder abwärts kopiert werden, weil MOVE die passende Routine dazu auswählt. Hat u den Wert Null, passiert nichts.  
 Siehe auch CMOVE und CMOVE> .

cmove ( addr0 addr1 u -- ) 83 "c-move"  
 Beginnend bei addr0 werden u Bytes zur Adresse addr1 kopiert. Zuerst wird das Byte von addr0 nach addr1 bewegt und dann aufsteigend fortgefahren. Wenn u Null ist, wird nichts kopiert.

cmove> ( addr0 addr1 u -- ) 83 "c-move-up"  
 Beginnend bei adDr0 werden u Bytes zur Adresse adDr1 kopiert. Zuerst wird das Byte von addr0 +u -1 nach addr1 +u -1 kopiert und dann absteigend fortgefahren. Wenn u Null ist, wird nichts kopiert. Das Wort wird benutzt, um Speicherinhalte auf höhere Adressen zu verschieben, wenn die Speicherbereiche sich überlappen.

- fill** ( addr u 8b -- )  
 Von der Adresse addr an werden u Bytes des Speichers mit 8b überschrieben. Hat u den Wert Null, passiert nichts.
- erase** ( addr u -- )  
 Von der Adresse addr an werden u Bytes im Speicher mit \$00 überschrieben. Hat u den Wert Null, passiert nichts.
- blank** ( addr u -- )  
 Von der Adresse addr an werden u Bytes im Speicher mit Leerzeichen BL (\$20) überschrieben. Hat u den Wert Null, passiert nichts.
- flip** ( u1 -- u2 )  
 ist das Byteswap des obersten Stackelements.  
 u1 ist eine 16-bit Zahl mit den Bits B15..B8 und B7..B0, wobei man B15..B8 als das "high-Byte", B8..B0 als das "low-Byte" bezeichnet. Durch FLIP wird das High- mit dem Low-Byte ausgetauscht, so daß u2 als Ergebnis die Bits in der Reihenfolge B7..B0 und B15..B8 angeordnet hat. Dieses FLIP entspricht der Definition:  
 : cswap ( 16b -- 16b' ) \$100 um\* or ;
- ctoggle** ( 8b addr -- ) 83 "c-toggle"  
 Für jedes gesetzte Bit in 8b wird im Byte mit der Adresse addr das entsprechende Bit invertiert (d.h. ein zuvor gesetztes Bit ist danach gelöscht und ein gelöscht Bit ist danach gesetzt). Für alle gelöschten Bits in 8b bleiben die entsprechenden Bits im Byte mit der Adresse addr unverändert. Der Ausdruck dup c@ rot xor swap c! wirkt genauso.
- off** ( addr -- )  
 schreibt den Wert FALSE in den Speicher mit der Adresse addr.
- on** ( addr -- )  
 schreibt den Wert TRUE in den Speicher mit der Adresse addr.
- here** ( -- addr ) 83  
 addr ist die Adresse des nächsten freien Dictionaryplatzes.
- align** ( -- )  
 rundet normalerweise den Dictionary-Pointer und damit auch HERE auf die nächste geraden Adresse auf. Ist HERE gerade, so geschieht nichts. Im volks4th für den 8088-Prozessor hat ALIGN keine Wirkung.

- , ( 16b -- ) 83 "comma"  
 ist ein 2 ALLOT für die gegebenen 16b und speichert diese 16b an  
 HERE 2- ab.
- c, ( 8b -- ) "c-comma"  
 ist ein ALLOT für ein Byte und speichert 1 Byte in HERE 1- ab.
- allot ( w -- ) 83  
 allokiert w Bytes im Dictionary. Die Adresse des nächsten freien Dic-  
 tionaryplatzes wird entsprechend verstellt.
- dp ( -- addr ) "d-p"  
 ist eine Uservariable, in der die Adresse des nächsten freien Dictionary-  
 platzes steht.
- uallot ( n1 -- n2 )  
 allokiert bzw. deallokiert n1 Bytes in der Userarea. n2 gibt den Anfang  
 des allokierten Bereiches relativ zum Beginn der Userarea an. Eine  
 Fehlerbehandlung wird eingeleitet, wenn die Userarea voll ist.
- udp ( -- addr ) "u-d-p"  
 ist eine Uservariable, in dem das Ende der bisher allokierten Userarea  
 vermerkt ist.
- pad ( -- addr ) 83  
 addr ist die Startadresse einer "scratch area". In diesem Speicherbereich  
 können Daten für Zwischenrechnungen abgelegt werden. Wenn die nächste  
 verfügbare Stelle für das Dictionary verändert wird, ändert sich auch die  
 Startadresse von PAD . Die vorherige Startadresse von PAD geht  
 ebenso wie die Daten dort verloren.
- count ( addr1 -- addr2 8b ) 83  
 addr2 ist addr1+1 und 8b der Inhalt von addr1. Das Byte mit der  
 Adresse addr1 enthält die Länge des Strings angegeben in Bytes. Die  
 Zeichen des Strings beginnen bei addr1+1. Die Länge eines Strings darf  
 im Bereich (0..255) liegen.  
 Vergleiche "counted String" .
- place ( addr1 +n addr2 -- )  
 bewegt +n Bytes von der Adresse addr1 zur Adresse addr2+1 und  
 schreibt den Wert +n in die Speicherstelle mit der Adresse addr2. Wird in  
 der Regel benutzt, um Text einer bestimmten Länge als "counted string"  
 abzuspeichern. addr2 darf gleich, größer und auch kleiner als addr1 sein.



**dump** ( addr Anzahl -- )  
 Dieser wichtige Befehl zeigt ab einer gegebenen Adresse eine bestimmte Anzahl Bytes des Hauptspeichers an. Denn ist es sinnvoll, nachdem man im Speicher ganze Bereiche reserviert und dort Strukturen abgelegt hat, sich diese Bereiche auch anzusehen. Ebenso zeigt Ihnen der Befehl  
 ( n) block b/blk dump  
 einen Ihrer Quelltextblöcke an.

**b/seg** ( -- n )  
 ist eine Konstante, die angibt, wieviele Bytes zwischen zwei Segmenten liegen. Dies sind beim 8086 16 Bytes, beim 80286 im 286-Modus jedoch 64 Bytes. volksFORTH auf dem 80286 setzt zur Zeit voraus, daß der 8086-Emulationsmodus eingeschaltet ist.

**ds@** ( -- seg )  
 legt die Segmentadresse des Segments auf den Stack, in dem sich das maximal 64kByte große FORTH-System gerade befindet. Das Daten-, Extra-, Stack- und Codesegment werden durch FORTH alle auf den gleichen Wert gesetzt.

### 8.2 Segmentierte Speicheroperationen

**l@** ( seg:addr -- n )  
 entspricht dem @ , jedoch im erweiterten Adreßraum.

**l!** ( n seg:addr -- )  
 entspricht dem ! , jedoch im erweiterten Adreßraum.

**lc@** ( seg:addr -- 8b )  
 entspricht dem C@ , jedoch im erweiterten Adreßraum.

**lc!** ( 8b seg:addr -- )  
 entspricht dem C! , jedoch im erweiterten Adreßraum.

**lmove** ( from:seg:addr to:seg:addr quan -- )  
 entspricht dem MOVE , jedoch im erweiterten Adreßraum. Es können hiermit maximal 64KBytes auf einmal bewegt werden.

- ltype** ( seg:addr len -- )  
entspricht dem **TYPE** , jedoch im erweiterten Adreßraum. Es ist zu beachten, daß **TYPE** in den Videodisplaytreibern BIOS.VID und MULTI.VID so implementiert ist, daß bei Erreichen des Zeilenendes nicht automatisch ein **CR** ausgeführt wird. Statt dessen werden alle Zeichen, die "jenseits" des rechten Rands liegen, nicht ausgegeben.
- ldump** ( seg:addr quan -- ) **TOOLS**  
entspricht dem **DUMP** , jedoch im erweiterten Adreßraum und zeigt ab einer angegebenen Adresse eine bestimmte Anzahl Bytes an.
- lallocate** ( #pages -- seg ff | rest err# ) **EXTEND**  
Hiermit können im erweiterten Adreßraum die Anzahl #pages Speicherplatz angefordert werden.  
Die Größe einer "Page" in Bytes entspricht der Konstanten **B/SEG** . Wenn die Speicheranforderung erfüllt werden kann, dann wird unter einer **NULL** als Flag für den Erfolg der Operation die Segmentadresse des ersten Segments innerhalb eines zusammenhängenden Speicherbereichs von #page Pages auf den Stack gelegt.  
Ansonsten liegt unter einem Fehlercode die maximale Anzahl von Pages, die noch als zusammenhängender Bereich verfügbar sind. Diese Funktion ist in dem Wort **SAVEVIDEO** benutzt, um den Bildschirminhalt in den Speicher zu kopieren.  
Die komplementären Funktionen sind **LFREE** und **RESTOREVIDEO** .
- lfree** ( seg -- err# )  
Der Speicherbereich, der an der Segmentadresse seg beginnt, wird wieder an das Betriebssystem zurückgegeben.  
Diese Operation ist nur definiert, wenn zu einem vorherigen Zeitpunkt eine **LALLOCATE**-Operation durchgeführt worden war, die als Ergebnis die Segmentadresse seg gehabt hatte.  
Es wird ein Fehlercode auf dem Stack übergeben, der im Erfolgsfall 0 ist.

## 9. Datentypen

Allgemein bestehen Programme aus einem Programm-Datenteil und einem Anweisungsteil. Dabei enthält der Befehlsteil die zu Algorithmen angeordneten Befehle, die sich wiederum aus Operanden und Operatoren zusammensetzen. Der Datenteil dagegen beschreibt eines Programmes den statischen Datenbereich, der die zu bearbeitenden Daten enthält.

Diese Datenbereiche sind durch die Interpretation ihres Inhaltes strukturiert, wobei Datenbereiche mit gleicher Strukturierung zum gleichen Datentyp gehören. Denn um mit einer Reihe von Bytes arbeiten zu können, muß man wissen, ob diese Bytes eine Zeichenkette, den Inhalt einer Variablen oder die Listenzellen einer :-Definition darstellen.

Die Strukturierung und Verwaltung sowie die Interpretation des Dateninhaltes im Datenteil wird in den PASCAL-ähnlichen Sprachen am Programm-anfang durch die Deklarationen der Datentypen, in FORTH dagegen erst im Befehlsteil eines Programmes durch die eingesetzten Befehle bestimmt.

Da in FORTH die Typisierung über die Auswahl und den Einsatz von geeigneten Operatoren erfolgt, gehören in FORTH alle die Datenstrukturen dem gleichen Typ an, denen die Operatoren gemeinsam sind. In FORTH bestimmt also die jeweilige Operation den Datentyp des entsprechenden Bereiche - so interpretiert die Befehlsfolge `pad count type` die im Speicherbereich `PAD` abgelegten Daten als Zeichenkette !

### Glossar

- Create** ( -- ) 83  
 ist ein definierendes Wort, das in der Form:  
`Create <name>`  
 benutzt wird. `CREATE` erzeugt einen Kopf für `<name>`.  
 Die nächste freie Stelle im Dictionary (vergl. `HERE` und `DP`) ist nach einem `CREATE <name>` das erste Byte des Parameterfelds von `<name>`.  
 Wenn `<name>` ausgeführt wird, legt es die Adresse seines Parameterfelds auf den Stack. `CREATE` reserviert keinen Speicherplatz im Parameterfeld von `<name>`.
- Constant** ( 16b -- ) 83  
 ist ein definierendes Wort, das in der Form:  
`16b Constant <name>`  
 benutzt wird. Wird `<name>` später ausgeführt, so wird 16b auf den Stack gelegt.

- 2Constant** ( 32b -- ) 83  
 ist ein definierendes Wort, das in der Form:  
     32b 2Constant <name>  
 benutzt wird. Erzeugt einen Kopf für <name> und legt 32b in dessen Parameterfeld so ab, daß bei Ausführung von <name> 32b wieder auf den Stack gelegt wird.
- Variable** ( -- ) 83  
 ist ein definierendes Wort, benutzt in der Form:  
     Variable <name>  
**VARIABLE** erzeugt einen Kopf für <name> im Dictionary und hält 2 Byte in seinem Parameterfeld frei. Siehe **ALLOT** .  
 Dieses Parameterfeld wird für den Inhalt der Variablen benutzt, jedoch nicht initialisiert. Wird <name> ausgeführt, so wird die Adresse des Parameterfeldes von <name> auf den Stack gelegt. Mit **@** und **!** kann der Wert von <name> gelesen und geschrieben werden.  
 Siehe auch **+!** .
- 2Variable** ( -- ) 83  
 ist ein definierendes Wort, das in der Form:  
     2Variable <name>  
 benutzt wird. Es erzeugt einen Kopf für <name> und hält 4 Byte in seinem Parameterfeld frei, die den Inhalt der doppelt-genauen Variablen aufnehmen. **2VARIABLE** wird nicht initialisiert. Wenn <name> ausgeführt wird, wird die Adresse des Parameterfeldes auf den Stack gelegt.  
 Siehe **VARIABLE** .
- :** ( -- sys ) 83 "colon"  
 ist ebenfalls ein definierendes Wort, das in der Form:  
     : <name> <actions> ;  
 benutzt wird.  
 Es erzeugt die Wortdefinition für <name> im Kompilations-Vokabular, schaltet den Compiler an und kompiliert anschließend den Quelltext wird.

## Achtung:

Das erste Vokabular der Suchreihenfolge - das "transient" Vokabular - wird durch das Kompilations-Vokabular ersetzt! Das Kompilations-Vokabular wird nicht geändert. .

Soll also das CONTEXT-Vokabular für eine :-Definition geändert werden, so ist das gewünschte Vokabular entweder innerhalb einer Definition mit den eckigen Klammern

```
: <name> [ <vocabulary> ] ... ;
```

zum CONTEXT zu machen oder außerhalb einer Definition zweimal in die Suchreihenfolge aufzunehmen ( also ) und später zu löschen ( toss ) :

```
<vocabulary> also
```

```
: <colondefinition> ; toss
```

<name> wird als "colon-definition" oder ":-Definition" bezeichnet. Die neue Wortdefinition für <name> kann nicht im Dictionary gefunden werden, bis das zugehörige ; oder ;CODE erfolgreich ausgeführt wurde. RECURSIVE macht <name> jedoch sofort auffindbar.

Vergleiche HIDE und REVEAL .

Eine Fehlerbehandlung wird eingeleitet, wenn ein Wort während der Kompilation nicht gefunden bzw. nicht in eine Zahl (siehe auch BASE ) gewandelt werden kann.

Der auf dem Stack hinterlassene Wert sys dient der Kompiler-Sicherheit und wird durch ; bzw. ;CODE abgebaut.

```
; ( -- ) 83 I C "semi-colon"
( sys -- ) compiling
```

beendet die Kompilation einer :-Definition.

; macht den Namen dieser colon definition im Dictionary auffindbar, schaltet den Kompiler aus, den Interpreter ein und kompiliert ein UNNEST (siehe auch EXIT ). Der Stackparameter sys, der in der Regel von : hinterlassen wurde, wird geprüft und abgebaut. Eine Fehlerbehandlung wird eingeleitet, wenn sys nicht dem erwarteten Wert entspricht.

## Defer

( -- )

ist ein definierendes Wort, das in der Form:

```
Defer <name>
```

benutzt wird. Erzeugt den Kopf für ein neues Wort <name> im Dictionary, hält 2 Byte in dessen Parameterfeld frei und speichert dort zunächst die Kompilationsadresse einer Fehlerroutine. Wird <name> nun ausgeführt, so wird eine Fehlerbehandlung eingeleitet.

Man kann dem Wort <name> jedoch zu jeder Zeit eine andere Funktion zuweisen mit der Sequenz:

```
' <action> Is <name>
```

Nach dieser Zuweisung verhält sich <name> wie <action>.

Mit diesem Mechanismus kann man zwei Probleme elegant lösen:

1. Einerseits läßt sich <name> bereits kompilieren, bevor ihm eine sinnvolle Aktion zugewiesen wurde.
2. Andererseits ist die Veränderung des Verhaltens von <name> für spezielle Zwecke auch nachträglich möglich, ohne neu kompilieren zu müssen.

Deferred Worte im System sind R/W 'COLD 'RESTART 'ABORT 'QUIT NOTFOUND .STATUS und DISKERR . Diese Worte sind DEFERred, damit ihr Verhalten für die Anwendung geändert werden kann.

## Is

( cfa -- )

ist ein Wort, mit dem das Verhalten eines deferred Wortes verändert werden kann. IS wird in der Form:

```
' <action> Is <name>
```

benutzt. Wenn <name> kein deferred Wort ist, wird eine Fehlerbehandlung eingeleitet, sonst verhält sich <name> anschließend wie <action>.

Siehe DEFER .

## Alias

( cfa -- )

ist ein definierendes Wort, das typisch in der Form:

```
' <oldname> Alias <newname>
```

benutzt wird. ALIAS erzeugt einen Kopf für <newname> im Dictionary. Wird <newname> aufgerufen, so verhält es sich wie <oldname>. Insbesondere wird beim Kompilieren nicht <newname>, sondern <oldname> im Dictionary eingetragen.

Vocabulary ( -- ) 83

ist ein definierendes Wort, das in der Form:

Vocabulary <name>

benutzt wird. VOCABULARY erzeugt einen Kopf für <name>, das den Anfang einer neuen Liste von Worten bildet. Wird <name> ausgeführt, so werden bei der Suche im Dictionary zuerst die Worte in der Liste von <name> berücksichtigt. Wird das VOCABULARY <name> durch die Sequenz:

<name> definitions

zum Kompilations-Vokabular, so werden neue Wort-Definitionen in die Liste von <name> gehängt.

Vergleiche auch CONTEXT CURRENT ALSO TOSS ONLY FORTH und ONLYFORTH .

Input: ( -- ) "input-colon"

ein definierendes Wort, benutzt in der Form:

Input: <name>

newKEY newKEY? newDECODE newEXPECT ;

INPUT: erzeugt einen Kopf für <name> im Dictionary und kompiliert einen Satz von Zeigern auf Worte, die für die Eingabe von Zeichen zuständig sind. Wird <name> ausgeführt, so wird ein Zeiger auf das Parameterfeld von <name> in die Uservariable INPUT geschrieben. Alle Eingaben werden jetzt über die neuen Eingabeworte abgewickelt.

Die Reihenfolge der Worte nach INPUT: <name> bis zum semi-colon muß eingehalten werden: ..key ..key? ..decode ..expect .

Im System ist das mit INPUT: definierte Wort keyboard enthalten, nach dessen Ausführung alle Eingaben von der Tastatur geholt werden.

Siehe DECODE und EXPECT .

Output: ( -- ) "output-colon"  
 ist ein definierendes Wort, benutzt in der Form:  
 Output: <name>  
 newEMIT newCR newTYPE newDEL newPAGE newAT newAT? ;  
 OUTPUT: erzeugt einen Kopf für <name> im Dictionary und kompiliert  
 einen Satz von Zeigern auf Worte, die für die Ausgabe von Zeichen zu-  
 ständig sind. Wird <name> ausgeführt, so wird ein Zeiger auf das Para-  
 meterfeld von <name> in die Uservariable OUTPUT geschrieben. Alle  
 Ausgaben werden jetzt über die neuen Ausgabeworte abgewickelt.

Die Reihenfolge der Worte nach OUTPUT: <name> bis zum semi-colon muß  
 eingehalten werden: ..emit ..cr ..type ..del ..page ..at ..at? .

Im System ist das mit OUTPUT: definierte Wort display enthalten,  
 nach dessen Ausführung alle Ausgaben auf den Bildschirm geleitet wer-  
 den. Vergleiche auch das Wort PRINT aus dem Printer-Interface.

User ( -- ) 83  
 ist ein definierendes Wort, benutzt in der Form:  
 User <name>  
 USER erzeugt einen Kopf für <name> im Dictionary und hält 2 Byte in  
 der Userarea frei (siehe UALLOT ). Diese 2 Byte werden für den Inhalt  
 der Uservariablen benutzt und werden nicht initialisiert. Im Parameterfeld  
 der Uservariablen im Dictionary wird nur ein Offset zum Beginn der  
 Userarea abgelegt.

Wird <name> ausgeführt, so wird die Adresse des Wertes der Uservariab-  
 len in der Userarea auf den Stack gelegt.

Uservariablen werden statt normaler Variablen z.B. dann benutzt, wenn  
 der Einsatz des Multitaskers geplant ist und mindestens eine Task die  
 Variable unbeeinflußt von anderen Tasks benötigt. Jede Task hat ihre  
 eigene Userarea.

### 9.1 Ein- und zweidimensionale Felder

Wie in dem Buch "In FORTH denken" [S.192/193] diskutiert wird, stellen die meisten  
 FORTH-Systeme aus gutem Grund kein Definitionswort für Arrays zur Verfügung.  
 Weil aber der Aufbau dieser Definitionen nicht immer problemlos zu handhaben ist  
 und die notwendigen Algorithmen in der Literatur manchmal falsch dargestellt wer-  
 den, seien hier die Definitionen für eindimensionale Felder (Vektoren) und zwei-  
 dimensionale Arrays angegeben.

Diese Datenstrukturen verbergen der zugrunde liegende Zellgröße in eigenen Defini-  
 tionen, um die Worte ohne die typischen 2+ 2- 2\* schreiben zu können. Die



Basisgröße der Adressierung in einem 16bit-System ist das Byte, die Adressberechnungen arbeiten mit 16Bit-, also 2Byte-Adressen.

```

1 Constant byte
2 Constant integer
4 Constant double

| : bytes ;           \ bytes ist 1*
| : integers 2* ;    \ Größe eines Bereiches
| : doubles 4* ;

| : byte+ 1+ ;       \ nächste byte-Adresse
| : integer+ 2+ ;   \ nächste word-Adresse
| : double+ 4+ ;

```

Obwohl FORTH als Sprache keine Typisierung der Daten erwartet, arbeitet die hier vorgestellte FELD:-Definition mit einer Typübernahme:

```

: Feld:
  Create ( #elements type - )
    dup c, *
    here over erase
    allot
  Does) ( index <addr> - addr )
    count rot * + ;

```

Diese FELD:-Definition übergibt zur Laufzeit die Adresse des gegebenen Elementes, abhängig vom Typ der Variablen. Das Feld wird zur Compile-Zeit mit Nullen initialisiert.

In der möglichen Anwendung einer Meßdatenerfassung von Meßstellen wird die Handhabung von FELD: deutlich:

```

5 byte Feld: Bonn
5 integer Feld: Köln
5 double Feld: Moers

```

```

: .Werte cr
  5 0 DO I Köln @ 7 u.r LOOP cr;

```

Der Zugriff auf eine solche Datenstruktur kann z.B. über eine Schleife erfolgen, wobei - wie immer in FORTH - die zum Datentyp passenden Operatoren eingesetzt werden müssen.

Die Arbeit mit einer Matrix ist ähnlich, nur wird hier der Zugriff über zwei geschachtelte Schleifen erfolgen. Bitte beachten Sie, daß beim Zugriff FORTH-untypische Schleifenvariablen eingesetzt werden, die über dieSynonymdeklaration Alias definiert wurden.

Diese Definition von MATRIX: übernimmt ebenfalls eine Typangabe:

```

: Matrix: \ #zeilen #spalten range (double|integer|byte)
          \ #zeile #spalte -- element_addr )
  Create
    2dup swap \ integer|byte #zeile
    c, c, \ store #? and #?
    * * dup \ amount bytes amount bytes
    here swap erase \ from here amount preset to 0
    allot \ amount allot

```

```

Does>
  dup c@ 3 roll *
    2 roll +
    over byte+
  c@ * +
  integer+ ;

\ Drei Meßstationen haben je 4 Meßwerte erfaßt
\ und gespeichert

3 Constant #Stationen      ' I Alias Station#
4 Constant #Werte          ' J Alias Wert#

#Stationen #Werte byte Matrix: Messung

\ Wert|#Stat.|#Wert|Feldname|Operation
  3   0   0   Messung   c!
  6   0   1   Messung   c!
  9   0   2   Messung   c!
 12   0   3   Messung   c!

  5  1 0 Messung c!    7  2 0 Messung c!
 10  1 1 Messung c!    9  2 1 Messung c!
 15  1 2 Messung c!    9  2 2 Messung c!
 10  1 3 Messung c!    7  2 3 Messung c!

: .all ( -- )
#Stationen 0 DO cr
  #Werte 0 DO Wert# Station# Messung c@ 3 .r
  LOOP
LOOP ;

```

Den einzelnen Operationen zum Einspeichern der Werte steht das Auslesen der Werte über zwei geschachtelte Schleifen gegenüber.

Eine weitere, oft genutzte Datenstruktur ist ein Ringpuffer (queue), hier in der Form eines 255 Byte langen counted strings. In diesen Puffer werden Zeichen hinten angehängt und vorne ausgelesen, so daß sich eine FIFO-Struktur ergibt:

```

Create QUEUE 0 c, 255 allot

: more? ( addr -- n) c@ ;

: q@ ( addr -- char)
dup more? IF detract exit
  ELSE ." Ringpuffer leer! " drop
  THEN ;

: q! ( addr char --) append ;

: q. ( addr --) count type ;

: qfill ( addr --) $FF >expect ;

```

## 9.2 Methoden der objektorientierte Programmierung

Der METHODS>-Ansatz entbindet den Programmierer von der Pflicht, jedesmal den zum Datentyp passenden Operator auswählen zu müssen. Mit den generalisierten Operatoren `get` `put` und `show` wird die jeweils notwendige Zugriffsmethode aus einer Tabelle von Methoden ausgeführt .

Mittlerweile bildet sich allerdings als Bezeichnung für generalisierte Operatoren auch die Syntax `x@ x! xtype` ein.

```
\ Terry Rayburn  METHODS>          euroFORML 87
```

```
: Methods> [compile] Does>
  compile exit ; immediate restrict
```

```
\ early binding
```

```
: [Method]: Create 2* , immediate Does>
  here 2- @ @ 5 + swap @ + @ , ;
```

```
0 [Method]: [get] 1 [Method]: [put] 2 [Method]: [show]
```

```
\ late binding
```

```
: Method: Create 2* ,
  Does> @ over 2- @ @ 5 + + perform ;
```

```
0 Method: get 1 Method: put 2 Method: show
```

Der METHODS>-Ansatz wird in den Definitionen so eingesetzt:

```
: Integer:
  Variable Methods> @ ! u? ;
```

```
| : 2? 2@ d. ;
```

```
: Double:
  2Variable Methods> 2@ 2! 2? ;
```

```
: Queue: Create 0 c, 255 allot
  Methods> q@ q! q. ;
```

```
clear \ löscht die Namen der Worte auf dem Heap
```

Später im Programm wird dann mit den generalisierten Methoden auf die Datenstrukturen zugegriffen:

```
Integer: Konto          1000 Konto put  Konto show
Double: Moleküle        200.000 Moleküle put  Moleküle get d.
Queue: Puffer           Puffer qfill
                        Ascii A Puffer put
                        Puffer show
```

Die Methods>-Operatoren können mit den bekannten FORTH-Operatoren gemischt werden. Ebenso können die Tabelle beliebig geändert oder erweitert werden, so z.B. mit `4 Method: init` .

Soll diese Methode des Initialisierens nur bei `Integer:` eingesetzt werden, sind die anderen Tabellen entsprechend an der vierten Position aufzufüllen:

```

: Integer:
  Variable Methods> @ ! u? off ;

| : 2init ( addr -- ) 0. 2swap 2! ;
: Double:
  2Variable Methods> 2@ 2! 2? 2init ;

| : qinit ( -- )
  ." Pufferinitialisierung noch nicht definiert! " ;
: Queue: Create 0 c, 255 allot
  Methods> q@ q! q. qinit ;

```

Der Zugriff erfolgt dann wie gezeigt mit Konto init oder Druckpuffer init. Die logische Konsequenz ist die Verbindung beider Möglichkeiten zu :METHODS> :

```
: :Methods> [compile] :Does> compile exit ; immediate
```

:Methods> erwartet eine Reihe von Operatoren und weist diese (Zugriffs-) Methoden dem zuletzt definierten Wort zu. Es wird analog zu :DOES> und METHODS> in dieser Form eingesetzt:

```
Create Puffer 0 c, 255 allot
:Methods> q@ q! q. qinit ;
```

```
Variable Zähler :Methods> @ ! u? off ;
```

```
Puffer show Zähler init
```

## 10. Manipulieren des Systemverhalten

Das grundlegende Systemverhalten ist bei den traditionellen Programmiersprachen in der Laufzeit-Bibliothek (runtime library) festgelegt.

Diese runtime library enthält die grundlegenden Routinen, die bei der Ausführung der Programme vorhanden sein müssen wie z.B. die Bildschirmansteuerung oder den Massenspeicherzugriff.

In FORTH entspricht das Programm KERNEL.COM dieser Laufzeit-Bibliothek. Auf diesen Systemkern wird dann die fertige Anwendung, also Ihr Programm, geladen. Ein Beispiel für eine solche Applikationserstellung ist das Erstellen des volksFORTH-Arbeitssystems volks4th.com von der MS-DOS Kommandoebene aus:

```
A:>kernel include volks4th.sys
```

Im Gegensatz zu anderen Compilern, denen meist ein Assembler-Quelltext zugrunde liegt, ist KERNEL.COM aus einem FORTH-Quelltext durch METACOMPILATION erzeugt worden. Da sich ein FORTH-Quelltext besser pflegen und leichter ändern läßt als ein Assemblerprogramm, kann das grundlegende Systemverhalten in KERNEL.COM leicht neuen Erfordernissen anpassen.

### 10.1 Patchen von FORTH-Befehlen

Möchte man ohne MetaCompilation das Verhalten von Funktion in KERNEL.COM ändern, so "patcht" man dieses Programm - eine Vorgehensweise, die WordStar-Nutzern gut bekannt ist. Soll beispielsweise die Funktion des Wortes . (DOT) im Kern geändert werden, ist so vorzugehen:

```
\ "patchen" von :-Definitionen im Kern, z.B. ".":
' . >body @ Constant altdot
: new. ( n -- ) RDROP <new.action> ;

' new. ' . >body ! \ nun läuft die neue Version
altdot ' . >body ! \ und nun wieder die alte Version
```

Dabei ist es wichtig, die neue Arbeitsweise von . mit RDROP einzuleiten.

Im Gegensatz dazu können Sie die Eigenschaften Ihres volksFORTH-Arbeitssystems über den Inhalt von VOLKS4TH.SYS und durch direkte Änderung der Systemquelltexte individuell anpassen werden.

## 10.2 Verwendung von DEFER-Wörtern

Eine weitere Möglichkeit, das Systemverhalten zur Laufzeit zu beeinflussen, wird in dem Wort `name` gezeigt:

Dort sorgt ein `exit` vor dem Definitions-abschließenden Semicolon für einen freien Platz im Wort. In diesen freien Platz kann später mit Hilfe des Wortes `'name` ein weiteres Wort "eingehängt" werden, das dann beim Aufruf von `name` mit ausgeführt wird. Auf diese Weise wird das Gesamtverhalten nicht vollständig geändert, sondern um eine weitere Funktion ergänzt.

Oft möchte man auch das Verhalten einer Anwendung an bestimmten Punkten umschaltbar machen. Dies wird in `volksFORTH` erreicht durch:

```
Defer <name>
```

So kann man schon auf ein Wort Bezug nehmen, bevor es definiert wurde. Dazu reicht es, `volks4TH` den Namen bekannt zu geben. Später können immer wieder andere Routinen an Stelle dieses Wortes ausgeführt werden:

```
' <action> Is <name>
( bitte auf das Häkchen ' achten!)
```

Wird allerdings `<name>` ausgeführt, bevor es mit `IS` initialisiert wurde, so erscheint die Meldung "Crash" auf dem Bildschirm. Deshalb finden Sie oft diesen Ausdruck:

```
Defer <name>      ' noop Is <name>
```

So simpel ist die vektorielle Programmausführung in FORTH implementiert. Damit kann man zwei Probleme elegant lösen:

1. Das Wort `<name>` läßt sich bereits kompilieren, ohne daß ihm eine sinnvolle Aktion zugewiesen wurde. Damit ist die Kompilation von Wörtern möglich, die erst später definiert werden (Vorwärts-Referenzen).
2. Die nachträglich Veränderung von bereits kompilierten Wörtern ist damit möglich; vorausgesetzt, sie wurden als Defer-Wörter eingetragen. Wörter, die mit der Defer-Anweisung erzeugt wurden, lassen sich also in ihrem Ablaufverhalten nachträglich ändern. Diese Wörter behalten also ihren Namen bei, verändern aber ihr Verhalten.

Als Beispiel betrachten Sie bitte folgendes:

```
: Versuch#1 ." erster Versuch" cr ;
: Versuch#2 ." zweiter Versuch" cr ;
```

```
Defer Beispiel
```

```
' Versuch#1 Is Beispiel
```

```
Beispiel perform
```

Damit haben Sie jetzt **BEISPIEL** ausgeführt. Schauen Sie sich die Meldung an und geben ein:

```
' Versuch#2 Is Beispiel
```

Führen Sie erneut **Beispiel perform** aus - Sie bekommen nun die andere Meldung! Auf diese Art und Weise können Sie dem Wort **BEISPIEL** beliebige Funktionen zuweisen.

Die Kombination **DEFER/IS** behält also den Namen bei, ändert aber das Verhalten eines Wortes. Damit ist die Kombination **DEFER/IS** inhaltlich das Gegenteil von **ALIAS**, das den Namen ändert, aber die Funktion beibehält.

Mit **ALIAS** können Sie in **volksFORTH** einem Wort recht einfach einen neuen Namen geben. Sollte Ihnen das **V** des Editors nicht gefallen, benennen Sie es um in **WHERE** ! Dieses Umbenennen läßt sich auf zwei Arten durchführen, entweder im traditionellen FORTH-Stil mit

```
: where v ;
```

oder dem **volksFORTH** gemäß mit

```
' v Alias where
```

### 10.3 Neudefinition von Befehlen

Sollen dagegen Worte des Compilers unter anderem Namen benutzt werden, so ist das in FORTH ebenfalls kein Thema:

```
: NeuName AltName ;
' THEN Alias ENDIF immediate restrict
: ENDIF [compile] THEN ; immediate restrict
```

Eine Anwendung dieser Synonym-Deklarationen besteht darin, einen sogenannten Standard-Prolog zu verwirklichen: Damit Programme auch Wörter benutzen können, die der jeweilige Compiler nicht zur Verfügung stellt, schreibt man für den jeweiligen Compiler einen kleinen Vorspann (prelude) in FORTH83-Code zum Programm, der die jeweilig benötigten Definitionen enthält.

Oft stellt ein anderer Compiler eine identische Funktion unter einem anderen Namen zur Verfügung. So verwenden viele FORTH-Systeme **GOTOXY** statt des **AT** zur Cursorpositionierung. Nun ändert man im Prolog den Namen über die **Alias**-Funktion und erspart sich das quälende Suchen/Ersetzen im Editor.

#### 10.4 DEFERred Wörter im volksFORTH83

Darüberhinaus besitzt das volksFORTH83 eine Reihe von Strukturen, die dazu dienen, das Verhalten des Systems zu ändern. Im System sind bereits folgende DEFERred Worte vorhanden:

```
R/W 'COLD 'RESTART 'ABORT 'QUIT NOTFOUND
.STATUS DISKERR MAKEVIEW und CUSTOM-REMOVE .
```

Um sie zu ändern, benutzt man den Ausdruck:

```
' <action> Is <name>
```

Hierbei ist, wie oben besprochen, <name> ein durch DEFER erzeugtes Wort und <action> der Name des Wortes, das in Zukunft bei Aufruf von <name> ausgeführt wird.

##### Anwendungsmöglichkeiten dieser deferred Worte:

Durch Ändern von R/W kann man andere Floppies oder eine RAM-Disk betreiben. Es ist auch leicht möglich, Teile einer Disk gegen überschreiben zu schützen.

Durch Ändern von NOTFOUND kann man z.B. Worte, die nicht im Dictionary gefunden wurden, anschließend in einer Disk-Directory suchen lassen oder automatisch als Vorwärtsreferenz vermerken.

Ändert man 'COLD , so kann man die Einschaltmeldung des volksFORTH83 unterdrücken und stattdessen ein Anwenderprogramm starten, ohne daß Eingaben von der Tastatur aus erforderlich sind (siehe z.B. "Erstellen einer Applikation").

Ähnliches gilt für 'RESTART .

'ABORT ist z.B. dafür gedacht, eigene Stacks, z.B. für Fließkommazahlen, im Fehlerfall zu löschen. Die Verwendung dieses Wortes erfordert aber schon eine gewisse Systemkenntnis.

Das gilt auch für das Wort 'QUIT . 'QUIT wird dazu benutzt, eigene Quitloops in den Compiler einzubetten. Wer sich für diese Materie interessiert, sollte sich den Quelltext des Tracer anschauen. Dort wird vorgeführt, wie man das macht.

.STATUS schließlich wird vor Laden eines Blocks ausgeführt. Man kann sich damit anzeigen lassen, welchen Block einer längeren Sequenz das System gerade lädt und z.B. wieviel freier Speicher noch zur Verfügung steht.

CUSTOM-REMOVE kann vom fortgeschrittenen Programmierer dazu benutzt werden, eigene Datenstrukturen, die miteinander durch Zeiger verkettet sind, zu vergessen. Ein Beispiel dafür sind die File Control Blöcke des Fileinterfaces.



## 10.5 Vektoren im volksFORTH83

Es gibt im System die Uservariable `ERRORHANDLER`. Dabei handelt es sich um eine normale Variable, die als Inhalt die Kompilationsadresse eines Wortes hat. Der Inhalt der Variablen wird auf folgende Weise ausgeführt:

```
errorhandler perform
```

Zuweisen und Auslesen dieser Variablen geschieht mit `@` und `!`. Der Inhalt von `ERRORHANDLER` wird ausgeführt, wenn das System `ABORT` oder `ERROR` ausführt und das von diesen Worten verbrauchte Flag wahr ist. Die Adresse des Textes mit der Fehlermeldung befindet sich auf dem Stack. Siehe z.B. `(ERROR)`.

Das volksFORTH83 benutzt die indirekten Vektoren `INPUT` und `OUTPUT`. Die Funktionsweise der sich daraus ergebenden Strukturen soll am Beispiel von `INPUT` verdeutlicht werden:

`INPUT` ist eine Uservariable, die auf einen Vektor zeigt, in dem wiederum vier Kompilationsadressen abgelegt sind. Jedes der vier Inputworte `KEY` `KEY?` `DECODE` und `EXPECT` führt eine der Kompilationsadressen aus. Kompiliert wird solch ein Vektor in der folgenden Form:

```
Input: vector <name1> <name2> <name3> <name4> ;
```

Wird `VECTOR` ausgeführt, so schreibt er seine Parameterfeld-Adresse in die UserVariable `INPUT`. Von nun an führen die Inputworte die ihnen so zugewiesenen Worte aus. `KEY` führt also `<NAME1>` aus usw...

Das Beispiel `KEY?` soll dieses Prinzip verdeutlichen:

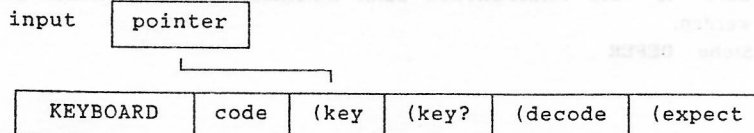
```
: key? ( -- c ) input @ 2+ perform ;
```

Tatsächlich wurde `KEY?` im System ebenso wie die anderen Inputworte durch das nicht mehr sichtbare definierende Wort `IN:` erzeugt. Ein Beispiel für einen Inputvektor, der die Eingabe von der Tastatur holt:

```
Input: keyboard
(key (key? (decode (expect ;
```

```
keyboard
```

ergibt:



Analog verhält es sich mit OUTPUT und den Outputworten EMIT CR TYPE DEL PAGE AT und AT?. Outputvektoren werden mit OUTPUT: genauso wie die Inputvektoren erzeugt.

Bitte beachten Sie, daß immer alle Worte in der richtigen Reihenfolge aufgeführt werden müssen! Soll nur ein Wort geändert werden, so müssen sie trotzdem die anderen mit hinschreiben.

## 10.6 Glossar

Defer

( -- )

ein definierendes Wort, das in der Form:

Defer <name>

benutzt wird. Erzeugt den Kopf für ein neues Wort <name> im Dictionary, hält 2 Byte in dessen Parameterfeld frei und speichert dort zunächst die Kompilationsadresse einer Fehlerroutine.

Wird <name> nun ausgeführt, so wird eine Fehlerbehandlung eingeleitet. Man kann dem Wort <name> jedoch zu jeder Zeit eine andere Funktion zuweisen mit der Sequenz:

' <action> Is <name>

Nach dieser Zuweisung verhält sich <name> wie <action>.

Mit diesem Mechanismus kann man zwei Probleme elegant lösen:

Einerseits läßt sich <name> bereits kompilieren, bevor ihm eine sinnvolle Aktion zugewiesen wurde. Damit ist die Kompilation erst später definierter Worte (Vorwärts-Referenzen) indirekt möglich.

Andererseits ist die Veränderung des Verhaltens von <name> für spezielle Zwecke auch nachträglich möglich, ohne neu kompilieren zu müssen.

Deferred Worte im System sind: R/W 'COLD 'RESTART 'ABORT 'QUIT NOTFOUND .STATUS und DISKERR. Diese Worte sind DEFERred, damit ihr Verhalten für die Anwendung geändert werden kann.

Is

( cfa -- )

ist ein Wort, mit dem das Verhalten eines deferred Wortes verändert werden kann. IS wird in der Form:

' <action> Is <name>

benutzt. Wenn <name> kein deferred Wort ist, wird eine Fehlerbehandlung eingeleitet, sonst verhält sich <name> anschließend wie <action>. Das Wort IS des volksFORTH83 kann innerhalb von Definitionen benutzt werden.

Siehe DEFER.

**perform** ( addr -- )  
 erwartet eine Adresse, unter der sich ein Zeiger auf die Kompilations-  
 adresse eines Wortes befindet. Dieses Wort wird dann ausgeführt.  
 perform entspricht der Anweisungsfolge @ execute .

**noop** ( -- )  
 macht gar nichts.

**Alias** ( cfa -- )  
 ist ein definierendes Wort, das typisch in der Form:  
 ' <oldname> Alias <newname>  
 benutzt wird. ALIAS erzeugt einen Kopf für <newname> im Dictionary.  
 Wird <newname> aufgerufen, so verhält es sich wie <oldname>. Insbeson-  
 dere wird beim Kompilieren nicht <newname>, sondern <oldname> im Dic-  
 tionary eingetragen.  
 Im Unterschied zu : <newname> <oldname> ; ist es mit ALIAS möglich,  
 Worte, die den Returnstack beeinflussen (z.B. >r oder r ), mit an-  
 derem Namen zu definieren. Außer dem neuen Kopf für <newname> wird  
 kein zusätzlicher Speicherplatz verbraucht. Gegenwärtig wird bei Aus-  
 führung von >name aus einer CFA in der Regel der letzte mit ALIAS  
 erzeugte Name gefunden.

**Input:** ( -- ) "input-colon"  
 ist ein definierendes Wort, benutzt in der Form:  
 Input: <name> newKEY newKEY? newDECODE newEXPECT ;  
 INPUT: erzeugt einen Kopf für <name> im Dictionary und kompiliert einen  
 Satz von Zeigern auf Worte, die für die Eingabe von Zeichen zuständig  
 sind. Wird <name> ausgeführt, so wird ein Zeiger auf das Parameterfeld  
 von <name> in die Uservariable INPUT geschrieben. Alle Eingaben  
 werden jetzt über die neuen Eingabeworte abgewickelt.  
 Die Reihenfolge der Worte nach INPUT: <name> bis zum Semicolon muß  
 eingehalten werden:  
 ..key ..key? ..decode ..expect  
 Im System ist das mit INPUT: definierte Wort keyboard enthalten,  
 nach dessen Ausführung alle Eingaben von der Tastatur geholt werden.  
 Siehe DECODE und EXPECT .

- Output: ( -- ) "output-colon"  
 ist ein definierendes Wort, benutzt in der Form:  
 Output: <name>  
 newEMIT newCR newTYPE newDEL newPAGE newAT newAT? ;  
 OUTPUT: erzeugt einen Kopf für <name> im Dictionary und kompiliert einen Satz von Zeigern auf Worte, die für die Ausgabe von Zeichen zuständig sind. Wird <name> ausgeführt, so wird ein Zeiger auf das Parameterfeld von <name> in die Uservariable OUTPUT geschrieben. Alle Ausgaben werden jetzt über die neuen Ausgabeworte abgewickelt. Die Reihenfolge der Worte nach OUTPUT: <name> muß eingehalten werden:  
 ..emit ..cr ..type ..del ..page ..at ..at?  
 Im System ist das mit OUTPUT: definierte Wort DISPLAY enthalten, nach dessen Ausführung alle Ausgaben auf den Bildschirm geleitet werden.  
 Vergleiche auch das Wort PRINT aus dem Printer-Interface.
- 'abort ( -- ) "tick-abort"  
 ist ein deferred Wort, das mit NOOP vorbesetzt ist. Es wird in ABORT ausgeführt, bevor QUIT aufgerufen wird. Es kann benutzt werden, um "automatisch" selbstdefinierte Stacks zu löschen.
- 'cold ( -- ) "tick-cold"  
 ist ein deferred Wort, das mit NOOP vorbesetzt ist. Es wird in COLD aufgerufen, bevor die Einschaltmeldung ausgegeben wird. Es wird benutzt, um Geräte zu initialisieren oder Anwenderprogramme automatisch zu starten.
- 'quit ( -- ) "tick-quit"  
 ist ein deferred Wort, das normalerweise mit (QUIT besetzt ist. Es wird in QUIT aufgerufen, nachdem der Returnstack enleert und der interpretierende Zustand eingeschaltet wurde. Es wird benutzt, um spezielle Kommandointerpreter (wie z.B. im Tracer) aufzubauen.
- 'restart ( -- ) "tick-restart"  
 Dies ist ein deferred Wort, das mit NOOP vorbesetzt ist. Es wird in RESTART aufgerufen, nachdem 'QUIT mit (QUIT besetzt wurde. Es wird benutzt, um Geräte nach einem Warmstart zu re-initialisieren.
- (quit ( -- ) "paren-quit"  
 Dieses Wort ist normalerweise der Inhalt von 'QUIT. Es wird von QUIT benutzt. Es akzeptiert eine Zeile von der aktuellen Eingabeeinheit, führt sie aus und druckt "ok" bzw. "compiling".

.status ( -- ) "dot-status"  
 Dieses Wort ist ein deferred Wort, das vor dem Einlesen einer Zeile bzw. dem Laden eines Blocks ausgeführt wird. Es ist mit NOOP vorbesetzt und kann dazu benutzt werden, Informationen über den Systemzustand oder den Quelltext auszugeben.

makeview ( -- 16b)  
 ist ein deferred Wort, dem mit IS ein Wort zugewiesen wurde. Dieses Wort erzeugt aus dem gerade kompilierten Block eine 16b Zahl, die von Create in das Viewfeld eingetragen wird. Das deferred Wort wird benötigt, um das Fileinterface löschen zu können.

## 11. Vokabular-Struktur

Eine Liste von Worten ist ein Vokabular. Ein FORTH-System besteht im allgemeinen aus mehreren Vokabularen, die nebeneinander existieren. Neue Vokabulare werden durch das definierende Wort `VOCABULARY` erzeugt und haben ihrerseits einen Namen, der in einer Liste enthalten ist. Gewöhnlich kann von mehreren Worten mit gleichem Namen nur das zuletzt definierte erreicht werden. Befinden sich jedoch die einzelnen Worte in verschiedenen Vokabularen, so bleiben sie einzeln erreichbar.

### 11.1 Die Suchreihenfolge

Die Suchreihenfolge gibt an, in welcher Reihenfolge die verschiedenen Vokabulare nach einem Wort durchsucht werden.

Sie besteht aus zwei Teilen, dem auswechselbaren und dem festen Teil. Der auswechselbare Teil enthält genau ein Vokabular. Dies wird zuerst durchsucht. Wird ein Vokabular durch Eingeben seines Namens ausgeführt, so trägt es sich in den auswechselbaren Teil ein. Dabei wird der alte Inhalt überschrieben. Einige andere Worte ändern ebenfalls den auswechselbaren Teil. Soll ein Vokabular immer durchsucht werden, so muß es in den festen Teil übertragen werden. Dieser enthält null bis sechs Vokabulare und wird nur vom Benutzer bzw. seinen Worten verändert. Zur Manipulation stehen u.a. die Worte `ONLY` `ALSO` `TOSS` zur Verfügung. Das Vokabular, in das neue Worte einzutragen sind, wird durch das Wort `DEFINITIONS` angegeben. Die Suchreihenfolge kann man sich mit `ORDER` ansehen.

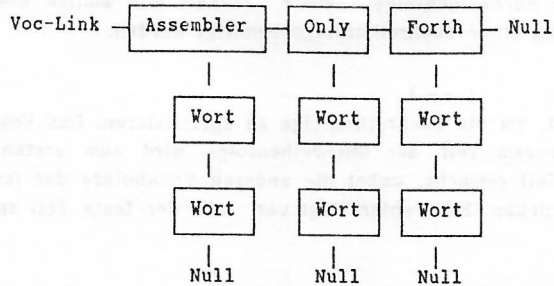
Ein Beispiele, das die Veränderung der Suchreihenfolge mit den eingegebenen Befehlen zeigt, wobei das Grundvokabular `ROOT` zwar in der Suchreihenfolge enthalten ist, aber nicht von `ORDER` angezeigt wird, um die Statuszeile nicht mit obligatorischen Informationen zu blockieren:

<u>Eingabe:</u>	<u>ORDER ergibt dann:</u>
Onlyforth	FORTH FORTH ROOT FORTH
Editor also	EDITOR EDITOR FORTH ROOT FORTH
Assembler	ASSEMBLER EDITOR FORTH ROOT FORTH
definitions Forth	FORTH EDITOR FORTH ROOT ASSEMBLER
: test ;	ASSEMBLER EDITOR FORTH ROOT ASSEMBLER

Hierbei ist vor allem auf colon (:) zu achten, das ebenfalls die Suchlaufpriorität ändert, indem es das Kompilationsvokabular `current` in den auswechselbaren Teil von `context` überträgt.

Der Inhalt eines Vokabulars besteht aus einer Liste von Worten, die durch ihre Linkfelder miteinander verbunden sind. Es gibt also genauso viele Listen wie Vokabulare. Alle Vokabulare sind selbst noch einmal über eine Liste verbunden,

deren Anfang in VOC-LINK steht. Diese Verkettung ist nötig, um ein komfortables FORGET zu ermöglichen. Man bekommt beispielsweise folgendes Bild:



## 11.2 Glossar

Vocabulary ( -- ) 83

ein definierendes Wort, das in der Form:

Vocabulary <name>

benutzt wird.

VOCABULARY erzeugt einen Kopf für <name>, das den Anfang einer neuen Liste von Worten bildet. Wird <name> ausgeführt, so werden bei der Suche im Dictionary zuerst die Worte in der Liste von <name> berücksichtigt. Wird das VOCABULARY <name> durch die Sequenz:

<name> definitions

zum Kompilations-Vokabular, so werden neue Wort-Definitionen in die Liste von <name> gehängt. Vergleiche auch CONTEXT CURRENT ALSO TOSS ONLY FORTH ONLYFORTH .

context ( -- addr )

addr ist die Adresse des auswechselbaren Teils der Suchreihenfolge. Sie enthält einen Zeiger auf das erste zu durchsuchende Vokabular.

current ( -- addr )

addr ist die Adresse eines Zeigers, der auf das Kompilationsvokabular zeigt, in das neue Worte eingefügt werden.

definitions ( -- ) 83

ersetzt das gegenwärtige Kompilationsvokabular durch das Vokabular im auswechselbaren Teil der Suchreihenfolge, d.h. neue Worte werden in dieses Vokabular eingefügt.

- Only ( -- )  
Das Nennen dieses Vokabular löscht die Suchreihenfolge vollständig und ersetzt sie durch das Vokabular ROOT im festen und auswechselbaren Teil der Suchreihenfolge. ROOT enthält nur wenige Worte, die für die Erzeugung einer Suchreihenfolge benötigt werden.
- also ( -- )  
Ein Wort, um die Suchreihenfolge zu spezifizieren. Das Vokabular im auswechselbarem Teil der Suchreihenfolge wird zum ersten Vokabular im festen Teil gemacht, wobei die anderen Vokabulare des festen Teils nach hinten rücken. Ein Fehler liegt vor, falls der feste Teil sechs Vokabulare enthält.
- toss ( -- )  
entfernt das erste Vokabular des festen Teils der Suchreihenfolge. Insofern ist es das Gegenstück zu ALSO .
- seal ( -- )  
löscht das Vokabular ROOT , so daß es nicht mehr durchsucht wird. Dadurch ist es möglich, nur die Vokabulare des Anwenderprogramms durchsuchen zu lassen.
- Onlyforth ( -- )  
entspricht der häufig benötigten Sequenz:  
ONLY FORTH ALSO DEFINITIONS
- Forth ( -- ) 83  
Das ursprüngliche Vokabular.
- Assembler ( -- )  
Ein Vokabular, das Prozessor-spezifische Worte enthält, die für Code-Definitionen benötigt werden.
- words ( -- )  
gibt die Namen der Worte des Vokabulars, das im auswechselbaren Teil der Suchreihenfolge steht, aus, beginnend mit dem zuletzt erzeugtem Namen.
- forth-83 ( -- ) 83  
Laut FORTH83-Standard soll dieses Wort sicherstellen, daß ein Standard-system benutzt wird. Im volksFORTH funktionslos.





## 12. Dictionary-Struktur

Das FORTH-System besteht aus einem Dictionary von Worten. Die Struktur der Worte und des Dictionaries soll im folgenden erläutert werden.

### 12.1 Aufbau

Die FORTH-Worte sind in Listen angeordnet (s.a. Struktur der Vokabulare). Die vom Benutzer definierten Worte werden ebenfalls in diese Listen eingetragen.

Jedes Wort besteht aus sechs Teilen. Es sind dies:

1. block                      Der Nummer des Blocks, in dem das Wort definiert wurde (siehe auch VIEW ).
2. link                        Die Adresse (Zeiger) namens lfa , die auf das "Linkfeld" des nächsten Wortes zeigt.
3. count                      Die Länge des Namens dieses Wortes und drei Markierungsbits. Die Adresse nfa zeigt auf dieses Byte, ebenso last .
4. name                        Der Name selbst.
5. code                        Eine Adresse (Zeiger), die auf den Maschinencode zeigt, der bei Aufruf dieses Wortes ausgeführt wird. Die Adresse dieses Feldes heißt Kompilationsadresse cfa .
6. parameter    Das Parameterfeld; die Adresse dieses Feldes heißt pfa .

Ein Wort sieht dann so aus :

Wort					
block	link	count	name	code	parameter ...

Im folgenden sollen diese sechs Felder einzeln detailliert betrachtet werden.

**Block**                                      Das Blockfeld enthält in codierter Form die Nummer des Blocks und den Namen des Files, in dem das Wort

definiert wurde. Wurde es von der Tastatur aus eingegeben, so enthält das Feld Null.

Link

Über das Linkfeld sind die Worte eines Vokabulars zu einer Liste verkettet. Jedes Link-Feld enthält die Adresse des vorherigen Link-Feldes. Jedes Wort zeigt also auf seinen Vorgänger. Das unterste Wort der Liste enthält im Link-Feld eine Null. Die Null zeigt das Ende der Liste an.

Countfeld

Das count field enthält die Länge des Namens (1..31 Zeichen) und drei Markierungsbits :

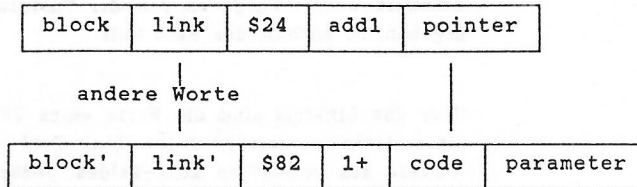
	restrict	immediate	indirect	Länge
Bit:	7	6	5	4..0

Ist das immediate-Bit gesetzt, so wird das entsprechende Wort im kompilierenden Zustand unmittelbar ausgeführt, und nicht ins Dictionary kompiliert (siehe auch IMMEDIATE).

Ist das restrict-Bit gesetzt, so kann das Wort nicht durch Eingabe von der Tastatur ausgeführt, sondern nur in anderen Worten kompiliert werden. Gibt man es dennoch im interpretierenden Zustand ein, so erscheint die Fehlermeldung "compile only" (siehe auch RESTRICT).

Ist das indirect-Bit gesetzt, so folgt auf den Namen kein Codefeld, sondern ein Zeiger darauf. Damit kann der Name vom Rumpf ( Code- und Parameterfeld ) getrennt werden. Die Trennung geschieht z.B. bei Verwendung der Worte ! oder ALIAS.

Beispiel: ' 1+ Alias add1  
ergibt folgende Struktur im Speicher (Dictionary) :

**Name**

Der Name besteht normalerweise aus ASCII-Zeichen. Bei der Eingabe werden Klein- in Großbuchstaben umgewandelt. Daher druckt **WORDS** auch nur großgeschriebene Namen.

Da Namen sowohl groß als auch klein geschrieben eingegeben werden können, haben wir eine Konvention erarbeitet, die die Schreibweise von Namen festlegt:

Bei Kontrollstrukturen wie **DO LOOP** etc. werden alle Buchstaben groß geschrieben.

Bei Namen von Vokabularen, immediate Worten und definierenden Worten, die **CREATE** ausführen, wird nur der erste Buchstabe groß geschrieben.

Beispiele sind: **Is FORTH Constant**

Alle anderen Worte werden klein geschrieben.

Beispiele sind: **dup cold base**

Bestimmte Worte, die von immediate Worten kompiliert werden, beginnen mit der öffnenden Klammer "(" , gefolgt vom Namen des immediate Wortes.

Ein Beispiel: **DO** kompiliert (**do** .

Diese Schreibweise ist nicht zwingend; Sie sollten sich aber daran halten, um die Lesbarkeit Ihrer Quelltexte zu erhöhen.

**Code**

Jedes Wort weist auf ein Stück Maschinencode. Die Adresse dieses Code-Stücks ist im Codefeld enthalten. Gleiche Worttypen weisen auf den gleichen Code. Es gibt verschiedene Worttypen, z.B. :-Definitionen, Variablen, Konstanten, Vokabulare usw. Sie haben jeweils ihren eigenen charakteristischen Code gemeinsam. Die Adresse des Code-Feldes heißt Kompilationsadresse.

**Parameter**

Das Parameterfeld enthält Daten, die vom Worttypen abhängen. Beispiele :

a) Typ "Constant"

Hier enthält das Parameterfeld des Wortes den Wert

der Konstanten. Der dem Wort zugeordnete Code liest den Inhalt des Parameterfeldes aus und legt ihn auf den Stack.

b) Typ "Variable"

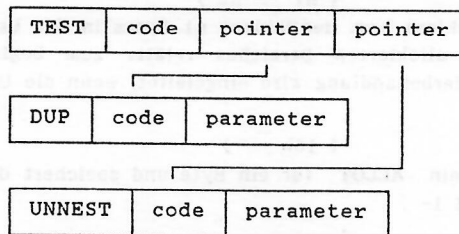
Das Parameterfeld enthält den Wert der Variablen, der zugeordnete Code liest jedoch nicht das Parameterfeld aus, sondern legt dessen Adresse auf den Stack. Der Benutzer kann dann mit dem Wort @ den Wert holen und mit dem Wort ! überschreiben.

c) Typ ":-definition"

Das ist ein mit : und ; gebildetes Wort. In diesem Fall enthält das Parameterfeld hintereinander die Kompilationsadressen der Worte, die diese Definition bilden. Der zugeordnete Code sorgt dann dafür, daß diese Worte der Reihe nach ausgeführt werden. Beispiel :

: test dup ;

ergibt:



Das Wort : hat den Namen TEST erzeugt. UNNEST wurde durch das Wort ; erzeugt

d) Typ "Code"

Worte vom Typ "Code" werden mit dem Assembler erzeugt. Hier zeigt das Codefeld in der Regel auf das Parameterfeld. Dorthin wurde der Maschinencode assembliert.

Codeworte im volksFORTH können leicht "umgepatcht" werden, da lediglich die Adresse im Codefeld auf eine

neue (andere) Maschinencodesequenz gesetzt werden muß.

## 12.2 Glossar

- here ( -- addr ) 83  
addr ist die Adresse des nächsten freien Dictionaryplatzes.
- dp ( -- addr ) "d-p"  
Eine Uservariable, die die Adresse des nächsten freien Dictionaryplatzes enthält.
- udp ( -- addr ) "u-d-p"  
Eine Uservariable, in dem das Ende der bisher allokierten Userarea vermerkt ist.
- allot ( w -- ) 83  
Allokiere w Bytes im Dictionary. Die Adresse des nächsten freien Dictionaryplatzes wird entsprechend verstellt.
- uallot ( n1 -- n2 )  
Allokiere bzw. deallokiere n1 Bytes in der Userarea. n2 gibt den Anfang des allokierten Bereiches relativ zum Beginn der Userarea an. Eine Fehlerbehandlung wird eingeleitet, wenn die Userarea voll ist.
- c, ( 16b -- ) "c-comma"  
ist ein ALLOT für ein Byte und speichert die unteren 8 Bit von 16b in HERE 1- .
- , ( 16b -- ) 83 "comma"  
ist 2 ALLOT für 16b und speichere 16b ab HERE 2- .
- ' ( -- addr ) 83 "tick"  
Wird in der Form ' <name> benutzt.  
addr ist die Kompilationsadresse von <name>. Wird <name> nicht in der Suchreihenfolge gefunden, so wird eine Fehlerbehandlung eingeleitet.

- `name` ( -- )  
Eine Besonderheit von `name` ist, daß dieses Wort über zwei EXITS verfügt. Zum einen das explizite `exit` und zum anderen das Semicolon, das ja auch ein `exit` ist. Die so einkompilierte Adresse kann dazu benutzt werden, beim Aufruf von `name` ein weiteres Wort ausführen zu lassen. Dieser Mechanismus kann als eine Variation von Deferred Worten aufgefaßt werden, weil auch hier das Wortverhalten geändert wird. Siehe 'NAME und auch DEFER .
- `'name` ( -- addr )  
liefert die Bezugsadresse für das Einhängen eines Wortes, wie es bei `name` dargestellt wurde. Diese Möglichkeit, das Verhalten von Worten zu erweitern, wird bsp. im Editor eingesetzt. Dort wird im Wort `showload` ein Wort `show` über `'name` in `name` eingehängt und dann in `showoff` durch das Eintragen von `exit` wieder überschrieben, d.h. ausgehängt.
- `.name` ( addr -- ) "dot-name"  
`addr` ist die Adresse des Countfeldes eines Namens. Dieser Name wird ausgedruckt. Befindet er sich im Heap, so wird das Zeichen `|` vorangestellt. Ist `addr` Null, so wird "???" ausgegeben.
- `align` ( -- )  
macht nichts, weil der 8086/88 Prozessor auch von ungeraden Adressen Befehle holen kann, im Gegensatz zum 68000er. Diese Worte sind vorhanden, damit Sourcecode zwischen den Systemen transportabel ist. Querverweis: HALIGN .
- `forget` ( -- ) 83  
Wird in der Form `FORGET <name>` benutzt. Falls `<name>` in der Suchreihenfolge gefunden wird, so werden `<name>` und alle danach definierten Worte aus dem Dictionary entfernt. Wird `<name>` nicht gefunden, so wird eine Fehlerbehandlung eingeleitet. Liegt `<name>` in dem durch `SAVE` geschützten Bereich, so wird ebenfalls eine Fehlerbehandlung eingeleitet. Es wurden Vorkehrungen getroffen, die es ermöglichen, aktive Tasks und Vokabulare, die in der Suchreihenfolge auftreten, zu vergessen.

- remove** ( dic sym thread -- dic sym )  
Dies ist ein Wort, das zusammen mit **CUSTOM-REMOVE** verwendet wird. dic ist die untere Grenze des Dictionarybereiches, der vergessen werden soll und sym die obere. Typisch zeigt sym in den Heap. thread ist der Anfang einer Kette von Zeigern, die durch einen Zeiger mit dem Wert Null abgeschlossen wird. Wird **REMOVE** dann ausgeführt, so werden alle Zeiger (durch Umhängen der übrigen Zeiger) aus der Liste entfernt, die in dem zu vergessenden Dictionarybereich liegen. Dadurch ist es möglich, **FORGET** und ähnliche Worte auf Datenstrukturen anzuwenden.
- (forget** ( addr -- ) "paren-forget"  
Entfernt alle Worte, deren Kompilationsadresse oberhalb von addr liegt, aus dem Dictionary und setzt **HERE** auf addr. Ein Fehler liegt vor, falls addr im Heap liegt.
- custom-remove** ( dic symb -- dic symb )  
Ein deferred Wort, daß von **FORGET**, **CLEAR** usw. aufgerufen wird. dic ist die untere Grenze des Dictionaryteils, der vergessen wird und symb die obere. Gewöhnlich zeigt symb in den Heap.  
Dieses Wort kann dazu benutzt werden, eigene Datenstrukturen, die Zeiger enthalten, bei **FORGET** korrekt abzuabearbeiten. Es wird vom Fileinterface verwendet, daher darf es nicht einfach überschrieben werden. Man kann es z.B. in folgender Form benutzen :
- ```

: <name> [ ' custom-remove >body @ , ]
  <liststart> @ remove ;
' <name> Is custom-remove

```
- Auf diese Weise stellt man sicher, daß das Wort, das vorher in **CUSTOM-REMOVE** eingetragen war, weiterhin ausgeführt wird.  
Siehe auch **REMOVE**.
- save** ( -- )  
Kopiert den Wert aller Uservariablen in den Speicherbereich ab **ORIGIN** und sichert alle Vokabularlisten. Wird später **COLD** ausgeführt, so befindet sich das System im gleichen Speicherzustand wie bei Ausführung von **SAVE**.
- empty** ( -- )  
Löscht alle Worte, die nach der letzten Ausführung von **SAVE** oder dem letzten Kaltstart definiert wurden. **DP** (und damit **HERE**) wird auf seinen Kaltstartwert gesetzt und der Heap gelöscht.
- last** ( -- addr )  
Variable, die auf das Countfeld des zuletzt definierten Wortes zeigt.  
Siehe auch **RECURSIVE** und **MYSELF**.



- hide ( -- )  
Entfernt das zuletzt definierte Wort aus der Liste des Vokabulars, in das es eingetragen wurde. Dadurch kann es nicht gefunden werden. Es ist aber noch im Speicher vorhanden. (s.a. REVEAL LAST)
- reveal ( -- )  
Trägt das zuletzt definierte Wort in die Liste des Vokabulars ein, in dem es definiert wurde.
- origin ( -- addr )  
addr ist die Adresse, ab der die Kaltstartwerte der Uservariablen abgespeichert sind.
- name> ( addr1 -- addr2 ) "name-from"  
addr2 ist die Kompilationsadresse, die mit dem Countfeld in addr1 korrespondiert.
- >body ( addr1 -- addr2 ) "to-body"  
addr2 ist die Parameterfeldadresse, die mit der Kompilationsadresse addr1 korrespondiert.
- >name ( addr1 -- addr2 ) "to-name"  
addr2 ist die Adresse eines Countfeldes, das mit der Kompilationsadresse addr1 korrespondiert. Es ist möglich, daß es mehrere addr2 für ein addr1 gibt. In diesem Fall ist nicht definiert, welche ausgewählt wird.

In der Literatur finden Sie für die Umrechnung von Feld-Adressen auch oftmals folgende Worte:

```

: >link    >name 2- ;
: link>    2+ name> ;
: n>link   2- ;
: l>name   2+ ;

```

## 13. Der HEAP

Eines der ungewöhnlichen und fortschrittlichen Konzepte des volksFORTH83 besteht in der Möglichkeit, Namen von Worten zu entfernen, ohne den Rumpf zu vernichten (headerless words).

Das ist insbesondere während der Kompilation nützlich, denn Namen von Worten, deren Benutzung von der Tastatur aus nicht sinnvoll wäre, tauchen am Ende der Kompilation auch nicht mehr im Dictionary auf. Man kann dem Quelltext sofort ansehen, ob ein Wort für den Gebrauch außerhalb des Programmes bestimmt ist oder nicht.

Die Namen, die entfernt wurden, verbrauchen natürlich keinen Speicherplatz mehr. Damit wird die Verwendung von mehr und längeren Namen und dadurch auch die Lesbarkeit gefördert.

Namen, die später eliminiert werden sollen, werden durch das Wort `!` gekennzeichnet. Das Wort `!` muß unmittelbar vor dem Wort stehen, das den zu eliminierenden Namen erzeugt. Der so erzeugte Name wird in einem besonderen Speicherbereich, dem Heap, abgelegt. Der Heap kann später mit dem Wort `CLEAR` gelöscht werden. Dann sind natürlich auch alle Namen, die sich im Heap befanden, verschwunden. Das folgende Beispiel soll eine Art Taschenrechner darstellen:

```
! Variable sum 1 sum !
```

Es werden weitere Worte definiert und dann `CLEAR` ausgeführt:

```
: clearsum ( -- ) 0 sum ! ;
: add      ( n -- ) sum +! ;
: show     ( -- ) sum @ . ;
```

```
clear
```

Diese Definitionen liefern die Worte `CLEARSUM` `ADD` und `SHOW`, während der Name der Variablen `SUM` durch `CLEAR` entfernt wurde. Das Codefeld und der Wert `0001` existieren jedoch noch. Man kann den Heap auch dazu "mißbrauchen", Code, der nur zeitweilig benötigt wird, nachher wieder zu entfernen. Der Assembler wird auf diese Art geladen, so daß er nach Fertigstellen der Applikation mit `CLEAR` wieder entfernt werden kann und keinen Platz im Speicher mehr benötigt.

### Glossar

`!` ( -- ) "headerless"  
setzt bei Ausführung `?HEAD` so, daß der nächste erzeugte Name nicht im normalen Dictionaryspeicher angelegt wird, sondern auf dem Heap.

- clear** ( -- )  
löscht alle Namen und Worte im Heap, so daß vorher mit `!` definierte Worte nicht mehr benutzt werden können. Auf diese Weise kann der Geltungsbereich von Prozeduren eingeschränkt werden.
- heap** ( -- addr )  
Hier ist `addr` der Anfang des Heap. Er wird z.B. durch `HALLLOT` geändert.
- hallot** ( n -- )  
allokiert bzw. deallokiert `n` Bytes auf dem Heap. Dabei wird der Stack verschoben, ebenso wie der Beginn des Heap.
- heap?** ( addr -- flag ) "heap-question"  
übergibt ein wahres Flag, wenn `addr` ein Byte im Heap adressiert, ansonsten `FALSE`.
- ?head** ( -- addr ) "question-head"  
ist eine Variable, die angibt, ob und wieviele der nächsten zu erzeugenden Namen im Heap angelegt werden sollen.  
Siehe auch `!`.
- halign** ( -- ) "h-align"  
dient nur der Softwarekompatibilität zu den anderen `volks4TH`-Systemen, da der `8086`-Prozessor nicht `align-ed` werden braucht.  
Querverweis: `ALIGN`.

## 14. Die Ausführung von FORTH-Worten

Der geringe Platzbedarf übersetzter FORTH-Worte rührt wesentlich von der Existenz des Adressinterpreters her.

Wie im Kapitel über die Dictionary-Struktur beschrieben, besteht eine :-Definition aus dem Codefeld und dem Parameterfeld. Im Parameterfeld steht eine Folge von Adressen. Ein Wort wird kompiliert, indem seine Kompilationsadresse dem Parameterfeld der :-Definition angefügt wird. Eine Ausnahme bilden die Immediate-Worte. Da sie während der Kompilation ausgeführt werden, können sie dem Parameterfeld der :-Definition alles mögliche hinzufügen. Daraus wird klar, daß die meisten Worte innerhalb der :-Definition nur eine Adresse, also in einem 16-Bit-System genau 2 Bytes an Platz verbrauchen. Wird die :-Definition nun aufgerufen, so sollen alle Worte, deren Kompilationsadresse im Parameterfeld stehen, ausgeführt werden. Das besorgt der Adressinterpreter.

### 14.1 Der Aufbau des Adressinterpreters

Beim volksFORTH83 benutzt der Adressinterpreter einige Register der CPU, die im Kapitel über den Assembler aufgeführt werden. Es gibt aber mindestens die folgenden Register :

IP ist der Instruktionszeiger (Instructionpointer ). Er zeigt auf die nächste auszuführende Instruktion. Das ist beim volks-FORTH83 die Speicherzelle, die die Kompilationsadresse des nächsten auszuführenden Wortes enthält.

W ist das Wortregister. Es zeigt auf die Kompilationsadresse des Wortes, das gerade ausgeführt wird.

SP ist der (Daten-) Stackpointer. Er zeigt auf das oberste Element des Stacks.

RP ist der Returnstackpointer. Er zeigt auf das oberste Element des Returnstacks.

### 14.2 Die Funktion des Adressinterpreters

NEXT ist die Anfangsadresse der Routine, die die Instruktion ausführt, auf die IP gerade zeigt. Die Routine NEXT ist ein Teil des Adressinterpreters. Zur Verdeutlichung der Arbeitsweise ist dieser Teil hier in High Level geschrieben:

```
Variable IP
Variable W

: Next
```

```
IP @ @ W !
2 IP +!
W perform ;
```

Tatsächlich ist NEXT jedoch eine Maschinencoderoutine, weil dadurch die Ausführungszeit von FORTH-Worten erheblich kürzer wird. NEXT ist somit ein Makro, die diejenige Instruktion ausführt, auf die das Register IP zeigt.

Ein Wort wird ausgeführt, indem der Code, auf den die Kompilations-adresse zeigt, als Maschinencode angesprochen wird. Der Code kann z.B. den alten Inhalt des IP auf den Returnstack bringen, die Adresse des Parameterfeldes im IP speichern und dann NEXT anspringen. Diese Routine gibt es wirklich, sie heißt "docol" und ihre Adresse steht im Codefeld jeder :-Definition.

Das Gegenstück zu dieser Routine ist ein FORTH-Wort mit dem Namen EXIT . Dabei handelt es sich um ein Wort, das das oberste Element des Returnstacks in den IP bringt und anschließend NEXT anspringt. Damit ist dann die Ausführung der Colon-Definition beendet.

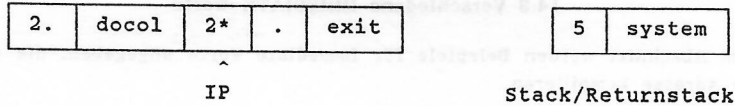
Beispiel :

```
: 2* dup + ;
: 2. 2* . ;
```

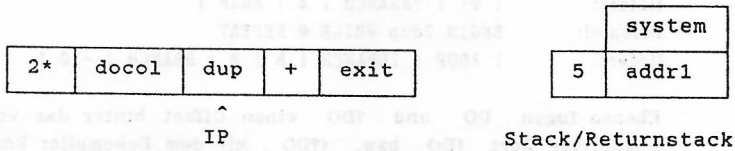
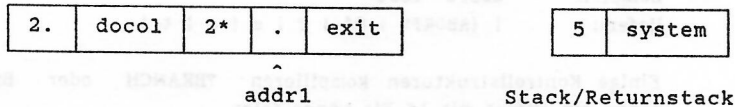
a.) Ein Aufruf von

5 2.

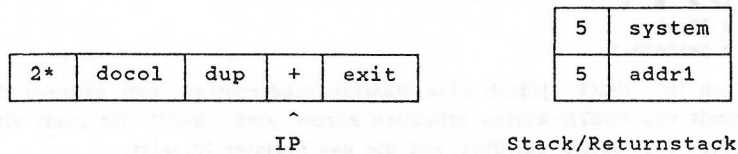
von der Tastatur aus führt zu folgenden Situationen :



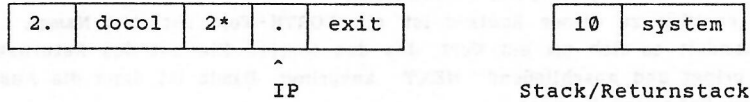
b.) Nach der ersten Ausführung von NEXT bekommt man :



c.) Nochmalige Ausführung von NEXT ergibt:  
( DUP ist ein Wort vom Typ "Code")



- d.) Nach der nächsten Ausführung von NEXT zeigt der IP auf EXIT und nach dem darauf folgenden NEXT wird addr1 wieder in den IP geladen:



- e.) Die Ausführung von . erfolgt analog zu den Schritten b,c und d. Anschließend zeigt IP auf EXIT in 2. . Nach Ausführung von NEXT kehrt das System wieder in den Text-Interpreter zurück. Dessen Rückkehradresse wird durch system ange-deutet. Damit ist die Ausführung von 2. beendet

### 14.3 Verschiedene IMMEDIATE-Worte

In diesem Abschnitt werden Beispiele für immediate Worte angegeben, die mehr als nur eine Adresse kompilieren.

- a.) Zeichenketten (strings), die durch " abgeschlossen werden, liegen als counted strings im Dictionary vor.  
Beispiel: abort" Test"  
liefert: | (ABORT" | 04 | T | e | s | t |
- b.) Einige Kontrollstrukturen kompilieren ?BRANCH oder BRANCH , denen ein Offset mit 16 Bit Länge folgt.  
Beispiel: 0< IF swap THEN  
liefert: | 0< | ?BRANCH | 4 | SWAP |  
Beispiel: BEGIN ?dup WHILE @ REPEAT  
liefert: | ?DUP | ?BRANCH | 8 | @ | BRANCH | -10 |
- c.) Ebenso fügen DO und ?DO einen Offset hinter das von ihnen kompilierte Wort (DO bzw. (?DO . Mit dem Dekompiler können Sie sich eigene Beispiele anschauen.
- d.) Zahlen werden in das Wort LIT , gefolgt von einem 16-Bit-Wert, kompiliert.

Beispiel: [ hex ] 8 1000  
 liefert: ! CLIT ! \$08 ! LIT ! \$1000 !

#### 14.4 Die Does>-Struktur

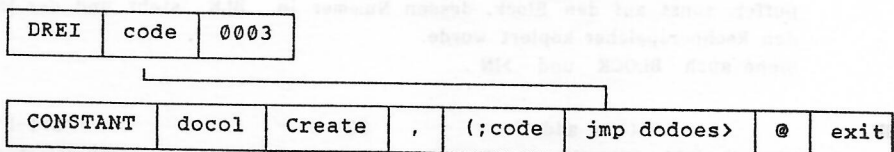
Die Struktur von Worten, die mit CREATE .. DOES> erzeugt wurden, soll anhand eines Beispiels erläutert werden.

Das Beispielprogramm lautet:

```
: Constant ( <name> -- )
  ( -- n )
  Create , Does> @ ;
```

```
3 Constant drei
```

Der erzeugte Code sieht folgendermaßen aus:



Das Wort (;CODE wurde durch DOES> erzeugt. Es setzt das Codefeld des durch CREATE erzeugten Wortes DREI und beendet dann die Ausführung von CONSTANT. Das Codefeld von DREI zeigt anschließend auf jmp dodoes>. Wird DREI später aufgerufen, so wird der zugeordnete Code ausgeführt. Das ist in diesem Fall jmp dodoes>.

dodoes> legt nun die Adresse des Parameterfeldes von DREI auf den Stack und führt die auf jmp dodoes> folgende Sequenz von Worten aus. (Man beachte die Ähnlichkeit zu :-Definitionen). Diese Sequenz besteht aus @ und EXIT. Der Inhalt des Parameterfeldes von DREI wird damit auf den Stack gebracht. Der Aufruf von DREI liefert also tatsächlich 0003.

Statt des jmp dodoes> und der Sequenz von FORTH-Worten kann sich hinter (;CODE auch ausschließlich Maschinencode befinden. Das ist der Fall, wenn wir das Wort ;CODE statt DOES> benutzt hätten. Wird diese Maschinencodesequenz später ausgeführt, so zeigt das W-Register des Adressinterpreters auf das Codefeld des Wortes, dem dieser Code zugeordnet ist. Erhöht man also das W-Register um 2, so zeigt es auf das Parameterfeld des gerade auszuführenden Wortes.

## 14.5 Glossar

|        |                                                                                                                                                                                                                                                                                                                                                                                   |           |         |
|--------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------|---------|
| state  | ( -- addr )                                                                                                                                                                                                                                                                                                                                                                       | 83        |         |
|        | Eine Variable, die den gegenwärtigen Zustand enthält. Der Wert Null zeigt den interpretierenden Zustand an, ein von Null verschiedener Wert den kompilierenden Zustand.                                                                                                                                                                                                           |           |         |
| >in    | ( -- addr )                                                                                                                                                                                                                                                                                                                                                                       | 83        | "to-in" |
|        | Eine Variable, die den Offset auf das gegenwärtige Zeichen im Quelltext enthält.<br>Siehe auch WORD .                                                                                                                                                                                                                                                                             |           |         |
| source | ( -- addr +n )                                                                                                                                                                                                                                                                                                                                                                    |           |         |
|        | liefert Anfang addr und maximale Länge +n des Quelltextes.<br>Ist BLK Null, beziehen sich Anfang und Länge auf den Text-eingabepuffer, sonst auf den Block, dessen Nummer in BLK steht und der in den Rechnerspeicher kopiert wurde.<br>Siehe auch BLOCK und >IN .                                                                                                                |           |         |
| blk    | ( -- addr )                                                                                                                                                                                                                                                                                                                                                                       | 83        | "b-l-k" |
|        | Eine Variable, die die Nummer des gerade als Quelltext interpretierten Blockes enthält. Ist der Wert von BLK Null, so wird der Quelltext vom Texteingabepuffer genommen.                                                                                                                                                                                                          |           |         |
| load   | ( n -- )                                                                                                                                                                                                                                                                                                                                                                          | 83        |         |
|        | Die Inhalte von >IN und BLK , die den gegenwärtigen Quelltext angeben, werden gespeichert. Der Block mit der Nummer n wird dann zum Quelltext gemacht. Der Block wird interpretiert. Die Interpretation wird bei Ende des Blocks abgebrochen, sofern das nicht explizit geschieht. Dann wird der alte Inhalt nach BLK und >IN zurückgebracht.<br>Siehe auch BLK , >IN und BLOCK . |           |         |
| (      | ( -- )                                                                                                                                                                                                                                                                                                                                                                            | 83,1      | "paren" |
|        | ( -- )                                                                                                                                                                                                                                                                                                                                                                            | compiling |         |
|        | wird in der folgenden Art benutzt:<br>( ccc )                                                                                                                                                                                                                                                                                                                                     |           |         |
|        | Die Zeichen ccc, abgeschlossen durch ) , werden als Kommentar betrachtet. Kommentare werden ignoriert. Das Leerzeichen zwischen ( und ccc ist nicht Teil des Kommentars. ( kann im interpretierenden oder kompilierenden Zustand benutzt werden. Fehlt ) , so werden alle Zeichen im Quelltext als Kommentar betrachtet.                                                          |           |         |



`\`                   ( -- )                   I                   "skip-line"  
                   ( -- )                   compiling  
 Ignoriere den auf dieses Wort folgenden Text bis zum Ende der Zeile.  
 Siehe auch `C/L` .

`\`                   ( -- )                   I                   "skip-screen"  
                   ( -- )                   compiling  
 Ignoriere den auf dieses Wort folgenden Text bis zum Ende des Blockes.  
 Siehe auch `B/BLK` .

`\needs`               ( -- )                   "skip-needs"  
 wird in der folgenden Art benutzt:  
                   `\needs <name>`  
 Wird `<name>` in der Suchreihenfolge gefunden, so wird der auf `<name>`  
 folgende Text bis zum Ende der Zeile ignoriert. Wird `<name>` nicht ge-  
 funden, so wird die Interpretation hinter `<name>` fortgesetzt.  
  
 Beispiel:       `\needs Editor 1+ load`  
 lädt den folgenden Block, falls `EDITOR` im Dictionary nicht vorhanden ist.  
  
 Ab der Version 3.81.3 wurde das Konzept um `have` eingeführt, so daß  
 die bedingte Kompilation so eingesetzt wird:  
                   `have Editor not .IF 1+ load .THEN`  
  
 In anderen Quellen finden Sie auch `exists?` .

`+load`               ( n -- )                   "plus-load"  
 ladet den Block, dessen Nummer um `n` höher ist, als die Nummer des  
 gegenwärtig interpretierten Blockes.

`thru`               ( n1 n2 -- )  
 ladet die Blöcke von `n1` bis inklusive `n2`.

`+thru`               ( n1 n2 -- )                   "plus-thru"  
 ladet hintereinander die Blöcke, die `n1..n2` vom gegenwärtigen Block ent-  
 fernt sind.  
 Beispiel `1 2 +thru` lädt die nächsten beiden Blöcke.

`-->`               ( -- )                   I                   "next-block"  
                   ( -- )                   compiling  
 Setze die Interpretation auf dem nächsten Block fort.

- loadfile** ( -- addr)  
addr ist die Adresse einer Variablen, die auf das FORTH-File zeigt, das gerade geladen wird. Diese Variable wird bei Aufruf von **LOAD**, **THRU** usw. auf das aktuelle File gesetzt.
- bye** ( -- )  
Dieses Wort führt **FLUSH** und **EMPTY** aus. Anschließend wird der Monitor des Rechners angesprochen oder eine andere implementationsabhängige Funktion ausgeführt. Der Befehl dient zum Verlassen des volksFORTH.
- cold** ( -- )  
Bewirkt den Kaltstart des Systems. Dabei werden alle nach der letzter Ausführung von **SAVE** definierten Worte entfernt, die Uservariablen auf den Wert gesetzt, den sie bei **SAVE** hatten, die Blockpuffer neu initialisiert, der Bildschirm gelöscht und die Einschaltmeldung  
"volksFORTH-83 rev..."  
ausgegeben. Anschliessend wird **RESTART** ausgeführt.
- restart** ( -- )  
Bewirkt den Warmstart des Systems. Es setzt **'QUIT'**, **ERRORHANDLER** und **'ABORT'** auf ihre normalen Werte und führt **ABORT** aus.
- interpret** ( -- )  
Beginnt die Interpretation des Quelltextes bei dem Zeichen, das durch den Inhalt von **>IN** indiziert wird. **>IN** indiziert relativ zum Anfang des Blockes, dessen Nummer in **BLK** steht. Ist **BLK** gleich Null, so werden Zeichen aus dem Texteingabepuffer interpretiert.
- Auch **INTERPRET** benötigte bisher zur Implementation einen sehr mysteriösen Systempatch. Dank einer Idee von Mike Perry ist auch diese letzte Ecke nun abgeschliffen:  
Das deferred Wort **PARSER** enthält entweder den Code für den Interpreter oder den Compiler (durch **[** und **]** umzuschalten) und Interpret ist nun eine **BEGIN...REPEAT** Schleife, in der das nächste Wort aus dem Quelltext geholt wird. Ist der Quelltext erschöpft, so wird die Schleife verlassen, andernfalls wird **PARSER** aufgerufen und dadurch das Wort entweder interpretiert oder kompiliert.  
Nun ist es auch sehr viel einfacher als vorher, selber eigene Worte zu definieren, die in **PARSER** eingehängt werden. Dies ist immer dann sinnvoll, wenn der Interpreter in einem Anwendungsprogramm anders als der übliche FORTH-Interpreter arbeiten soll.

- prompt** ( -- )  
ist ein deferred Wort, das für die Ausgabe des "ok" verantwortlich ist. Es wurde auch das Wort **QUIT** implementiert.  
Nun ist es möglich, den FORTH-Interpreter auch wie ein "klassisches" Betriebssystem arbeiten zu lassen, in dem eine Meldung nicht nach jeder Aktion hinter der Eingabezeile ausgegeben wird, sondern vor einer Aktion am Anfang der Eingabezeile. Ein entsprechendes Beispiel befindet sich im Quelltext hinter der Definition von **QUIT**.
- parser** siehe INTERPRET.
- word** ( char -- addr ) 83  
erzeugt einen counted String durch Lesen von Zeichen vom Quelltext, bis dieser erschöpft ist oder der Delimiter char auftritt.  
Der Quelltext wird nicht zerstört. Führende Delimiter werden ignoriert. Der gesamte String wird im Speicher beginnend ab Adresse addr als eine Sequenz von Bytes abgelegt. Das erste Byte enthält die Länge des Strings (0..255). Der String wird durch ein Leerzeichen beendet, das nicht in der Längenangabe enthalten ist. Ist der String länger als 255 Zeichen, so ist die Länge undefiniert. War der Quelltext schon erschöpft, als **WORD** aufgerufen wurde, so wird ein String der Länge Null erzeugt. Wird der Delimiter nicht im Quelltext gefunden, so ist der Wert von **>IN** die Länge des Quelltextes. Wird der Delimiter gefunden, so wird **>IN** so verändert, dass **>IN** das Zeichen hinter dem Delimiter indiziert. **#TIB** wird nicht verändert. Der String kann sich oberhalb von **HERE** befinden.
- parse** ( char -- addr +n )  
liefert die Adresse addr und Länge +n des nächsten Strings im Quelltext, der durch den Delimiter char abgeschlossen wird.  
+n ist Null, falls der Quelltext erschöpft oder das erste Zeichen char ist. **>IN** wird verändert.
- name** ( -- addr )  
holt den nächsten String, der durch Leerzeichen eingeschlossen wird, aus dem Quelltext, wandelt ihn in Grossbuchstaben um und hinterlässt die Adresse addr, ab der der String im Speicher steht.  
Siehe auch **WORD**.

find ( addr1 -- addr2 n ) 83  
 addr1 ist die Adresse eines counted string. Der String enthält einen Namen, der in der aktuellen Suchreihenfolge gesucht wird.  
 Wird das Wort nicht gefunden, so ist addr2 = addr1 und n = Null.  
 Wird das Wort gefunden, so ist addr2 dessen Kompilations-adresse und n erfüllt folgende Bedingungen:  
 n ist positiv, wenn das Wort immediate ist, sonst negativ.  
 n ist vom Betrag 2 , falls das Wort restrict ist, sonst vom Betrag 1 .

notfound ( addr -- )  
 Ein deferred Wort, das aufgerufen wird, wenn der Text aus dem Quelltext weder als Name in der Suchreihenfolge gefunden wurde, noch als Zahl interpretiert werden kann.  
 Kann benutzt werden, um eigene Zahl- oder Stringeingabeformate zu erkennen. Ist mit NO.EXTENSIONS vorbesetzt. Dieses Wort bricht die Interpretation des Quelltextes ab und druckt die Fehlermeldung "?" aus.

quit ( -- ) 83  
 Entleert den Returnstack, schaltet den interpretierenden Zustand ein, akzeptiert Eingaben von der aktuellen Eingabeeinheit und beginnt die Interpretation des eingegebenen Textes.

' ( -- addr ) 83 "tick"  
 wird in der Form benutzt:  
 ' <name>  
 addr ist die Kompilationsadresse von <name>. Wird <name> nicht in der Suchreihenfolge gefunden, so wird eine Fehlerbehandlung eingeleitet.

['] ( -- addr ) 83,I,C "bracket-tick"  
 ( -- ) compiling  
 wird in der folgenden Art benutzt:  
 ['] <name>  
 Kompiliert die Kompilationsadresse von <name> als eine Konstante. Wenn die :-definition später ausgeführt wird, so wird addr auf den Stack gebracht. Ein Fehler tritt auf, wenn <name> in der Suchreihenfolge nicht gefunden wird.

compile ( -- ) 83,C  
 Typischerweise in der folgenden Art benutzt:  
 : <name> ... compile <namex> ... ;  
 Wird <name> ausgeführt, so wird die Kompilationsadresse von <namex> zum Dictionary hinzugefügt und nicht ausgeführt. Typisch ist <name> immediate und <namex> nicht immediate.

- [compile] ( -- ) 83,I,C "bracket-compile"  
 ( -- ) compiling  
 Wird in der folgenden Art benutzt:  
 [compile] <name>  
 Erzwingt die Kompilation des folgenden Wortes <name>. Damit ist die Kompilation von immediate-Worten möglich.
- immediate ( -- ) 83  
 Markiert das zuletzt definierte Wort als "immediate", d.h. dieses Wort wird auch im kompilierenden Zustand ausgeführt.
- restrict ( -- )  
 Markiert das zuletzt definierte Wort als "restrict", d.h. dieses Wort kann nicht vom Textinterpreter interpretiert, sondern ausschließlich in anderen Worten kompiliert werden.
- [ ( -- ) 83,I "left-bracket"  
 ( -- ) compiling  
 Schaltet den interpretierenden Zustand ein. Der Quelltext wird sukzessive ausgeführt. Typische Benutzung siehe LITERAL .
- ] ( -- ) 83,I "right-bracket"  
 ( -- ) compiling  
 Schaltet den kompilierenden Zustand ein. Der Text vom Quelltext wird sukzessive kompiliert. Typische Benutzung siehe LITERAL .
- Literal ( -- 16b ) 83,I,C  
 ( 16b -- ) compiling  
 Typisch in der folgenden Art benutzt:  
 [ 16b ] Literal  
 Kompiliert ein systemabhängiges Wort, so daß bei Ausführung 16b auf den Stack gebracht wird.
- ," ( -- ) "comma-quote"  
 Speichert einen counted String im Dictionary ab HERE . Dabei wird die Länge des Strings in dessen erstem Byte, das nicht zur Länge hinzugezählt wird, vermerkt.

```

Ascii          ( -- char )      I
                ( -- )          compiling

```

Wird in der folgenden Art benutzt:

```
Ascii ccc
```

wobei ccc durch ein Leerzeichen beendet wird. char ist der Wert des ersten Zeichens von ccc im benutzten Zeichensatz (gewöhnlich ASCII). Falls sich das System im kompilierenden Zustand befindet, so wird char als Konstante kompiliert. Wird die :-definition später ausgeführt, so liegt char auf dem Stack.

```

Does>          ( -- addr )      83,I,C      "does"
                ( -- )          compiling

```

Definiert das Verhalten des Wortes, das durch ein definierendes Wort erzeugt wurde. Wird in der folgenden Art benutzt:

```

: <typ.name>
  <defining time action>
  <create> <compiletime action>
  Does> <runtime action> ;

```

und später:

```
<typ.name> <instance.name>
```

wobei <create> CREATE ist oder ein anderes Wort, das CREATE ausführt. Zeigt das Ende des Wort-erzeugenden Teils des definierenden Wortes an. Beginnt die Kompilation des Codes, der ausgeführt wird, wenn <instance.name> aufgerufen wird. In diesem Fall ist addr die Parameterfeldadresse von <name>. addr wird auf den Stack gebracht und die Sequenz zwischen DOES> und ; wird ausgeführt.

:Does>

wird benutzt in der Form:

```
Create <name> :Does> ... ;
```

Folgende Definition liegt im volks4TH :DOES> zugrunde:

```
| : (does> here >r [compile] Does> ;
```

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

:DOES> legt das Laufzeit-Verhalten des zuletzt definierten Wortes durch den bis zum ; nachfolgenden Code fest. Das betreffende Wort wurde mit Create oder einem Create benutzenden Wort definiert. Eine Anwendung finden Sie bei den Datentypen.

recursive ( -- ) I,C  
 ( --- ) compiling

Erlaubt die rekursive Kompilation des gerade definierten Wortes in diesem Wort selbst. Ferner kann Code für Fehlerkontrolle erzeugt werden.  
 Siehe auch MYSELF und RECURSE .

Die hier definierten Wörter sind als Teil des FORTH-Systems zu verstehen und sind nicht als separate Wörter zu betrachten.

15.1 Registerführung

Die Registerführung ist in der folgenden Tabelle dargestellt. Die Register sind in der Reihenfolge der Tabelle von links nach rechts zu betrachten.

|          |         |
|----------|---------|
| A ←→ AX  | A ←→ AX |
| A+ ←→ AH |         |

|          |         |
|----------|---------|
| C ←→ CX  | C ←→ CX |
| C+ ←→ CH |         |

Register A und C sind von allgemeinen Benennungen frei.

|          |         |
|----------|---------|
| D ←→ DX  | D ←→ DX |
| D+ ←→ DH |         |

Das D-Register enthält das obere Element des (Distanz-)Stacks.

|          |         |
|----------|---------|
| B ←→ BX  | B ←→ BX |
| B+ ←→ BH |         |

Das B-Register hält den letzten Stack-Pointer.

Die weiteren Prozessor-Register werden wie folgt benannt:

## 15. Der Assembler

Für das volksFORTH stehen zwei Assembler zur Verfügung:

- Ein Assembler entsprechend dem Laxen&Perry F83
- und der eigentliche volks4TH-Assembler.

### 15.1 Registerbelegung

Im Assembler sind FORTH-gemäße Namen für die Register gewählt worden. Dabei ist die Zuordnung zu den Intel-Namen folgendermassen:

|                        |                         |
|------------------------|-------------------------|
| A $\Leftrightarrow$ AX | A- $\Leftrightarrow$ AL |
|                        | A+ $\Leftrightarrow$ AH |

|                        |                         |
|------------------------|-------------------------|
| C $\Leftrightarrow$ CX | C- $\Leftrightarrow$ CL |
|                        | C+ $\Leftrightarrow$ CH |

Register A und C sind zur allgemeinen Benutzung frei.

|                        |                         |
|------------------------|-------------------------|
| D $\Leftrightarrow$ DX | D- $\Leftrightarrow$ DL |
|                        | D+ $\Leftrightarrow$ DH |

Das D Register enthält das oberste Element des (Daten)-Stacks.

|                        |                         |
|------------------------|-------------------------|
| R $\Leftrightarrow$ BX | R- $\Leftrightarrow$ RL |
|                        | R+ $\Leftrightarrow$ RH |

Das B Register hält den Return\_stack\_pointer

Die weiteren Prozessor-Register werden wie folgt genutzt:



U <=> BP User\_area\_pointer  
 S <=> SP Daten\_stack\_pointer  
 I <=> SI Instruction\_pointer  
 W <=> DI Word\_pointer, im allgemeinen zur Benutzung frei.

D: <=> DS E: <=> ES S: <=> SS C: <=> CS

Alle Segmentregister werden beim Booten auf den Wert des Codesegments C: gesetzt und muessen, wenn sie "verstellt" werden, wieder auf C: zurueckgesetzt werden.

Assemblerdefinitionen werden in FORTH mit code eingeleitet und mit der Sequenz Next end-code beendet:

```
Code sp! ( addr -- ) D S mov D pop Next end-code
```

## 15.2 Registeroperationen

### Returnstack

```
Code rp@ ( -- addr ) D push R D mov Next end-code
Code rp! ( addr -- ) D R mov D pop Next end-code
```

```
Code rdrop R inc R inc Next end-code restrict
Code >r ( 16b -- ) R dec R dec D R ) mov D pop
Next end-code restrict
Code r> ( -- 16b ) D push R ) D mov R inc R inc
Next end-code restrict
```

```
Code r@ ( -- 16b ) D push R ) D mov Next end-code
```

```
Code execute ( acf -- ) D W mov D pop
W ) jmp end-code
```

```
Code perform ( addr -- ) D W mov D pop
W ) W mov W ) jmp end-code
```

```
\ : perform ( addr -- ) @ execute ;
```

### Segmentregister

```
Code ds@ ( -- addr ) D push D: D mov Next end-code
```

## 15.3 Besonderheiten im volksFORTH83

Nachfolgend sind einige interessante Besonderheiten beschrieben:

NEXT ist ein Makro, sog. in-line code; wenn NEXT kompiliert wird, werden die in NEXT enthaltenen Mnemonics in die Definition eingetragen:

```
Assembler also definitions
: Next   lods  A W xchg  W ) jmp
        there tnext-link @ T , H tnext-link ! ;
```

Das Wort EI hinterläßt durch here die Adresse von NEXT auf dem Stack. Diese Adresse wird in das Codefeld von NOOP geschrieben, so daß ein explizites NEXT unnötig wird.

```
| Code ei   sti   here   Next   end-code

Code noop   here 2- !   end-code
```

```
Create recover Assembler
      R dec  R dec  I R ) mov  I pop  Next
end-code
```

RECOVER soll nie ausgeführt werden, nur die Adresse dieser Routine ist für den TargetCompiler von Interesse:

```
: ;c:  @ T recover # call ] end-code H ;
```

In ähnlicher Form findet sich dann die endgültige Form von ;c: für das Target-System wieder:

```
: ;c:  recover # call last off end-code @ ] ;
```

#### 15.4 Glossar

**assembler** ( -- )  
ist das Vokabular, das die Assembler-Worte enthält. Die Ausführung dieses Wortes nimmt ASSEMBLER als context in die Suchreihenfolge mit auf.

**code <name>** ( -- )  
leitet eine Assembler-Definition ein und schaltet die Suchreihenfolge auf das Vokabular Assembler .  
Als Beispiel:  
Code sp@ ( -- addr ) D push S D mov Next end-code

**end-code** ( -- )  
beendet eine Assembler-Definition.

```

Label <name>      ( -- )
                  ( -- addr )
legt eine benannte Marke an und liefert eine Adresse zurück:

Code exit
Label >exit  R ) I mov  R inc  R inc  Next  end-code

Code unnest  >exit here 2- !  end-code
Code ?exit  ( flag -- )
  D D or  D pop  >exit 0= ?]  [[ Next  end-code
Code 0=exit ( flag -- )
  D D or  D pop  >exit 0= not ?] ]]  end-code

\ : ?exit ( flag -- )  IF rdrop THEN ;

```

```

>label <name>    ( -- )
                  ( -- addr )
kann als temporäre Konstante betrachtet werden, die sich mit clear
löschen läßt. Ein von >label zurückgelieferter Wert wird als Literal
kompiliert.
Beispiel:        Assembler
                  nop 5555 # jmp          here 2- >label >cold
                  nop 5555 # jmp          here 2- >label >restart

```

```

;code1          ( -- )
beendet den high level Teil eines defining words und leitet dessen
Assembler-Teil ein. Wird benutzt in der Form:
: <name> <Create> ... ;code ... end-code

```

```

;c:              ( -- )
leitet den high level Teil einer Assembler-Definition ein. Das Wort wird
typisch im Handhaben einer Fehlersituation eingesetzt, da dann
Geschwindigkeit keine Rolle mehr spielt. Als Beispiel:
: stackfull ( -- )  depth $20 > Abort" tight stack"
  reveal last? IF dup heap? IF name> ELSE 4- THEN
    (forget
      THEN
      true Abort" dictionary full" ;

Code ?stack  u' dp U D) A mov  S A sub  CS
  ?[ $100 # A add  CS ?[ ;c: stackfull ; Assembler ]? ]?
  u' s0 U D) A mov  A inc  A inc  S A sub
  CS not ?[ Next ]? ;c: true Abort" stack empty" ;

\ : ?stack  sp@ here - $100 u< IF stackfull THEN
\          sp@ s0 @ u> Abort" stack empty" ;

```

## 15.5 Kontrollstrukturen im Assembler

?[ ]?

benutzt in der Form, wobei cc ein condition code ist:

cc IF ... THEN

Als Beispiel:

```
Code dup ( 16b -- 16b 16b ) D push Next end-code
Code ?dup ( 16b -- 16b 16b / false )
D D or 0= not ?[ D push ]? Next end-code
```

?[ ][ ]?

benutzt in der Form:

cc IF ... ELSE ... THEN

[[ ]]

benutzt in der Form:

BEGIN ... REPEAT

Als Beispiel:

```
Code 1+ ( n1 -- n2 ) [[ D inc Next
Code 2+ ( n1 -- n2 ) [[ D inc swap ]]
Code 3+ ( n1 -- n2 ) [[ D inc swap ]]
Code 4+ ( n1 -- n2 ) [[ D inc swap ]]
! Code 6+ ( n1 -- n2 ) D inc D inc ]] end-code
```

```
Code 1- ( n1 -- n2 ) [[ D dec Next
```

```
Code 2- ( n1 -- n2 ) [[ D dec swap ]]
```

[[ cc ]]?

benutzt in der Form:

BEGIN ... cc UNTIL

[[ cc ?[ ][ ]]?

benutzt in der Form:

BEGIN ... cc WHILE ... REPEAT

## 15.6 Beispiele aus dem volksFORTH

pop

```
Code drop ( 16b -- ) D pop Next end-code
```

```

Code ! ( 16b addr -- ) D W mov W ) pop D pop
Next end-code

Code +! ( 16b addr -- )
D W mov A pop A W ) add D pop Next end-code

push
Code 2dup ( 32b -- 32b 32b )
S W mov D push W ) push Next end-code

\ : 2dup ( 32b -- 32b 32b ) over over ;

Code over ( 16b1 16b2 -- 16b1 16b2 16b1 )
A D xchg D pop D push A push Next end-code

mov
Code @ ( addr -- 16b ) D W mov W ) D mov
Next end-code

Code c@ ( addr -- 8b )
D W mov W ) D- mov 0 # D+ mov Next end-code

Code c! ( 16b addr -- )
D W mov A pop A- W ) mov D pop Next end-code

xchg
Code flip ( 16b1 -- 16b1' ) D- D+ xchg Next end-code

\ : flip ( 16b1 -- 16b1' ) $100 um* or ;

neg
Code negate ( n1 -- n2 ) D neg Next end-code
\ : negate ( n1 -- n2 ) not 1+ ;

and
Code and ( 16b1 16b2 -- 16b3 )
A pop A D and Next end-code

or
Code or ( 16b1 16b2 -- 16b3 )
A pop A D or Next end-code
\ : or ( 16b1 16b2 -- 16b3 ) not swap not and not ;

xor
Code ctoggle ( 8b addr -- )
D W mov A pop A- W ) xor D pop Next end-code

```

```

Code xor ( 16b1 16b2 -- 16b3 )
  A pop A D xor Next end-code

\ : ctoggle ( 8b addr -- ) under c@ xor swap c! ;

                                add
Code + ( n1 n2 -- n3 ) A pop A D add Next end-code

                                sub
Code - ( n1 n2 -- n3 )
  A pop D A sub A D xchg Next end-code
\ : - ( n1 n2 -- n3 ) negate + ;

                                com
Code not ( 16b1 -- 16b2 ) D com Next end-code

                                inc
Code nip ( 16b1 16b2 -- 16b2 ) S inc S inc Next end-code
\ : nip swap drop ;

Code 2drop ( 32b -- ) S inc S inc D pop Next end-code
\ : 2drop ( 32b -- ) drop drop ;

                                sal
Code pick ( n -- 16b.n )
  D sal D W mov S W add W ) D mov Next end-code
\ : pick ( n -- 16b.n ) 1+ 2* sp@ + @ ;

                                std rep byte movs cld
Code roll ( n -- )
  A I xchg D sal D C mov D I mov S I add
  I ) D mov I W mov I dec W inc std
  rep byte movs cld A I xchg S inc S inc Next
end-code
\ : roll ( n -- )
\   dup >r pick sp@ dup 2+ r> 1+ 2* cmove> drop ;

Code -roll ( n -- ) A I xchg D sal D C mov
  S W mov D pop S I mov S dec S dec
  rep byte movs D W ) mov D pop A I xchg Next
end-code

\ : -roll ( n -- ) >r dup sp@ dup 2+
\   dup 2+ swap r@ 2* cmove r> 1+ 2* + ! ;

```