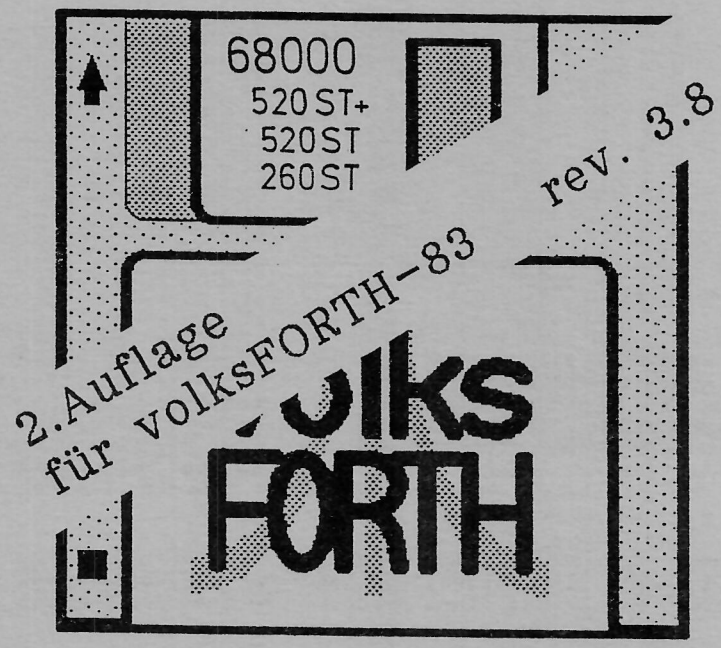
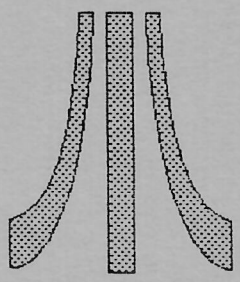


HANDBUCH



für

ATARI

260ST, 520ST, 520ST+

Handbuch zum **volksFORTH83 rev 3.8**
2. Auflage 18.12.1986

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

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

INHALTSVERZEICHNIS

Teil 3 - Glossare

Teil 4 - Definition der verwendeten Begriffe

Anhang

Prolog : über das volksFORTH83

volksFORTH83 ist eine Sprache, die in verschiedener Hinsicht ungewöhnlich ist. Einen ersten Eindruck vom volksFORTH83 und von unserem Stolz darüber soll dieser Prolog vermitteln.

volksFORTH83 braucht nicht geknackt oder geklaut zu werden. Im Gegenteil, wir hoffen, daß viele Leute das volksFORTH83 möglichst schnell bekommen und ihrerseits weitergeben.

Warum stellen wir dieses System als "public domain" zur Verfügung ?

Die Verbreitung, die die Sprache FORTH gefunden hat, war wesentlich an die Existenz von figFORTH geknüpft. Auch figFORTH ist ein public domain Programm, d.h. es darf weitergegeben und kopiert werden. Trotzdem haben sich bedauerlicherweise verschiedene Anbieter die einfache Adaption des figFORTH an verschiedene Rechner sehr teuer bezahlen lassen. Das im Jahr 1979 erschienene figFORTH ist heute nicht mehr so aktuell, weil mit der weiteren Verbreitung von Forth eine Fülle von eleganten Konzepten entstanden sind, die z.T. im Forth-Standard von 1983 Eingang gefunden haben. Daraufhin wurde von Laxen und Perry das F83 geschrieben und als Public Domain verbreitet.
Dieses freie 83-Standard-Forth mit zahlreichen Utilities ist recht komplex, es wird auch nicht mit Handbuch geliefert.
Insbesondere gibt es keine Version für C64- und Apple-Computer. Der C64 spielt jedoch in Deutschland eine große Rolle.

Wir haben ein neues Forth für verschiedene Rechner entwickelt. Das Ergebnis ist das volksFORTH83, eines der besten Forth-Systeme, die es gibt. Es wurde zunächst für den C64 geschrieben. Nach Erscheinen der Rechner der Atari ST-Serie entschlossen wir uns, auch für sie ein volksFORTH83 zu entwickeln. Die erste ausgelieferte Version 3.7 war, was Editor und Massenspeicher betraf, noch stark an den C64 angelehnt. Sie enthielt jedoch schon einen verbesserten Tracer, die GEM-Bibliothek und die anderen Tools für den ST. Der nächste Schritt bestand in der Einbindung der Betriebssystem-Files. Nun konnten Quelltexte auch vom Desktop und mit anderen Utilities verarbeitet werden. Mit der jetzt verteilten Version 3.8 ist das volksFORTH vollständig: Der GEM-unterstützte Editor erleichert auch Anfängern das Programmieren, und der Code ist relokatibel geworden, sodaß Applikationen sehr einfach zu realisieren sind. Die Probleme der früheren Versionen mit Accessories etc. entfallen ebenfalls. Schließlich wurde die Dokumentation erheblich erweitert, und für die wichtigsten Files liegen Shadowscreens vor. Der Editor dürfte mit seinen fast 50 kByte Kommentar das am besten dokumentierte umfangreiche GEM-Programm sein, das frei erhältlich ist. Gleichzeitig führt er exemplarisch vor, wie man GEM in Forth programmiert.

Warum soll man in volksFORTH83 programmieren ?

Das volksFORTH83 ist ein ausgesprochen leistungsfähiges und kompaktes Werkzeug. Durch residente Runtime-library, Compiler, Editor und Debugger sind die ermüdenden ECLG-Zyklen ("Edit, Compile, Link and Go") überflüssig; der Code wird Modul für

Modul entwickelt, kompiliert und getestet. Der integrierte Debugger ist die perfekte Testumgebung für Worte; es gibt keine riesigen Hexdumps oder Assemblerlistings, die kaum Ähnlichkeit mit dem Quelltext haben.
Ein anderer wichtiger Aspekt ist das Multitasking. So wie man ein Programm in einzelne, unabhängige Module oder Worte aufteilt, so sollte man es auch in einzelne, unabhängige Prozesse aufteilen können. Das ist in den meisten Sprachen nicht möglich, auch mit den sog. Desk-Accessories nicht. Das volksFORTH83 besitzt einen einfachen, aber leistungsfähigen Multitasker, der auch für Aufgaben eingesetzt werden kann, die über einen Druckerspooler hinausgehen.
Schließlich besitzt das volksFORTH83 noch eine Fülle von Details, die andere Forthsysteme nicht haben. Es benutzt an vielen Stellen Vektoren und sog. deferred Worte, die eine einfache Umgestaltung des Systems für verschiedene Gerätekonfigurationen ermöglichen. Es besitzt einen Heap (für "namenlose" Worte oder für Code, der nur zeitweilig benötigt wird). Der Blockmechanismus ist so schnell, daß er auch sinnvoll für die Bearbeitung großer Datenmengen, die in Files vorliegen, eingesetzt werden kann.
Die mit dem System gelieferten Disketten umfassen Tracer, Decompiler, Multitasker, Assembler, Editor, Disassembler, GEM-Bibliothek, Graphikdemos, Printerinterface ...

Das volksFORTH83 erzeugt, verglichen mit anderen Forthsystemen, relativ schnellen Code, der aber langsamer als der anderer Compilersprachen ist. Alle anderen Interpreter hängt es aber mühelos ab, auch sogenannte "schnelle" BASICs.

Kurz: Das System stellt in einer Vielzahl von Details einen Fortschritt dar!

Noch einmal : Ihr dürft und sollt diese Disks an eure Freunde weitergeben.
Aber wenn sich jemand erdreistet, damit einen Riesenreibach zu machen, dann werden wir ihn bis an das Ende der Welt und seiner Tage verfolgen !

Wir behalten uns die kommerzielle Verwertung des volksFORTH83 vor!

Mit diesem Handbuch soll die Unterstützung des volksFORTH83 noch nicht zuende sein. Die **Forth Gesellschaft e.v.**, ein gemeinnütziger Verein, bietet dafür die Plattform. Sie gibt die **"VIERTE DIMENSION** - Forth Magazin" heraus und betreibt den **FORTH-Tree**, einen ungewöhnlichen, aber sehr leistungsfähigen "Mailbox"-ähnlichen Rechner.

Forth Gesellschaft e.V.
Friedensalle 92
2000 HAMBURG 50
040 - 390 42 04 (Dienstags 18-20 Uhr mündlich, sonst Tree !)

Schickt uns bitte auch eure Meinung zum volksFORTH, Programme, die fertig oder halbfertig sind, Ideen zum volksFORTH, Artikel, die in der Presse erschienen sind (schreibt selbst welche !) , kurz: Schickt uns alles, was entfernt mit dem volksFORTH zu tun hat. Und natürlich : Anregungen, Ergänzungen und Hinweise auf evtl. Fehler im Handbuch und volksFORTH83.

Wir sind nämlich, trotz der vielen verteilten Systeme, des Lobes und der Kritik, die uns erreichte, etwas enttäuscht über die geringe "forthige" Resonanz !

Wenn euch das volksFORTH gefällt, so erwarten wir eine Spende von ca DM 20,-, denn die Entwicklung des volksFORTH sowie des Handbuchs war teuer und der Preis, den wir verlangen, deckt nur die Unkosten.

Für die Autoren des volksFORTH :

Bernd Pennemann
Steilshooper Str. 46
2000 Hamburg 60

Über dieses Handbuch

Dieses Handbuch ist kein Lehrbuch. Für Anfänger sei auf das immer noch beste Lehrbuch "Starting Forth" von Leo Brodie verwiesen. Das Buch ist auf Deutsch unter dem Titel "Programmieren in Forth" im Hanser Verlag erschienen. Dieses Handbuch enthält auch eine Liste der Abweichungen des volksFORTH83 von dem im Brodie besprochenen Forth. Diese Liste ist erforderlich, weil Brodie sein Buch vor 1983 geschrieben hat, als es den 83-Standard noch nicht gab.
Es wird vorausgesetzt, daß der Leser dieses Handbuchs "Starting Forth" oder ein ähnliches Lehrbuch kennt. Darüber hinausgehende Kenntnisse sind jedoch nicht unbedingt erforderlich.

Dieses Handbuch dokumentiert die benutzten Konzepte und Worte.

Der erste Teil führt den Leser in die Benutzung des volksFORTH83 auf dem Atari ST ein.
Der zweite Teil erklärt wie dieses Forth-System aufgebaut ist und wie es funktioniert. Dieser Teil soll dem Anfänger die Möglichkeit geben, sich in diese umfangreiche Sprache vollständig einzuarbeiten, also auch zu wissen, wie sie funktioniert. Dem Fortgeschrittenen soll er die Übertragung auf andere Rechner oder Prozessoren erleichtern.

Der dritte Teil enthält, nach Sachgruppen geordnet, die Worte des volksFORTH83. Soweit sie durch den 83er-Standard festgelegt sind, wurden die Definitionen aus dem Standard übersetzt.

Der vierte Teil enthält Definitionen der benutzten Begriffe. Sie entstammen direkt dem Standard und wurden angepaßt.

Schließlich befinden sich im Anhang Erläuterungen zu Editor, Assembler, GEM, Tools, Fehlermeldungen und Abweichungen von anderen Forth-Systemen.

Namen von Forthworten werden im Text groß geschrieben.

Teil 1 Getting started ...

Um volksFORTH83 vollständig nutzen zu können, sollten Sie dieses Handbuch sorgfältig studieren. Damit Sie möglichst schnell einsteigen können, werden wir in diesem Kapitel beschreiben,

- wie man mit den Disketten umgeht
- wie man das System startet
- wie man eine fertige Applikation erstellt.
- wie man ein eigenes Arbeitssystem zusammenstellt

Umgang mit den Disketten

Zu Ihrem Handbuch haben Sie drei Disketten erhalten. Fertigen Sie auf jeden Fall Sicherheitskopien von diesen Disketten an. Die Gefahr eines Datenverlustes ist groß, da FORTH Ihnen in jeder Hinsicht freie Hand läßt - auch beim versehentlichen Löschen Ihrer Systemdisketten !! Auf der einen Diskette befinden sich die Programme 4TH.PRG, und FORTHKER.PRG sowie ein Ordner mit Textdokumenten und das Demoprogramm 'Super copy'. Diese Diskette wird im Folgenden Systemdiskette genannt.

4TH.PRG ist das normale Arbeitssystem, FORTHKER.PRG eine Minimalversion, die nur den Sprachkern enthält. Damit können Sie eigene FORTH-Versionen z.B. mit einem veränderten Editor oder anderer Graphic zusammenstellen und mit **SAVESYSTEM** als fertiges System abspeichern. In der gleichen Art können Sie auch fertige Applikationen herstellen, denen man ihre 'FORTH-Abstammung' nicht mehr ansieht (ein Beispiel dazu ist SUPER COPY auf Ihrer Systemdiskette.) 4TH.PRG ist ein komplettes Arbeitsssystem mit residentem Fileinterface, Editor, Assembler, Tools, usw.

Die beiden anderen Disketten enthalten alle weiteren Quelltexte des Systems. Ein Verzeichnis erhalten Sie mit **FILES** (analog zu **WORDS**). Bei dem File FORTH_83.SCR handelt es sich um den Quelltext des Sprachkerns. Eben dieser Quelltext ist mit einem Target-Compiler kompiliert worden und entspricht exakt dem FORTHKER.PRG . Sie können sich also den Compiler ansehen, wenn Sie wissen wollen, wie das volksFORTH83 funktioniert. Im File STARTUP.SCR gibt es einen Loadscreen, der alle Teile kompiliert, die zu 4TH.PRG gehören. Mit diesem Loadscreen ist aus FORTHKER.PRG das File 4TH.PRG zusammengestellt worden.

Erster Einstieg

Legen Sie die Systemdiskette in Laufwerk B, die Quelltextdiskette Files 2 in Laufwerk A. (Beim Arbeiten mit einem Laufwerk werden Sie an den entsprechenden Stellen zum Wechseln der Diskette aufgefordert.) Laden Sie 4TH.PRG wie üblich durch Anklicken mit der Maus. volksFORTH83 meldet sich nun mit einer Einschaltmeldung, die die Versionsnummer enthält.

Geben Sie jetzt ein:

```
use tutorial.scr    <Return>
1 l                 <Return>
```

Auf die Aufforderung 'Enter your ID :' antworten Sie zunächst mit <Return>. Sie befinden sich jetzt im Editor und können sich diesen und die folgenden Screens ansehen und dabei den ersten Umgang mit dem Editor erlernen. Der Editor läuft komplett unter einer GEM-Umgebung, sodaß sich Anleitungen erübrigen. Trotzdem finden Sie einige Erklärungen im Anhang. Probieren Sie ein wenig herum und verlassen Sie dann den Editor, z.B. durch Drücken von <Esc>.

Sie können sich jetzt z.B. die Graphik-Demos ansehen. Sie befinden sich auf derselben Diskette. Zusätzlich wird allerdings noch der Assembler von Diskette Files 1 gebraucht. Legen Sie also diese anstelle der Systemdiskette in Laufwerk B. Geben Sie ein

```
include demo.scr  <Return>
```

Das Laufwerk läuft an, Sie sehen zuerst, welche Screens gerade kompiliert werden und dann einige Graphic-Beispiele. Nach jedem Bild können Sie mit einer beliebigen Taste weitermachen oder mit <ESC> abbrechen.

Jetzt gehts aber richtig los ....

Wir wollen jetzt einige eigene Worte kompilieren und dazu ein neues Quelltextfile anlegen. Legen Sie dazu eine neue Diskette in Laufwerk A ein. Geben Sie dann ein:

```
makefile first.scr   2 more   <Return>
```

Sie erzeugen damit ein File namens FIRST.SCR mit einer Länge von 2 Blöcken (2048 Byte), bestehend aus Screen 0 und 1. Um einige Definitionen auf Screen 1 zu schreiben, rufen Sie den Editor auf mit

```
1 l   <Return>
```

In der obersten Zeile eines Screens sollte eine Kurzbeschreibung dessen, was der Screen enthält, stehen. (Mit dem Wort INDEX erhält man so ein Inhaltsverzeichnis des Files.) Schreiben Sie also, beginnend in der linken oberen Ecke :

```
\ Mein erstes FORTH-Programm
```

Der \ (Backslash) sorgt dafür, daß diese Zeile nicht kompiliert wird, sondern als Kommentar aufgefaßt wird. Vielleicht ist Ihnen auch bereits aufgefallen, daß bei unseren Quelltexten in der oberen rechten Ecke Datum und Autor der letzten Änderung vermerkt sind. Diese Kennung erzeugt der Editor automatisch, wenn beim ersten Aufruf eine ID angegeben wurde. Sie können Ihre ID jedoch auch nachträglich setzen, wenn Sie die Tastenkombination CTRL-G drücken oder den entsprechenden Menüpunkt "GET-ID" anwählen.

Die zweite Zeile des Screens bleibt frei, in der dritten definieren wir:

```
: test      ." Das ist mein erstes 'Programm'." ;
```

Verlassen Sie nun den Editor mit CTRL-S. Dabei wird der - inzwischen ja veränderte - Screen auf Diskette zurückgeschrieben. Nun rufen wir den Compiler auf mit

```
1 load   <Return>
```

Mit für 'C'- oder Pascal-Programmierer unvorstellbarer Geschwindigkeit wird unser Mini-Programm kompiliert und steht jetzt zur Ausführung bereit. Geben Sie einmal WORDS ein, dann werden Sie feststellen, daß Ihr neues Wort TEST ganz oben im Dictionary steht (drücken Sie die <ESC>-Taste, um die Ausgabe von WORDS abzubrechen oder irgendeine andere Taste, um sie anzuhalten). Um das Ergebnis unseres ersten Programmierversuchs zu überprüfen, geben wir nun ein:

```
test    <Return>
```

und siehe da, es tut sich etwas. Schöner wäre es allerdings, wenn der Satz auf einer neuen Zeile beginnen würde. Um dies zu ändern, werfen wir erst einmal das alte Wort TEST weg.

```
forget test
```

Nun rufen wir erneut den Editor auf mit V . Vor dem String fügen wir ein CR ein. Das Wort sieht dann so aus :

```
: test   cr ." Das ist mein erstes 'Programm'." ;
```

Dann kompilieren wir wie gehabt. Unsere Änderung erweist sich als erfolgreich, und Sie haben gelernt, wie einfach in FORTH das Schreiben, Austesten und Ändern von Programmteilen ist.

Herstellung einer Applikation

Wir wollen unser 'Programm' nun als eigenständige Applikation abspeichern. Dazu erweitern wir es zunächst ein klein wenig (Editor mit V aufrufen.) FÜgen Sie nun noch folgende Definition in einer neuen Zeile hinzu:

```
: runalone      test  key drop  bye ;
```

RUNALONE führt zuerst TEST aus, wartet dann auf eine Taste und kehrt zum Desktop zurück. Kompilieren Sie nun erneut, führen Sie RUNALONE aber nicht aus, sonst würden wir ja FORTH verlassen. Das Problem besteht vielmehr darin, das System so abzuspeichern, daß es gleich nach dem Laden RUNALONE ausführt und sonst gar nichts.

volksFORTH83 ist an zwei Stellen für solche Zwecke vorbereitet. In den Worten COLD und RESTART befinden sich zwei 'deferred words' namens 'COLD bzw. 'RESTART , die im Normalfall nichts tun, vom Anwender aber nachträglich verändert werden können. Wir benutzen hier 'COLD, um auch schon die Startmeldung zu unterbinden.

Geben Sie also ein

```
' runalone Is 'cold     <Return>
```

und speichern Sie das Ganze mit

```
savesystem myprog.prg
```

auf Diskette zurück. Laden Sie dann MYPROG.PRG durch das übliche anklicken. Sie haben Ihre erste Applikation erstellt !

Etwas enttäuschend ist es aber schon. Das angeblich so kompakte FORTH benötigt über 40 kByte, um einen lächerlichen String auszugeben ?? Da stimmt doch etwas nicht. Natürlich, wir haben ja eine Reihe Systemteile mit abgespeichert, vom Fileinterface über den Assembler, die halbe GEM-Bibliothek, den Editor usw., die für unser Programm überhaupt nicht benötigt werden.

Um dieses und ähnliche Probleme zu lösen, gibt es das File FORTHKER.PRG. Dieses Programm enthält nur den Systemkern und das Fileinterface. Laden Sie also FORTHKER.PRG und kompilieren Sie Ihre Applikation mit

```
include first.scr
```

Dann wie gehabt, RUNALONE in 'COLD patchen und das System auf Diskette zurückspeichern. Sie haben jetzt eine verhältnismäßig kompakte Version vorliegen. Natürlich ließe sich auch diese noch erheblich kürzen, aber dafür bräuchten Sie einen Target-Compiler, mit dem Sie nur noch die wirklich benötigten Systemteile selektiv aus dem Quelltext zusammenstellen könnten. Mit der beschriebenen Methode lassen sich aber auch größere Programme kompilieren und als Stand-alone-Applikationen abspeichern.

Für Fortgeschrittene

Man kann allerdings im obigen Beispiel ohne Target-Compiler auskommen, wenn man auf das File-Interface, das ca. 4 KByte belegt, verzichten kann. Dazu muß man zunächst eine FORTH-Version ohne Fileinterface herstellen. Laden Sie dazu FORTHKER.PRG und geben Sie ein:

```
' blk@   Is makeview     <Return>
' noop   Is custom-remove     <Return>
' STr/w  Is r/w     <Return>
  direct     <Return>
```

Damit werden die 'deferred words', die das Fileinterface umgestellt hat, wieder auf Direktmodus umgeschaltet und das Fileinterface selbst abgeschaltet.

Das erste Wort im Fileinterface oberhalb von SAVESYSTEM ist DOS. Dieses Wort kann man aber nicht mit FORGET vergessen, da es sich im 'geschützten Bereich' befindet. Probieren Sie es ruhig aus; Sie erhalten die Meldung: 'DOS is protected'. Für solche Fälle gibt es das Wort (FORGET .

```
' Dos >name 4- (forget    save     <Return>
```

Damit haben Sie ein System 'ohne alles'. Speichern Sie es mit

```
savesystem minimal.prg     <Return>
```

auf Diskette zurück.

Natürlich können wir unser Testfile nicht mehr mit INCLUDE laden, denn es gibt ja kein Fileinterface mehr. Stattdessen suchen wir uns die entsprechenden Blocks im Direktzugriff. Laden Sie also wieder 4TH.PRG, schalten Sie das Fileinterface mit

```
direct
```

ab, und geben Sie ein:

```
1 500 index
```

(500 ist zwar sicher mehr als reichlich, aber INDEX stoppt sowieso automatisch beim letzten Disketten- oder Fileblock.) Unser Testscreen dürfte ziemlich weit hinten stehen, aber wir bekommen so auch gleich einen Überblick über den Diskettenin-halt. Merken Sie sich die Nummer n des Testscreens, und verlassen Sie FORTH. Laden Sie dann MINIMAL.PRG und kompilieren Sie Ihren Screen ohne Angabe eines Filenamens mit <n> LOAD.
Ganz trickreiche Programmierer sind sicherlich auch in der Lage, mit diesem System das Fileinterface auf den Heap zu laden, ähnlich wie das beim Assembler vorgeführt wird. Dann kann man es zum Zusammenstellen der Applikation benutzen, SAVESYSTEM speichert es aber nicht mit ab.

Herstellen eines eigenen Systems

Das File 4TH.PRG ist als Arbeitsversion gedacht. Es enthält alle wichtigen Systemteile wie Editor, Printer-Interface, Tools, GEM-Graphic, Decompiler, Tracer usw. Sollte Ihnen die Zusammenstellung nicht gefallen, können Sie sich jederzeit ein Ihren speziellen Wünschen angepaßtes System zusammenstellen. Schlüssel dazu ist der Loadscreen im File STARTUP.SCR der Diskette Files 1. Sie können dort Systemteile, die Sie nicht benötigen, mit dem Backslash '\' wegkommentieren oder die entsprechenden Zeilen ganz löschen. Ebenso können Sie natürlich dem Loadscreen eigene Files hinzufügen.

Entspricht der Loadscreen Ihren Wünschen, speichern Sie ihn mit CTRL-S zurück, und verlassen Sie das System mit BYE. Laden Sie nun File FORTHKER.PRG. Geben Sie dann ein:

```
include startup.scr    <Return>
```

Ist das System fertig kompiliert, legen Sie die Systemdiskette ein und schreiben:

```
savesystem 4th.prg   <Return>
```

Damit wird Ihr altes File überschrieben (Sicherheitskopie !!!), sodaß Sie beim nächsten Laden von 4TH.PRG Ihr eigenes System erhalten. Natürlich können Sie 'Ihr' System auch unter einem anderen Namen mit SAVESYSTEM abspeichern.

Ebenso können Sie Systemvoreinstellungen ändern. Unsere Arbeitsversion arbeitet - voreingestellt - neuerdings im dezimalen Zahlensystem. Natürlich können Sie mit

```
hex    <Return>
```

auf Hexadezimalsystem umstellen; wir halten das für sehr viel sinnvoller, weil vor allem Speicheradressen im Dezimalsystem kaum etwas aussagen (oder wissen Sie, ob Speicherstelle 978584 im Bildschirmspeicher liegt oder nicht ?) . Wollen Sie bereits unmittelbar nach dem Laden im Hexadezimalsystem arbeiten, können Sie sich dies mit SAVESYSTEM, wie oben beschrieben, abspeichern.

Im Übrigen empfehlen wir bei allen Zahlen über 9 dringend die Benutzung der sogenannten Präfixe $ für Hexadezimal-, & für Dezimal- und % für Binärzahlen. Man vermeidet so, daß irgendwelche Files nicht - oder noch schlimmer, falsch - kompiliert werden, weil man gerade im anderen Zahlensystem ist. Außerdem ist es möglich, hexadezimale und dezimale Zahlen beliebig zu kombinieren, je nachdem, was gerade sinnvoller ist. In unseren Quelltexten finden Sie genug entsprechende Beispiele.

Wenn Sie nur ein Laufwerk zur Verfügung haben, sollten Sie den voreingestellten Suchpfad - es wird erst Laufwerk A, dann Laufwerk B durchsucht - ändern. Wenn nur auf Laufwerk A gesucht werden soll, geben Sie folgende Sequenz ein

```
path ;  a:
```

und speichern das System mit SAVESYSTEM auf Diskette zurück.

Umkopieren des Systems auf doppelseitige Disketten

Wenn Sie über doppelseitige Laufwerke verfügen, können Sie das System auf doppelseitige Disketten umkopieren. Der Vorteil liegt darin, daß die VIEW-Funktion natürlich nur dann funktioniert, wenn die entsprechenden Quelltextfiles verfügbar sind. Und auf ein oder zwei doppelseitigen Disketten läßt sich schon eine Reihe Files zur Verfügung halten.

Ebenso ist es natürlich möglich, alle Teile des Systems auf einer Festplatte abzulegen. Sie sollten dafür einen eigenen Ordner einrichten und PATH und DIR entsprechend einstellen.

Auch die Arbeit mit einer RAM-Disk ist prinzipiell möglich, allerdings nicht sehr zu empfehlen. FORTH ist sehr maschinennah und Systemabstürze daher vor allem zu Anfang nicht so ganz auszuschließen. Wir empfehlen stattdessen die Verwendung des Files RAMDISK.SCR . Damit werden sehr viele Blockbuffer im RAM - außerhalb des FORTH-Systems - eingerichtet, was die Kompilationszeiten erheblich beschleunigt. Trotzdem ist große Datensicherheit vorhanden, weil alle geänderten Blocks immer auf Diskette zurückgeschrieben werden, wenn man den Editor mit CTRL-S verläßt. Die Gefahr eines Datenverlustes bei Systemabsturz ist so nahezu ausgeschlossen.

Ausdrucken der Quelltexte

Sicher ist es sinnvoll, die Quelltexte des Systems auszudrucken, um Beispiele für den Umgang mit volksFORTH83 zu sehen. Inzwischen ist ein großer Teil der Quelltexte mit Kommentarscreens versehen, was - wie wir hoffen - gerade für den Einsteiger eine große Hilfe darstellt. Welche Files sich im Einzelnen auf Ihren Disketten befinden und ob sie Kommentarscreens enthalten, steht im File README.DOC.

Zunächst müssen Sie das Printerinterface hinzuladen, falls es nicht schon vorhanden ist. Reagiert Ihr volksFORTH83 auf die Eingabe von PRINTER mit einem kräftigen Haeh? , so ist das Printerinterface nicht vorhanden. Legen Sie die entsprechende Diskette ein und laden Sie es mit :

```
include printer.scr
```

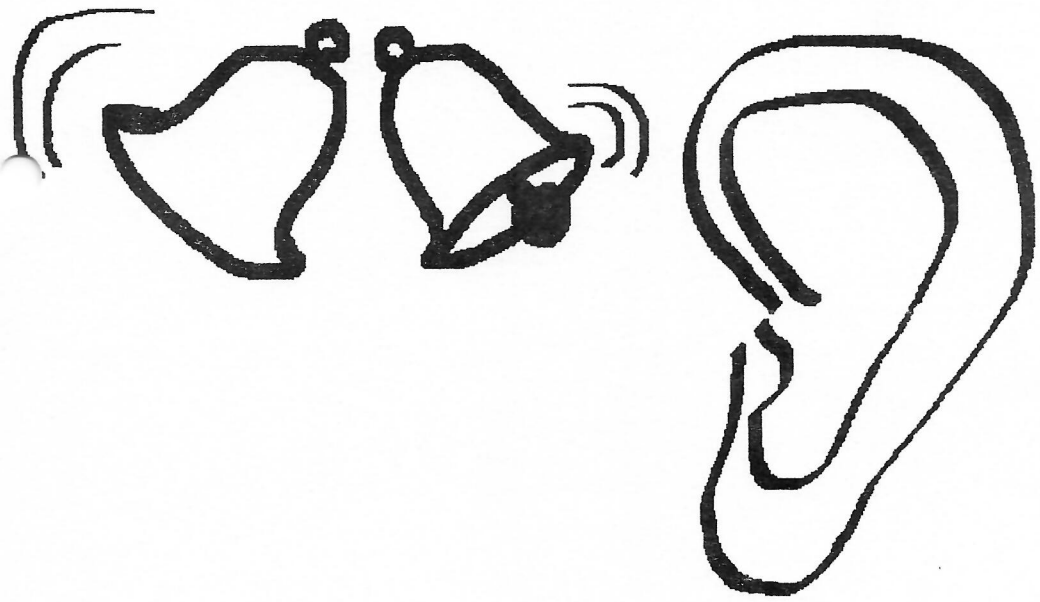
Zum Ausdrucken von Files mit Shadowscreens gibt man ein :

```
USE <filename>    LISTING
```

Für Files ohne Shadowscreens benutzt man :

```
USE <filename>    PRINTALL
```

Näheres zum Drucker-Interface finden Sie im Kapitel 4 des zweiten Teils.



Erläuterungen

Teil 2

1) Dictionarystruktur des volksFORTH83

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

1.1) Struktur der Worte

Die Forthworte 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:

a) "block"
Der Nummer des Blocks, in dem das Wort definiert wurde (siehe auch VIEW im Glossar).

b) "link"
Einer Adresse (Zeiger) , die auf das "Linkfeld" des nächsten Wortes zeigt.

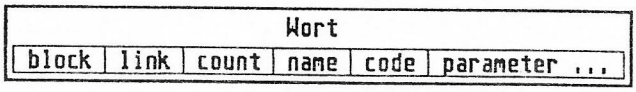
c) "count"
Die Länge des Namens dieses Wortes und drei Markierungsbits.

d) "name"
Der Name selbst.

e) "code"
Einer Adresse (Zeiger) , die auf den Maschinencode zeigt, der bei Aufruf dieses Wortes ausgeführt wird. Die Adresse dieses Feldes heißt Kompilationsadresse.

f) "parameter"
Das Parameterfeld.

Ein Wort sieht dann so aus :



1.1.1) Countfeld

Das count-Feld 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 im Glossar).

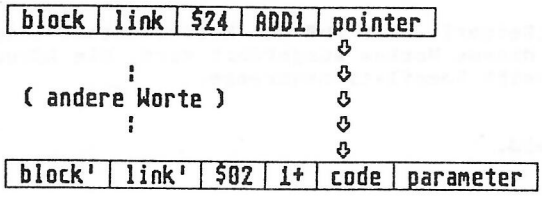
Ist das restrict-Bit gesetzt, so kann das Wort nicht durch Eingabe von der Tastatur ausgeführt, sondern nur in anderen Worten kompiliert werden. Gibt man es dennoch im interpretierenden Zustand ein, so erscheint die Fehlermeldung "compile only" (siehe auch RESTRICT im Glossar).

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) :



(Bei allen folgenden Bildern werden die Felder block link count nicht mehr extra dargestellt.)

1.1.2) Name

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

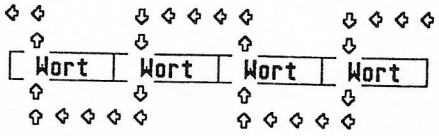
Alle anderen Worte werden klein geschrieben. Beispiele sind: dup cold base

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. Das volksFORTH unterscheidet nicht zwischen Groß- und Kleinschreibung, sodaß Sie die Worte beliebig eingeben können.

1.1.3) Link

Über das link-Feld 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.



1.1.4) Block

Das block-Feld 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.

1.1.5) Code

Jedes Wort weist auf ein Stück Maschinencode. Die Adresse dieses Code-Stücks ist im Code-Feld 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.

1.1.6) Parameter

Das Parameterfeld enthält Daten, die vom Typ des Wortes abhängen.
Beispiele :

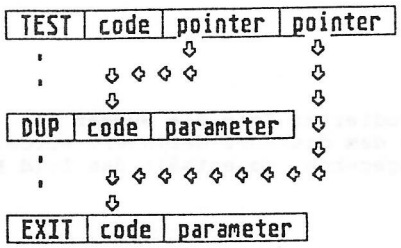
a) Typ "Constant"
Hier enthält das Parameterfeld des Wortes den Wert der Konstanten. Der dem Wort zugeordnete Code liest den Inhalt des Parameterfeldes aus und legt ihn auf den Stack.

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.
EXIT 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.

1.2) 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.

1.2.1) Die Suchreihenfolge

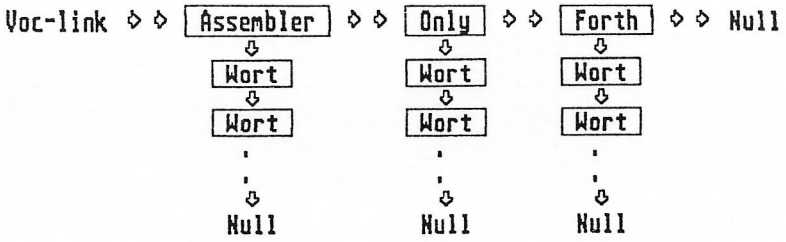
Die Suchreihenfolge gibt an, in welcher Reihenfolge die verschiedenen Vokabulare nach einem Wort durchsucht werden.
Sie besteht aus zwei Teilen, dem auswechselbaren und dem festen Teil. Der auswechselbare Teil enthält genau ein Vokabular. Dies wird zuerst durchsucht. Wird ein Vokabular durch Eingeben 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 befördert 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.

Beispiele :

Eingabe :	ORDER ergibt dann :
Onlyforth	FORTH FORTH ONLY FORTH
Editor also	EDITOR EDITOR FORTH ONLY FORTH
Assembler	ASSEMBLER EDITOR FORTH ONLY FORTH
definitions Forth	FORTH EDITOR FORTH ONLY ASSEMBLER
: test ;	ASSEMBLER EDITOR FORTH ONLY ASSEMBLER

1.2.2) Struktur des Dictionaries

Der Inhalt eines Vokabulars besteht aus einer Liste von Worten, die durch ihre link-Felder 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:



2) Die Ausführung von Forth-Worten

Der geringe Platzbedarf übersetzter Forth-Worte rührt wesentlich von der Existenz des Adressinterpreters her.
Wie aus dem Kapitel 1.1.6 Absatz c) ersichtlich, 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.

2.1) Aufbau des Adressinterpreters beim volksFORTH83

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

IP W SP RP

a) IP ist der Instruktionszeiger (englisch : Instruction Pointer). Er zeigt auf die nächste auszuführende Instruktion. Das ist beim volksFORTH83 die Speicherzelle, die die Kompilationsadresse des nächsten auszuführenden Wortes enthält.

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

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

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

2.2) 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 schreiben wir hier diesen Teil in High Level:

```
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 die Einsprungadresse einer Routine, die diejenige Instruktion ausführt, auf die das Register IP zeigt. Ein Wort wird ausgeführt, indem der Code, auf den die Kompilationsadresse zeigt, als Maschinencode angesprungen 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.

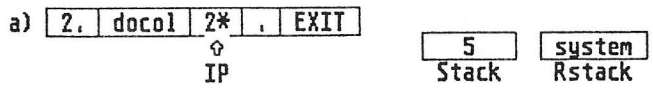
Diese Routine gibt es wirklich, sie heißt "docol" und ihre Adresse steht im Codefeld jeder :-Definition. Das Gegenstück zu dieser Routine ist ein Forth-Wort mit dem Namen EXIT . Dabei handelt es sich um ein Wort, das das oberste Element des Returnstacks in den IP bringt und anschließend NEXT anspringt. Damit ist dann die Ausführung der Colon-definition beendet.

Beipiel :

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

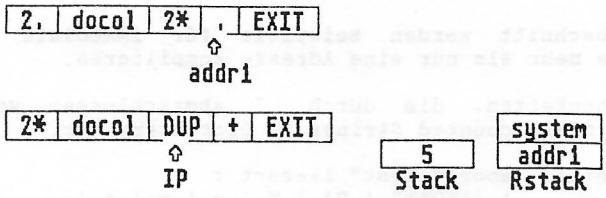
Ein Aufruf von
```
    5 2.
```
von der Tastatur aus führt zu folgenden Situationen :

a)



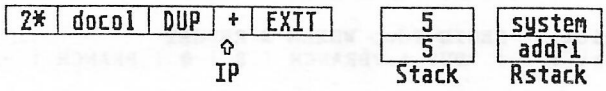
Nach der ersten Ausführung von NEXT bekommt man :

b)



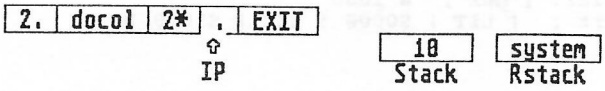
Nochmalige Ausführung von NEXT ergibt :
(DUP ist ein Wort vom Typ "Code")

c)



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

d)



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 Textinterpreter zurück. Dessen Rückkehradresse wird durch system angedeutet. Damit ist die Ausführung von 2. beendet.

2.3) Verschiedene Immediate Worte

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

a) Zeichenketten, 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 : | LIT | $0008 | LIT | $1000 | .

3) Die Does>-Struktur

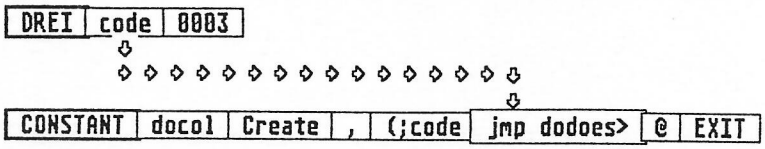
Die Struktur von Worten, die mit CREATE .. DOES> erzeugt wurden, soll anhand eines Beispiels erläutert werden.

Das Beispielprogramm lautet :

```
: Constant     Create ,  Does>  @  ;
3 Constant drei
```

Der erzeugte Code sieht folgendermaßen aus:

Der jump-Befehl ist symbolisch gemeint. In Wirklichkeit wird eine andere Adressierungsart benutzt !



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 Forthworten 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.

4) Vektoren und Deferred Worte

Das volksFORTH83 besitzt eine Reihe von Strukturen, die dazu dienen, das Verhalten des Systems zu ändern.

4.1) deferred Worte

Sie werden durch das Wort DEFER erzeugt. 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 die Phrase:

```
' <name> Is <namex>
```

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

Wird <namex> ausgeführt bevor es mit IS initialisiert wurde, so erscheint die Meldung "Crash" auf dem Bildschirm.

Anwendungsmöglichkeiten von deferred Worten :

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. Teil 1, "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.

4.2) >interpret

Dieses Wort ist ein spezielles deferred Wort, das für die Umschaltung des Textinterpreters in den Kompilationszustand und zurück benutzt wird.

4.3) Variablen

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 .

4.4) Vektoren

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 Parameterfeldadresse 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:

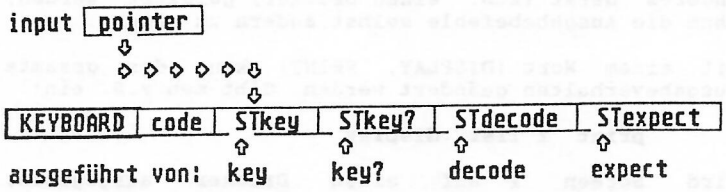
```
: key?   ( -- c )
     input @  2+  perform ;
```

(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
       STkey STkey? STdecode STexpect ;
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.
Mit der Input/Output-Vektorisierung kann man z.B. mit einem Schlag die Eingabe von der Tastatur auf ein Modem umschalten.

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.

Die Druckeranpassung

Alle Ausgabeworte (EMIT, TYPE, SPACE etc.) sind im volksFORTH-83 vektorisiert, d.h. bei ihrem Aufruf wird die Codefeldadresse des zugehörigen Befehls aus einer Tabelle entnommen und ausgeführt. Im System enthalten ist eine Tabelle mit Namen DISPLAY, die für die Ausgabe auf das Bildschirmterminal sorgt. Dieses Verfahren bietet entscheidende Vorteile:

- Mit einer neuen Tabelle können alle Ausgaben auf ein anderes Gerät (z.B. einen Drucker) geleitet werden, ohne die Ausgabebefehle selbst ändern zu müssen.

- Mit einem Wort (DISPLAY, PRINT) kann das gesamte Ausgabeverhalten geändert werden. Gibt man z.B. ein:

  ```
  print 1 list display
  ```

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

Eine neue Tabelle wird mit dem Wort OUTPUT: erzeugt. (Die Definition können Sie mit VIEW OUTPUT: nachsehen.) OUTPUT: erwartet eine Liste von Ausgabeworten, die mit ; abgeschlossen werden muß. Beispiel:

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

Damit wird eine neue Tabelle mit dem Namen >PRINTER angelegt. Beim späteren Aufruf von >PRINTER wird die Adresse dieser Tabelle in die Uservariable OUTPUT geschrieben. Ab sofort führt EMIT ein PEMIT aus, TYPE ein PTYPE usw. Die Reihenfolge der Worte nach OUTPUT: userEMIT userCR userTYPE userDEL userPAGE userAT userAT? muß unbedingt eingehalten werden.

Der folgende Quelltext enthält die Anpassung für einen Epson-Drucker RX80/FX80 oder kompatible am Atari ST. Zusätzlich zu den reinen Ausgaberoutinen (Screen 7) sind eine Reihe nützlicher Worte enthalten, mit denen die Druckersteuerung sehr komfortabel vorgenommen werden kann.

Arbeiten Sie mit einem Drucker, der andere Steuercodes als Epson verwendet, müssen Sie die Screens 2 und 4 bis 6 anpassen. Bei IBM-kompatiblen Druckern ist es meist damit getan, daß die Umlautwandlung (Screen 6) weggelassen wird. Dies erreicht man, indem man in Zeile 1 einen doppelten Backslash (\\) setzt.

Im Arbeitssystem ist das Printerinterface bereits enthalten. Müssen Sie Änderungen vornehmen, können Sie entweder jedesmal Ihr geändertes Printerinterface mit

```
include printer.scr
```

dazuladen oder sich ein neues Arbeitssystem zusammenstellen wie im Kapitel 'Getting started ...' beschrieben. Sie können natürlich auch den Loadscreen in seiner jetzigen Fassung benutzen und das Printer-Interface 'von Hand' nachladen.

Im zweiten Teil des Printer-Interface (Screen 9 ff.) sind einige Worte zur Ausgabe eines formatierten Listings enthalten. PTHRU druckt einen Bereich von Screens, PRINTALL ein ganzes File, jeweils 6 Screens auf einer DIN A4 Seite in komprimierter Schrift. Ganz ähnlich arbeiten die Worte DOCUMENT und LISTING , jedoch wird bei diesen Worten neben einen Quelltextscreen der zugehörige Shadowscreen gedruckt. (Mehr über das Shadowscreen-Konzept erfahren Sie im Kapitel über den Editor.) Man erhält so ein übersichtliches Listing eines Files mit ausführlichen Kommentaren. Es empfiehlt sich daher, alle Files, die Shadowscreens enthalten, einmal mit LISTING auszudrucken.

An dieser Stelle folgt das Listing des Files PRINTER.SCR.

```
  PRINTER.SCR Scr 0 Dr 0
 0 \\                  *** Printer-Interface ***            10oct86we
 1
 2 Dieses File enthält das Printer-Interface. Die Definitionen für
 3 die Druckersteuerung müssen ggf. an Ihren Drucker angepaßt wer-
 4 den.
 5
 6 PRINT  lenkt alle Ausgabeworte auf den Drucker um, mit  DISPLAY
 7 wird wieder auf dem Bildschirm ausgegeben.
 8
 9 Zum Ausdrucken der Quelltexte gibt es die Worte
10
11    pthru      ( from to -- )   druckt Screen from bis to
12    document   ( from to -- )  wie pthru, aber mit Shadow-Screens
13    printall   ( -- )    wie pthru, aber druckt das ganze File
14    listing    ( -- )    wie document, aber für das ganze File
15
```

```
  PRINTER.SCR Scr 1 Dr 0
 0 \ Printer Interface Epson RX80\FX80                    21oct86we
 1
 2 Onlyforth
 3
 4 \needs file?            ' noop  ¦ Alias file?
 5 \needs capacity         ' blk/drv Alias capacity
 6
 7 Vocabulary Printer   Printer definitions also
 8
 9  1 &13 +thru
10
11 Onlyforth  \ clear
12
13
14
15
```

```
  PRINTER.SCR Scr 2 Dr 0
 0 \ Printer  p! and controls                             18nov86we
 1
 2 ' bcostat ¦ Alias ready?   ' 0 ¦ Alias printer
 3
 4 : p!  ( n -- )
 5   BEGIN  pause  printer ready?  UNTIL  printer bconout ;
 6
 7
 8 ¦ : ctrl:  ( 8b -- )   Create c,   does>  ( -- )   c@ p! ;
 9
10  07   ctrl: BEL      $7F ¦ ctrl: DEL       $0D ¦ ctrl: RET
11 $1B ¦ ctrl: ESC      $0A   ctrl: LF        $0C   ctrl: FF
12
13
14
15
```

```
  PRINTER.SCR Scr 15 Dr 0
 0 \\                *** Printer-Interface ***          13oct86we
 1
 2 Eingestellt ist das Druckerinterface auf Epson und kompatible
 3  Drucker. Die Steuersequenzen auf den Screens 2, 4 und 5 müssen
 4  gegebenenfalls auf Ihren Drucker angepaßt werden. Bei uns gab
 5  es mit verschiedenen Druckern allerdings keine Probleme, da
 6  sich inzwischen die meisten Druckerhersteller an die Epson-
 7  Steuercodes halten.
 8
 9 Arbeiten Sie mit einem IBM-kompatiblen Drucker, muß die Umlaut-
10  wandlung auf Screen 6 wegkommentiert werden.
11
12 Zusätzliche 'exotische' Steuersequenzen können nach dem Muster
13  auf den Screens 4 und 5 jederzeit eingebaut werden.
14
15
```

```
  PRINTER.SCR Scr 16 Dr 0
 0 \ Printer Interface Epson RX80                        13oct86we
 1
 2 setzt order auf  FORTH FORTH ONLY   FORTH
 3
 4 falls das Fileinterface nicht im System ist, werden die ent-
 5  sprechenden Worte ersetzt.
 6
 7 Printer-Worte erhalten ein eigenes Vocabulary.
 8
 9
10
11
12
13
14
15
```

```
  PRINTER.SCR Scr 17 Dr 0
 0 \ Printer  p! and controls                            10oct86we
 1
 2 nur aus stilistischen Gründen. Das Folgende liest sich besser.
 3
 4 Hauptausgabewort; gibt ein Zeichen auf den Drucker aus. Es wird
 5  gewartet, bis der Drucker bereit ist. (PAUSE für Multitasking)
 6
 7
 8 gibt Steuerzeichen an Drucker
 9
10 Steuerzeichen für Drucker. Gegebenenfalls anpassen!
11
12
13
14
15
```

```
  PRINTER.SCR Scr 3 Dr 0
 0 \ Printer controls                                        09sep86re
 1
 2 | : esc:  ( 8b -- )   Create c,   does>  ( -- )   ESC c@ p! ;
 3
 4 | : esc2  ( 8b0 8b1 -- )   ESC p! p! ;
 5
 6 | : on:  ( 8b -- )  Create c,  does>  ( -- )  ESC c@ p!  1 p! ;
 7
 8 | : off: ( 8b -- )  Create c,  does>  ( -- )  ESC c@ p!  0 p! ;
 9
10
11
12
13
14
15
```

```
  PRINTER.SCR Scr 4 Dr 0
 0 \ Printer Escapes Epson RX-80/FX-80                       12sep86re
 1
 2 $0F | ctrl: (+17cpi              $12 | ctrl: (-17cpi
 3
 4 Ascii P | esc: (+10cpi           Ascii M | esc: (+12cpi
 5 Ascii 0   esc: 1/8"              Ascii 1   esc: 1/10"
 6 Ascii 2   esc: 1/6"              Ascii T   esc: suoff
 7 Ascii N   esc: +jump             Ascii O   esc: -jump
 8 Ascii G   esc: +dark             Ascii H   esc: -dark
 9 \ Ascii 4   esc: +cursive          Ascii 5   esc: -cursive
10
11 Ascii W   on:  +wide             Ascii W   off: -wide
12 Ascii -   on:  +under            Ascii -   off: -under
13 Ascii S   on:  sub               Ascii S   off: super
14
15
```

```
  PRINTER.SCR Scr 5 Dr 0
 0 \ Printer Escapes Epson RX-80/FX-80                       12sep86re
 1
 2 : 10cpi   (-17cpi (+10cpi ;      ' 10cpi   Alias pica
 3 : 12cpi   (-17cpi (+12cpi ;      ' 12cpi   Alias elite
 4 : 17cpi   (+10cpi (+17cpi ;      ' 17cpi   Alias small
 5
 6 : lines  ( #.of.lines -- )   Ascii C esc2 ;
 7
 8 : "long  ( inches -- )   0 lines p! ;
 9
10 : american   0 Ascii R esc2 ;
11
12 : german     2 Ascii R esc2 ;
13
14 : normal     10cpi  american  suoff  1/6"  &12 "long  RET ;
15
```

```
  PRINTER.SCR Scr 18 Dr 0
 0 \ Printer controls                                        10oct86we
 1
 2 gibt Escape-Sequenzen an den Drucker aus.
 3
 4 gibt Escape und zwei Zeichen aus.
 5
 6 gibt Escape, ein Zeichen und eine 1 an den Drucker aus.
 7
 8 gibt Escape, ein Zeichen und eine 0 an den Drucker aus.
 9
10
11
12
13
14
15
```

```
  PRINTER.SCR Scr 19 Dr 0
 0 \ Printer Escapes Epson RX-80/FX-80                       10oct86we
 1
 2 setzt bzw. löscht Ausgabe komprimierter Schrift.
 3
 4 setzt Zeichenbreite auf 10 bzw. 12 cpi.
 5 Zeilenabstand in Zoll.
 6                                   schaltet Super- und Subscript ab
 7 Perforation überspringen ein- und ausschalten.
 8 Es folgen die Steuercodes für Fettdruck, Kursivschrift, Breit-
 9  schrift, Unterstreichen, Subscript und Superscript.
10  Diese müssen ggf. an Ihren Drucker angepaßt werden.
11  Selbstverständlich können auch weitere Fähigkeiten Ihres Druk-
12  kers genutzt werden wie Proportionalschrift, NLQ etc.
13
14
15
```

```
  PRINTER.SCR Scr 20 Dr 0
 0 \ Printer Escapes Epson RX-80/FX-80                       13oct86we
 1
 2 Hier wird die Zeichenbreite eingestellt. Dazu kann man sowohl
 3  Worte mit der Anzahl der characters per inch (cpi) als auch
 4  pica, elite und small benutzen.
 5
 6 setzt Anzahl der Zeilen pro Seite; Einstellung:
 7  &66 lines      oder     &12 "long
 8
 9
10 schaltet auf amerikanischen Zeichensatz.
11
12 schaltet auf deutschen Zeichensatz.
13
14 Voreinstellung des Druckers auf 'normale' Werte; wird beim
15  Einschalten mit PRINT ausgeführt.
```

```
  PRINTER.SCR Scr 6 Dr 0
 0 \ Umlaute                                                14oct86we
 1
 2 ¦ Create DIN
 3 Ascii ä c,       Ascii ö c,       Ascii ü c,       Ascii ß c,
 4 Ascii Ä c,       Ascii Ö c,       Ascii Ü c,       Ascii § c,
 5
 6 ¦ Create AMI
 7 Ascii { c,       Ascii ¦ c,       Ascii } c,       Ascii ~ c,
 8 Ascii [ c,       Ascii \ c,       Ascii ] c,       Ascii @ c,
 9
10 here AMI - ¦ Constant tablen
11
12 ¦ : p!  ( char -- )   dup $80 < IF  p! exit  THEN
13    tablen 0 DO  dup  I DIN + c@  =
14                 IF  drop  I AMI + c@  LEAVE  THEN  LOOP
15    german p! american ;
```

```
  PRINTER.SCR Scr 7 Dr 0
 0 \ Printer Output                                         12sep86re
 1
 2 ¦ Variable pcol   pcol off      ¦ Variable prow   prow off
 3
 4 ¦ : pemit  ( 8b -- )     p!  1 pcol +! ;
 5 ¦ : pcr  ( -- )          RET LF  1 prow +!  pcol off ;
 6 ¦ : pdel  ( -- )         DEL  pcol @ 1- 0 max pcol ! ;
 7 ¦ : ppage  ( -- )        FF  prow off  pcol off ;
 8 ¦ : pat  ( row col -- )   over  prow @ <  IF  ppage  THEN
 9      swap  prow @ -  0 ?DO  pcr  LOOP
10      dup  pcol @ <  IF  RET  pcol off  THEN  pcol @ - spaces ;
11 ¦ : pat?  ( -- row col )   prow @  pcol @ ;
12 ¦ : ptype  ( adr len -- )
13      dup pcol +!  bounds ?DO  I c@ p!  LOOP ;
14
15
```

```
  PRINTER.SCR Scr 8 Dr 0
 0 \ Printer output                                         18nov86we
 1
 2 Output: >printer   pemit pcr ptype pdel ppage pat pat? ;
 3
 4 Forth definitions
 5
 6 : print   >printer  normal ;
 7
 8
 9
10
11
12
13
14
15
```

```
  PRINTER.SCR Scr 21 Dr 0
 0 \ Umlaute                                                bp 12oct86
 1
 2 Auf diesem Screen werden die Umlaute aus dem IBM-(ATARI)-Zeichen
 3  satz in DIN-Umlaute aus dem deutschen Zeichensatz gewandelt.
 4
 5 Wenn Sie einen IBM-kompatiblen Drucker benutzen, kann dieser
 6  Screen mit \\ in der ersten Zeile wegkommentiert werden.
 7
 8
 9
10
11
12 p! wird neu definiert. Daher brauchen die folgenden Worte p!
13  nicht zu ändern, egal, ob mit oder ohne Umlautwandlung gearbei-
14  tet wird.
15
```

```
  PRINTER.SCR Scr 22 Dr 0
 0 \ Printer Output                                       10oct86we
 1
 2 aktuelle Druckerzeile und -spalte.
 3 Routinen zur Druckerausgabe     entspricht Befehl
 4 ein Zeichen auf Drucker         emit
 5 CR und LF auf Drucker           cr
 6 ein Zeichen löschen (?!)        del
 7 neue Seite                      page
 8 Drucker auf Zeile und Spalte    at
 9  positionieren; wenn nötig,
10  neue Seite.
11 Position feststellen           at?
12 Zeichenkette ausgeben          type
13
14 Damit sind die Worte für eine eigene Output-Struktur vorhanden.
15
```

```
  PRINTER.SCR Scr 23 Dr 0
 0 \ Printer output                                       10oct86we
 1
 2 erzeugt die Output-Tabelle >printer.
 3
 4 Die folgenden Worte sind von FORTH aus zugänglich.
 5
 6 schaltet Ausgabe auf Printer um. (Zurückschalten mit DISPLAY)
 7
 8
 9
10
11
12
13
14
15
```

```
  PRINTER.SCR Scr 9 Dr 0
 0 \ Variables and Setup                                    bp 12oct86
 1
 2 Printer definitions
 3
 4 ' 0 ¦ Alias logo
 5
 6 ¦ : header  ( pageno -- )
 7     12cpi  +dark  ."   volksFORTH-83    FORTH-Gesellschaft eV "
 8     -dark  17cpi  ." (c) 1985/86 we/bp/re/ks  "  12cpi +dark
 9     file?  -dark  17cpi ."  Seite "  . ;
10
11
12
13
14
15
```

```
  PRINTER.SCR Scr 10 Dr 0
 0 \ Print 2 screens across on a page                    26oct86we
 1
 2 ¦ : 2lines  ( scr#1 scr#2 line# -- )
 3     cr  dup 2 .r space  c/l * >r
 4     pad  c/l 2* 1+  bl fill  swap
 5     block  r@ +  pad          c/l cmove
 6     block  r> +  pad c/l + 1+  c/l cmove
 7     pad  c/l 2* 1+  -trailing  type ;
 8
 9 ¦ : 2screens  ( scr#1 scr#2 -- )
10     cr cr  &30 spaces
11     +wide +dark over 4 .r  &28 spaces  dup 4 .r  -wide -dark
12     cr  1/s 0 DO  2dup  I 2lines  LOOP  2drop ;
13
14
15
```

```
  PRINTER.SCR Scr 11 Dr 0
 0 \ print 6 screens on a page                           18sep86we
 1
 2 ¦ : pageprint  ( last+1 first pageno -- )
 3     header  2dup - 1+  2/  dup 0
 4     ?DO  >r  2dup under r@ + >
 5          IF  dup r@ +  ELSE  logo  THEN  2screens 1+ r>  LOOP
 6     drop 2drop  page ;
 7
 8 ¦ : >shadow   ( n1 -- n2 )
 9     capacity 2/  2dup < IF + ELSE - THEN ;
10
11 ¦ : shadowprint  ( last+1 first pageno -- )
12     header  2dup -  0
13     ?DO  dup dup >shadow  2screens  1+  LOOP
14     2drop page ;
15
```

```
  PRINTER.SCR Scr 24 Dr 0
 0 \ Variables and Setup                                    10oct86we
 1
 2 Diese Worte sind nur im Printer-Vokabular enthalten.
 3
 4 Dieser Screen wird gedruckt, wenn es nichts besseres gibt.
 5
 6 Druckt die Überschrift der Seite pageno.
 7
 8
 9
10
11
12
13
14
15
```

```
  PRINTER.SCR Scr 25 Dr 0
 0 \ Print 2 screens across on a page                       10oct86we
 1
 2 druckt nebeneinander die Zeilen line# der beiden Screens.
 3  Die komplette Druck-Zeile wird erst in PAD aufbereitet.
 4
 5
 6
 7
 8
 9 formatierte Ausgabe der beiden Screens nebeneinander
10  mit fettgedruckten Screennummern. Druck erfolgt mit 17cpi, also
11  in komprimierter Schrift.
12
13
14
15
```

```
  PRINTER.SCR Scr 26 Dr 0
 0 \ print 6 screens on a page                              10oct86we
 1
 2 gibt eine Seite aus. Anordnung der Screens auf der Seite:  1 4
 3  Wenn weniger als 6 Screens vorhanden sind, werden         2 5
 4  Lücken auf der rechten Seite mit dem Logo-Screen (0)      3 6
 5  aufgefüllt.
 6
 7
 8 berechnet zu Screen n1 den Shadowscreen n2 (Kommentarscreen wie
 9  dieser hier).
10
11 wie pageprint, aber anstelle der Screens 4, 5 und 6 werden die
12  Shadowscreens zu 1, 2 und 3 gedruckt.
13
14
15
```

```
  PRINTER.SCR Scr 12 Dr 0
 0 \ Printing without Shadows                                  b11nov86we
 1
 2 Forth definitions  also
 3
 4 ! Variable printersem    0 printersem !    \ for multitasking
 5
 6 : pthru  ( first last -- )       2 arguments
 7    printersem lock   output push  print
 8    1+  capacity umin  swap  2dup -  6 /mod  swap 0<> -  0
 9    ?DO  2dup 6 + min  over  I 1+  pageprint  6 +  LOOP
10    2drop  printersem unlock ;
11
12 : printall  ( -- )   0 capacity 1- pthru ;
13
14
15
```

```
  PRINTER.SCR Scr 13 Dr 0
 0 \ Printing with Shadows                                      bp 12oct86
 1
 2 : document  ( first last -- )
 3    printersem lock   output push   print
 4    1+  capacity 2/ umin  swap  2dup -  3 /mod  swap 0<> -  0
 5    ?DO  2dup 3+ min  over  I 1+  shadowprint  3+  LOOP
 6    2drop   printersem unlock ;
 7
 8 : listing  ( -- )   0 capacity 2/ 1- document ;
 9
10
11
12
13
14
15
```

```
  PRINTER.SCR Scr 14 Dr 0
 0 \ Printerspool                                               14oct86we
 1
 2 \needs Task        \\
 3
 4 $100 $200 Task spooler
 5
 6 : spool'   ( -- )    \ reads word
 7    '  isfile@  offset @  base @   spooler  depth 1-  6 min  pass
 8    base !  offset !  isfile !  execute
 9    true abort" SPOOL' ready for next job!" stop ;
10
11
12
13
14
15
```

```
  PRINTER.SCR Scr 27 Dr 0
 0 \ Printing without Shadows                                  b22oct86we
 1
 2 Die folgenden Definitionen stellen das Benutzer-Interface dar.
 3  Daher sollen sie in FORTH gefunden werden.
 4
 5 PRINTERSEM ist ein Semaphor für das Multitasking, der den Zugang
 6  auf den Drucker für die einzelnen Tasks regelt.
 7
 8 PTHRU gibt die Screens von  from  bis  to  aus.
 9  Ausgabegerät merken und Drucker einschalten. Multitasking wird,
10  sofern es den Drucker betrifft, gesperrt.
11  Die Screens werden mit pageprint ausgegeben.
12
13
14 wie oben, jedoch wird das komplette File gedruckt.
15
```

```
  PRINTER.SCR Scr 28 Dr 0
 0 \ Printing with Shadows                                     10oct86we
 1
 2 wie pthru, aber mit Shadowscreens.
 3
 4
 5
 6
 7
 8 wie printall, aber mit Shadowscreens.
 9
10
11
12
13
14
15
```

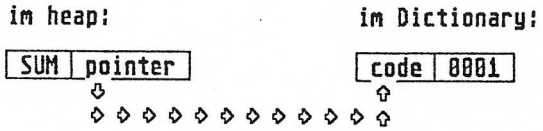
```
  PRINTER.SCR Scr 29 Dr 0
 0 \ Printerspool                                              10oct86we
 1
 2 Falls der Multitasker nicht vorhanden ist, wird abgebrochen.
 3
 4 Der Arbeitsbereich der Task wird erzeugt.
 5
 6 Mit diesem Wort wird das Drucken im Hintergrund gestartet.
 7 Aufruf mit :
 8   spool' listing
 9   spool' printall
10   from to spool' pthru
11   from to spool' document
12 Vor (oder auch nach) dem Aufruf von spool' muß der Multitasker
13  mit multitask eingeschaltet werden.
14
15
```

5) Der Heap

Eine der ungewöhnlichen und fortschrittlichen Konzepte des volksFORTH83 besteht in der Möglichkeit, Namen von Worten zu entfernen, ohne den Rumpf zu vernichten.
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 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 Speicherbereich abgelegt, der Heap heißt. Der Heap kann später mit dem Wort CLEAR gelöscht werden. Dann sind natürlich auch alle Namen, die sich im Heap befanden, verschwunden.

Beispiel:
```
¦ Variable sum
1 sum !
```

ergibt :



Es werden weitere Worte definiert und dann CLEAR ausgeführt:

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

liefert die Worte CLEARSUM ADD SHOW , während der Name SUM durch CLEAR entfernt wurde; das Codefeld und der Wert 0001 existieren jedoch noch. (Das Beispiel soll eine Art Taschenrechner darstellen.)

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. Schauen Sie bitte in den Quelltext des Assemblers, um zu sehen, wie man das macht.

6) Der Multitasker

Das volksFORTH besitzt einen recht einfachen, aber leistungsfähigen Multitasker. Er ermöglicht die Konstruktion von Druckerspoolern, Uhren, Zählern und anderen einfachen Tasks. Als Beispiel soll gezeigt werden, wie man einen einfachen Druckerspooler konstruiert. Dieser Spooler ist in verbesserter Form auch im Quelltext des Multitaskers enthalten, hier wird er aus didaktischen Gründen möglichst simpel gehalten.

6.1) Anwendungsbeispiel: Ein Kochrezept

Das Programm für einen Druckerspooler lautet:

```
$F0 $100  Task background
: spool    background activate   1 100 pthru stop ;
multitask   spool
```

Normalerweise würde PTHRU den Rechner "lahmlegen", bis die Screens von 1 bis 100 ausgedruckt worden sind. Bei Aufruf von SPOOL ist das nicht so; der Rechner kann sofort weitere Eingaben verarbeiten. Damit alles richtig funktioniert, muß PTHRU allerdings einige Voraussetzungen erfüllen, die dieses Kapitel erklären will.
Das Wort TASK ist ein definierendes Wort, das eine Task erzeugt. Die Task besitzt übrigens Userarea, Stack, Returnstack und Dictionary unabhängig von der sog. Konsolen- oder Main-Task. Im Beispiel ist $F0 die Länge des reservierten Speicherbereichs für Returnstack und Userarea, $100 die Länge für Stack und Dictionary, jeweils in Bytes. Der Name der Task ist in diesem Fall BACKGROUND. Die neue Task tut nichts, bis sie aufgeweckt wird. Das geschieht durch das Wort SPOOL .
MULTITASK sagt dem Rechner, daß in Zukunft womöglich noch andere Tasks außer der Konsolentask auszuführen sind. Es schaltet also den Taskwechsler ein.
Bei Ausführen von SINGLETASK wird der Taskwechsler abgeschaltet. Dann wird nur noch die gerade aktive Task ausgeführt.

Bei Ausführung von SPOOL geschieht nun folgendes:

Die Task BACKGROUND wird aufgeweckt und ihr wird der Code hinter ACTIVATE (nämlich 1 100 PTHRU STOP) zur Ausführung übergeben. Damit ist die Ausführung von SPOOL beendet, es können jetzt andere Worte eingetippt werden. Die Task jedoch führt unverdrossen 1 100 PTHRU aus, bis sie damit fertig ist. Dann stößt sie auf STOP und hält an. Man sagt, die Task schläft.
Will man die Task während des Druckvorganges anhalten, z.B. um Papier nachzufüllen, so tippt man BACKGROUND SLEEP ein. Dann wird BACKGROUND vom Taskwechsler übergangen. Soll es weitergehen, so tippt man BACKGROUND WAKE ein.

Häufig möchte man erst bei Aufruf von SPOOL den Bereich als Argument angeben, der ausgedruckt werden soll. Das geht wie folgt:

```
: newspool  ( from to -- )
          2 background pass   pthru  stop ;
```

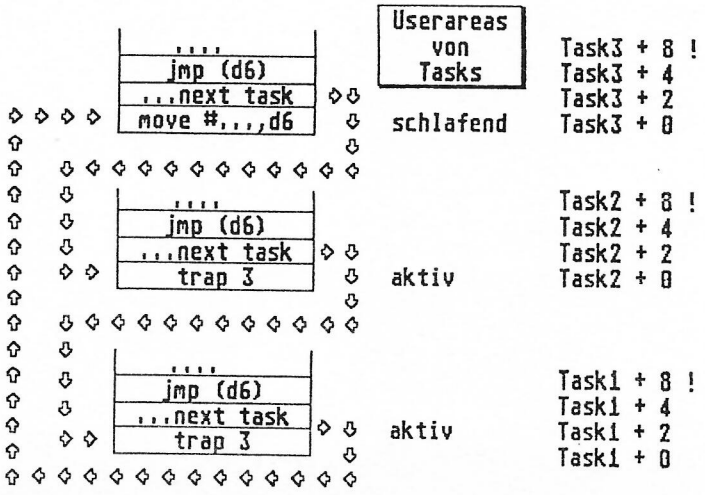
Die Phrase 2 BACKGROUND PASS funktioniert ähnlich wie BACKGROUND ACTIVATE , jedoch werden der Task auf dem Stack zusätzlich die beiden obersten Werte (hier from und to) übergeben. Um die Screens 1 bis 100 auszudrucken, tippt man jetzt ein:

```
1 100 newspool
```

Es ist klar, daß BACKGROUND ACTIVATE gerade der Phrase 0 BACKGROUND PASS entspricht.

6.2) Implementation

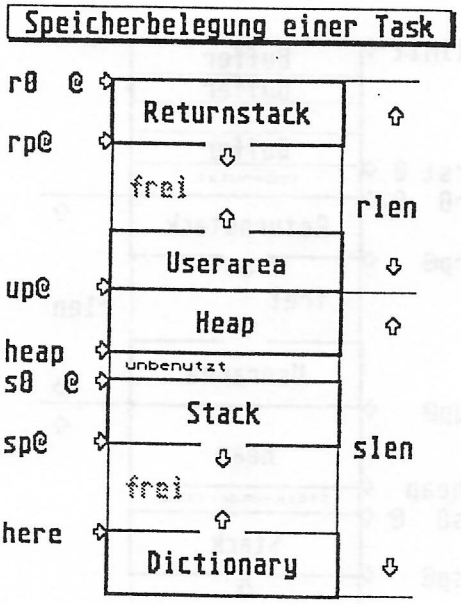
Der Unterschied dieses Multitaskers zu herkömmlichen liegt in seiner kooperativen Struktur begründet. Damit ist gemeint, daß jede Task explizit die Kontrolle über den Rechner und die Ein/Ausgabegeräte aufgeben und damit für andere Tasks verfügbar machen muß. Jede Task kann aber selbst "wählen", wann das geschieht. Es ist klar, daß das oft genug geschehen muß, damit alle Tasks ihre Aufgaben wahrnehmen können.
Die Kontrolle über den Rechner wird durch das Wort PAUSE aufgegeben. PAUSE führt den Code aus, der den gegenwärtigen Zustand der gerade aktiven Task rettet und die Kontrolle des Rechners an den Taskwechsler übergibt. Der Zustand einer Task besteht aus den Werten von Interpreterpointer (IP) , Returnstackpointer (RP) und des Stackpointers (SP).
Der Taskwechsler besteht aus einer geschlossenen Schleife. Jede Task enthält einen Maschinencodesprung auf die nächste Task, gefolgt von der Aufweckprozedur. Beim volksFORTH83 für den Atari besteht dieser Sprung aus zwei Befehlen, nämlich "move.w #next task,d6" und "jmp 0(FP,d6)". Zunächst wird also die Adresse (vom Anfang des Forthsystems aus gerechnet) geladen, durch den Sprungbefehl wird diese Adresse auf den Anfang des Speichers umgerechnet und dann auf diese Adresse gesprungen (mehr zu dieser Umrechnung finden Sie im Kapitel über den Assembler). Dort befindet sich die entsprechende Instruktion der nächsten Task. Ist die Task gestoppt, so wird dort ebenfalls ein Maschinencodesprung zur nächsten Task ausgeführt. Ist die Task dagegen aktiv, so ist der Sprung durch den Trap #3 ersetzt worden, der die Aufweckprozedur auslöst. Diese Prozedur lädt den Zustand der Task (bestehend aus SP , RP und IP) und setzt den Userpointer (UP), so daß er auf diese Task zeigt (siehe Bild).



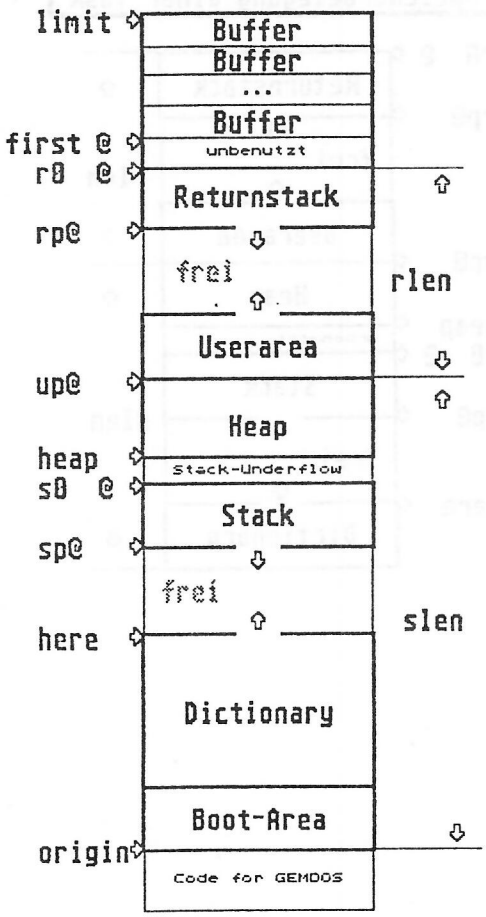
SINGLETASK ändert nun PAUSE so, daß überhaupt kein Taskwechsel stattfindet, wenn PAUSE aufgerufen wird. Das ist in dem Fall sinnvoll, wenn nur eine Task existiert, nämlich die Konsolentask, die beim Kaltstart des Systems "erzeugt" wurde. Dann würde PAUSE unnötig Zeit damit verbrauchen, einen Taskwechsel auszuführen, der sowieso wieder auf dieselbe Task führt. STOP entspricht PAUSE , jedoch mit dem Unterschied, daß die leere Anweisung durch einen Sprungbefehl ersetzt wird. Das System unterstützt den Multitasker, indem es während vieler Ein/Ausgabeoperationen wie KEY , TYPE und BLOCK usw. PAUSE ausführt. Häufig reicht das schon aus, damit eine Task (z.B. der Druckerspooler) gleichmäßig arbeitet.

Tasks werden im Dictionary der Konsolentask erzeugt. Jede besitzt ihre eigene Userarea mit einer Kopie der Uservariablen. Die Implementation des Systems wird aber durch die Einchränkung vereinfacht, daß nur die Konsolentask Eingabetext interpretieren bzw. kompilieren kann. Es gibt z.B. nur eine Suchreihenfolge, die im Prinzip für alle Tasks gilt. Da aber nur die Konsolentask von ihr Gebrauch macht, ist das nicht weiter störend.
Es ist übrigens möglich, aktive Tasks mit FORGET usw. zu vergessen. Das ist eine Eigenschaft, die nicht viele Systeme aufweisen! Allerdings geht das manchmal auch schief... Nämlich dann, wenn die vergessene Task einen "Semaphor" (s.u.) besaß. Der wird beim Vergessen nämlich nicht freigegeben und damit ist das zugehörige Gerät blockiert.

Schließlich sollte man noch erwähnen, daß beim Ausführen eines Tasknamens der Beginn der Userarea dieser Task auf dem Stack hinterlassen wird.


Speicherbelegung einer Task
r0 @
rp@
Returnstack
frei
rlen
Userarea
up@
Heap
heap
unbenutzt
s0 @
Stack
sp@
slen
frei
here
Dictionary

Memory Map des volksFORTH83



6.3) Semaphore und "Lock"

Ein Problem, daß bisher noch nicht erwähnt wurde, ist: Was passiert, wenn zwei Tasks gleichzeitig drucken (oder Daten von der Diskette lesen) wollen?
Es ist klar: Um ein Durcheinander oder Fehler zu vermeiden, darf das immer nur eine Task zur Zeit.
Programmtechnisch wird das Problem durch "Semaphore" gelöst:

```
Create disp 0 ,
: newtype   disp lock  type  disp unlock;
```

Der Effekt ist der folgende:
Wenn zwei Tasks gleichzeitig NEWTYPE ausführen, so kann doch nur eine zur Zeit TYPE ausführen, unabhängig davon, wie viele PAUSE in TYPE enthalten sind. Die Phrase DISP LOCK schaltet nämlich hinter der ersten Task, die sie ausführt, die "Ampel auf rot" und läßt keine andere Task durch. Die anderen machen solange PAUSE , bis die erste Task die Ampel mit DISP UNLOCK wieder auf grün umschaltet. Dann kann eine (!) andere Task die Ampel hinter sich umschalten usw. .
Übrigens wird die Task, die die Ampel auf rot schaltete, bei DISP LOCK nicht aufgehalten, sondern durchgelassen. Das ist notwendig, da ja TYPE ebenfalls DISP LOCK enthalten könnte (Im obigen Beispiel natürlich nicht, aber es ist denkbar) .
Die Implementation sieht nun folgendermaßen aus:

(Man muß sich noch vor Augen halten, daß jede Task eindeutig durch den Anfang ihrer Userarea identifizierbar ist.)

DISP ist ein sog. Semaphor; er muß den Anfangswert 0 haben!

LOCK schaut sich nun den Semaphor an:
Ist er Null, so wird die gerade aktive Task (bzw. der Anfang ihrer Userarea) in den Semaphor eingetragen und die Task darf weitermarschieren.
Ist der Wert des Semaphors gerade die aktive Task, so darf sie natürlich auch weiter.
Wenn aber der Wert des Semaphors von dem Anfang der Userarea der aktiven Task abweicht, dann ist gerade eine andere Task hinter der Ampel aktiv und die Task muß solange PAUSE machen, bis die Ampel wieder grün, d.h. der Semaphor null ist.

UNLOCK muß nun nichts anderes mehr tun, als den Wert des Semaphors wieder auf Null setzen.

BLOCK und BUFFER sind übrigens auf diese Weise für die Benutzung durch mehrere Tasks gesichert: Es kann immer nur eine Task das Laden von Blöcken von der Diskette veranlassen.
Ob TYPE , EMIT usw. ebenfalls gesichert sind, hängt von der Implementation ab.

6.4) Eine Bemerkung bzgl. BLOCK und anderer Dinge

Wie man dem Glossar entnehmen kann, ist immer nur die Adresse des zuletzt mit BLOCK oder BUFFER angeforderten Blockpuffers gültig, d.h. ältere Blöcke sind, je nach der Zahl der Blockpuffer, womöglich schon wieder auf die Diskette ausgelagert worden. Auf der sicheren Seite liegt man, wenn man sich vorstellt, daß nur ein Blockpuffer im gesamten System existiert.
Nun kann jede Task BLOCK ausführen und damit anderen Tasks die Blöcke "unter den Füßen" wegnehmen. Daher sollte man nicht die Adresse eines Blocks nach einem Wort, das PAUSE ausführt, weiter benutzen, sondern lieber neu mit BLOCK anfordern. Ein Beispiel:

```
: .line  ( block -- )
   block  c/l  bounds  DO  I c@ emit  LOOP ;
```

ist FALSCH, denn nach EMIT stimmt der Adressbereich, den der Schleifenindex überstreicht, womöglich gar nicht mehr.

```
: .line  ( block -- )
   c/l 0  DO  dup block  I +  c@  emit  LOOP  drop ;
```

ist RICHTIG, denn es wird nur die Nummer des Blocks, nicht die Adresse seines Puffers aufbewahrt.

```
: .line  ( block -- )    block  c/l  type ;
```

ist FALSCH, da TYPE ja EMIT wiederholt ausführen kann und somit die von BLOCK gelieferte Adresse in TYPE ungültig wird.

```
: >type  ( adr len -- ) >r  pad  r@  cmove  pad r> type ;
: .line  ( block -- )    block  c/l  >type ;
```

ist RICHTIG, denn PAD ist für jeden Task verschieden.

7) Debugging - Techniken

Fehlersuche ist in allen Programmiersprachen die aufwendigste Aufgabe des Programmierers.
Für verschiedene Programme sind in der Regel auch verschiedene Hilfsmittel erforderlich. Daher kann dieses Kapitel die Fehlersuche nicht erschöpfend behandeln. Da aber Anfänger häufig typische Fehler machen, kann man gerade für diese Gruppe brauchbare Hilfsmittel angeben.

7.1) Voraussetzungen für die Fehlersuche

Voraussetzung für die Fehlersuche ist immer ein übersichtliches und verständliches Programm. In Forth bedeutet das :

-) suggestive und prägnante Namen für Worte

-) starke Faktorisierung, d.h. sinnvoll zusammengehörende Teile eines Wortes sind zu einem eigenen Wort zusammengefaßt. Worte sollten durchschnittlich nicht länger als 2 - 3 Zeilen lang sein !

-) Übergabe von Parametern auf dem Stack statt in Variablen, überall wo das möglich ist.

Guter Stil in Forth ist nicht schwer, erleichtert aber sehr die Fehlersuche. Ein Buch, das auch diesen Aspekt behandelt, sei unbedingt empfohlen :
"Thinking Forth" von Leo Brodie, Prentice Hall 1984.

Sind diese Bedingungen erfüllt, ist es meist möglich, die Worte interaktiv zu testen. Damit ist gemeint, daß man bestimmte Parameter auf dem Stack versammelt und anschließend das zu testende Wort aufruft. Anhand der abgelieferten Werte auf dem Stack kann man dann beurteilen, ob das Wort korrekt arbeitet.
Sinnvollerweise testet man natürlich die zuerst definierten Worte auch zuerst, denn ein Wort, das fehlerhafte Worte aufruft, funktioniert natürlich nicht korrekt.
Wenn nun ein Wort auf diese Weise als fehlerhaft identifiziert wurde, muß man das Geschehen innerhalb des Wortes verfolgen. Das geschieht meist durch "tracen".

7.2) Der Tracer

Angenommen, Sie wollen folgendes Wort auf Fehler untersuchen: (bitte tippen Sie ein, alle nötigen Eingaben sind fett und alle Ausgaben des volksFORTH83 sind unterstrichen dargestellt.)

```
: -trailing  ( addr1 n1 -- addr1 n2 )  compiling
   2dup bounds  ?DO  2dup + 1-  c@ bl = compiling
   IF  LEAVE  THEN  1-  LOOP ;  ok
```

Die Funktion dieses Wortes können Sie, wenn sie Ihnen unklar ist, dem Glossar entnehmen. Zum Testen des Wortes wird ein String benötigt. Sie definieren:

```
Create teststring ," Dies ist ein Test       "  ok
```

wobei Sie bitte absichtlich einige zusätzliche Leerzeichen eingefügt haben. Sie geben nun ein :

```
teststring count .s 16 7E39  ok
-trailing .s 16 7E39  ok
```

und stellen zu Ihrem Erstaunen fest, daß -TRAILING kein einziges Leerzeichen abgeschnitten hat. (Spätestens jetzt sollten Sie am Rechner sitzen und den Tracer laden, wenn er noch nicht im System vorhanden ist. Prüfen Sie, ob es das Wort DEBUG im FORTH Vokabular gibt, dann ist der Tracer vorhanden. Der Tracer gehört zu den sogenannten Tools. Die Quelltexte finden Sie auf Ihrer Diskette).

Mit dem Tracer können Sie Worte, die mit dem : definiert wurden, schrittweise testen. Um den Tracer zu benutzen, geben Sie folgendes ein:

```
debug <name1>  ok
```

Hierbei ist <name1> das zu tracende Wort. Zunächst geschieht noch gar nichts. DEBUG hat nur den Tracer "scharf gemacht". Geben Sie nun eine Sequenz ein, die <name1> enthält, unterbricht der Tracer die Ausführung, wenn er auf <name1> stößt. Es erscheint folgendes Bild.

```
addr1 addr2 <name2>                (Werte)
```

Hierbei ist addr1 eine Adresse im Parameterfeld von <name1>, nämlich die, in der addr2 steht. addr2 ist die Kompilations-Adresse von <name2>. <name2> ist das Wort, das als nächstes ausgeführt werden soll. (Werte) sind die Werte, die gerade auf dem Stack liegen.

Bleiben wir bei unserem Beispiel, Sie geben ein:

```
debug -trailing ok
```

Es geschieht zunächst nichts.

Nun versuchen Sie es wieder mit der Sequenz

```
teststring count .s 16 7E39 ok
-trailing
```

und erhalten folgendes Bild:

```
7E04  536  2DUP                    16 7E39
```

16 7E39 ist der Stackinhalt, wie er von TESTSTRING COUNT geliefert wurde, nämlich Adresse und Länge des Strings TESTSTRING .

Natürlich können die Zahlen bei Ihnen anders aussehen, je nachdem, wohin TESTSTRING und -TRAILING kompiliert wurde. 536 ist die Kompilationsadresse von 2DUP , 7E04 die Position von 2DUP in -TRAILING . (Auch diese Adressen können sich geändert haben!) Diese Zahlen werden mit ausgegeben, so daß auch im Falle mehrerer Worte mit gleichem Namen eine Identifizierung möglich ist.

Drücken Sie jetzt so lange <Return>, bis OK erscheint. Insgesamt wird dabei folgendes ausgegeben :

```
debug -trailing   teststring count .s 16 7E39  ok
-trailing
7E04   536  2DUP                    16 7E39
7E06   954  BOUNDS                  16 7E39 16 7E39
7E08   92C  (?DO                    7E39 7E4F 16 7E39
7E0C   536  2DUP                    16 7E39
7E0E   546  +                       16 7E39 16 7E39
7E10   64C  1-                      7E4F 16 7E39
7E12   34E  C@                      7E4E 16 7E39
7E14  215C  BL                      20 16 7E39
7E16   816  =                       20 20 16 7E39
7E18   A0C  ?BRANCH                 FFFF 16 7E39
7E1C   BE2  LEAVE                   16 7E39
7E24   2F4  UNNEST                  16 7E39    ok
```

Sehen wir uns die Ausgabe nun etwas genauer an. Bei den ersten beiden Zeilen wächst der Wert ganz links immer um 2. Es ist der Inhalt des Instructionpointers (IP), der immer auf die nächste auszuführende Adresse zeigt. Der Inhalt dieser Adresse ist jeweils eine Kompilationsadresse (536 bei 2DUP usw.). Jede Kompilationsadresse benötigt zwei Bytes, daher muß der IP immer um 2 erhöht werden.

Immer? Nein, denn schon die nächste Zeile zeigt eine Ausnahme. Das Wort (?DO erhöht den IP um 4 ! Woher kommt eigentlich (?DO , in der Definition von -TRAILING stand doch nur ?DO . (?DO ist ein von ?DO kompiliertes Wort, das zusätzlich zur Kompilationsadresse noch einen 16-Bit-Wert benötigt, nämlich für den Sprungoffset hinter LOOP, wenn die Schleife beendet ist. Zwei ähnliche Fälle treten noch auf. Das IF aus dem Quelltext hat ein ?BRANCH kompiliert. Es wird gesprungen, wenn der oberste Stackwert FALSE (= 0) ist. Auch ?BRANCH benötigt einen zusätzlichen 16-Bit-Wert für den Sprungoffset.

Nach LEAVE geht es hinter LOOP weiter, es wird UNNEST ausgeführt, das vom ; in -TRAILING kompiliert wurde und das gleiche wie EXIT bewirkt, und damit ist das Wort -TRAILING auch beendet. Das hier gelistete Wort UNNEST ist nicht zu verwechseln mit dem UNNEST des Tracers (siehe unten).

Wo liegt nun der Fehler in unserer Definition von -TRAILING ? Bevor Sie weiterlesen, sollten Sie die Fehlerbeschreibung, den Tracelauf und Ihre Vorstellung von der korrekten Arbeitsweise des Wortes noch einmal unter die Lupe nehmen.

Der Stack ist vor und nach -TRAILING gleich geblieben, die Länge des Strings also nicht verändert worden. Offensichtlich wird die Schleife gleich beim ersten Mal verlassen, obwohl das letzte Zeichen des Textes ein Leerzeichen war. Die Schleife hätte also eigentlich mit dem vorletzten Zeichen weiter machen müssen. Mit anderen Worten:

Die Abbruchbedingung in der Schleife ist falsch. Sie ist genau verkehrt herum gewählt. Ersetzt man = durch = NOT oder - , so funktioniert das Wort korrekt. Überlegen Sie bitte, warum - statt = NOT eingesetzt werden kann. (Tip: der IF -Zweig wird nicht ausgeführt, wenn der oberste Stackwert FALSE (also = 0) ist.)

Der volksFORTH83-Tracer gestattet es, jederzeit Befehle einzugeben, die vor dem Abarbeiten des nächsten Trace-Kommandos ausgeführt werden. Damit kann man z. B. Stack-Werte verändern oder das Tracen abbrechen. Ändern Sie probehalber beim nächsten Trace-Lauf von -TRAILING das TRUE-Flag ($FFFF) auf dem Stack vor der Ausführung von ?BRANCH durch Eingabe von NOT auf 0 und verfolgen Sie den weiteren Trace-Lauf. Sie werden bemerken, daß die LOOP ein zweites Mal durchlaufen wird.

Wollen Sie das Tracen und die weitere Ausführung des getraceten Wortes abbrechen, so geben Sie **RESTART** ein. RESTART führt einen Warm-Start des FORTH-Systems aus und schaltet den Tracer ab. RESTART ist auch die Katastrophen-Notbremse, die man einsetzt, wenn man sieht, daß das System mit dem nächsten Befehl zwangsläufig im Computer-Nirwana entschwinden wird.

Nützlich ist auch die Möglichkeit, das Wort, das als nächstes

zur Ausführung ansteht, solange zu tracen, bis es ins aufrufende Wort zurückkehrt. Dafür ist das Wort **NEST** vorgesehen. Wollen Sie also wissen, was BOUNDS macht, so geben Sie bitte, wenn BOUNDS als nächstes auszuführendes Wort angezeigt wird, NEST ein und Sie erhalten dann:

```
7E04  536  2DUP                16 7E39
7E06  954  BOUNDS              16 7E39 16 7E39  nest
  956  44C  OVER                16 7E39 16 7E39
  958  546  +                   7E39 16 7E39 16 7E39
  95A  40A  SWAP                7E4F 7E39 16 7E39
  95C  2F4  UNNEST              7E39 7E4F 16 7E39
7E08  92C  (?DO                 7E39 7E4F 16 7E39
...
```

Beachten Sie bitte, daß die Zeilen jetzt eingerückt dargestellt werden, bis der Tracer automatisch in das aufrufende Wort zurückkehrt. Der Gebrauch von NEST ist nur dadurch eingeschränkt, daß sich einige Worte, die den Return-Stack manipulieren, mit NEST nicht tracen lassen, da der Tracer selbst Gebrauch vom Return-Stack macht. Auf solche Worte muß man den Tracer mit DEBUG ansetzen.

Wollen Sie das Tracen eines Wortes beenden, ohne die Ausführung des Wortes abzubrechen, so benutzen Sie **UNNEST** . (Ist der Tracer geladen, so kommen Sie an das tief im System steckende UNNEST, einem Synonym für EXIT, das ausschließlich vom ; kompiliert wird, nicht mehr heran und benutzen statt dessen das Tracer-UNNEST, das Sie eine Ebene im Trace-Lauf zurückbringt.)

Manchmal hat man in einem Wort vorkommende Schleifen beim ersten Durchlauf als korrekt erkannt und möchte diese nicht weiter tracen. Das kann sowohl bei DO...LOOPs der Fall sein als auch bei Konstruktionen mit BEGIN...WHILE...REPEAT oder BEGIN...UNTIL. In diesen Fällen gibt man am Ende der Schleife das Wort **ENDLOOP** ein. Die Schleife wird dann in Echtzeit abgearbeitet und der Tracer meldet sich erst wieder, wenn er nach dem Wort angekommen ist, bei dem ENDLOOP eingegeben wurde.

Haben Sie den Fehler gefunden und wollen deshalb nicht mehr tracen, so müssen Sie nach dem Ende des Tracens END-TRACE oder jederzeit RESTART eingeben, ansonsten bleibt der Tracer "scharf", was zu merkwürdigen Erscheinungen führen kann; außerdem verringert sich bei eingeschaltetem Tracer die Laufzeit des Systems.

Der Tracer läßt sich auch mit dem Wort **TRACE'** starten, das seinerseits ein zu tracendes Forth-Wort erwartet. Sie könnten also auch im obigen Beispiel eingeben:

```
teststring count trace' -trailing
```

und hätten damit dieselbe Wirkung erzielt. TRACE' schaltet aber im Gegensatz zu DEBUG nach Durchlauf des Wortes den Tracer automatisch mit END-TRACE wieder ab.

Beachten Sie bitte auch, daß die Worte DEBUG und TRACE' das Vokabular TOOLS mit in die Suchreihenfolge aufnehmen. Sie sollten also nach - hoffentlich erfolgreichem - Tracelauf die Suchordnung wieder umschalten.

Wenn man sich eingearbeitet hat, ist der Tracer ein wirklich verblüffendes Werkzeug, mit dem man sehr viele Fehler schnell finden kann. Er ist gleichsam ein Mikroskop, mit dem man sehr tief ins Innere von Forth schauen kann.

7.3) Stacksicherheit

Anfänger neigen häufig dazu, Fehler bei der Stackmanipulation zu machen. Erschwerend kommt häufig hinzu, daß sie viel zu lange Worte schreiben, in denen es dann von unübersichtlichen Stackmanipulationen nur so wimmelt. Es gibt einige Worte, die sehr einfach sind und Fehler bei der Stackmanipulation früh erkennen helfen. Denn leider führen schwerwiegende Stackfehler zu "mysteriösen" Systemcrashs.

In Schleifen führt ein nicht ausgeglichener Stack oft zu solchen Fehlern. Während der Testphase eines Programms oder Wortes sollte man daher bei jedem Schleifendurchlauf prüfen, ob der Stack evtl. über- oder leerläuft.
Das geschieht durch Eintippen von :

```
: LOOP   compile ?stack  [compile] LOOP ; immediate restrict
: +LOOP  compile ?stack  [compile] +LOOP ; immediate restrict
: UNTIL  compile ?stack  [compile] UNTIL ; immediate restrict
: REPEAT compile ?stack  [compile] REPEAT ; immediate restrict
: :      : compile ?stack ;
```

Versuchen Sie ruhig, herauszufinden wie die letzte Definition funktioniert. Es ist nicht kompliziert.
Durch diese Worte bekommt man sehr schnell mitgeteilt, wann ein Fehler auftrat. Es erscheint dann die Fehlermeldung :

```
<name> stack full
```

wobei <name> der zuletzt vom Terminal eingegebene Name ist.
Wenn man nun überhaupt keine Ahnung hat, wo der Fehler auftrat, so gebe man ein :

```
: unravel   rdrop rdrop rdrop \ delete errorhandler-nest
      cr  ." trace dump on  abort is :"  cr
      BEGIN  rp@  r0 @ -      \ until stack empty
      WHILE  r> dup  8 u.r  space
             2- @ >name  .name  cr  REPEAT
      (error ;
' unravel  errorhandler !
```

Sie bekommen dann bei Eingabe von

```
1 2 0 */
```

ungefähr folgenden Ausdruck :

```
trace dump on  abort is:
    4678 M/MOD
    4692 */MOD
    9248 EXECUTE
   10060 INTERPRET
   10104 'QUIT/ division overflow
```

'QUIT INTERPRET und EXECUTE rühren vom Textinterpreter her. Interessant wird es bei */MOD . Wir wissen nämlich, daß */MOD von */ aufgerufen wird. */MOD ruft nun wieder M/MOD auf, in M/MOD gehts weiter nach UM/MOD . Dieses Wort ist in Code geschrieben und "verursachte" den Fehler, indem es eine Division durch Null ausführte.

Nicht immer treten Fehler in Schleifen auf. Es kann auch der Fall sein, daß ein Wort zu wenig Argumente auf dem Stack vorfindet, weniger nämlich, als Sie für dieses Wort vorgesehen haben. Diesen Fall sichert ARGUMENTS .
Die Definition dieses Wortes ist:

```
: ARGUMENTS   ( n -- )
    depth 1- <  abort" not enough arguments" ;
```

Es wird folgendermaßen benutzt:

```
: -trailing  ( adr len -- )  2 arguments   ... ;
```

wobei die drei Punkte den Rest des Quelltextes andeuten sollen. Findet -TRAILING nun weniger als zwei Werte auf dem Stack vor, so wird eine Fehlermeldung ausgegeben. Natürlich kann man damit nicht prüfen, ob die Werte auf dem Stack wirklich für -TRAILING bestimmt waren.

Sind Sie als Programmierer sicher, daß an einer bestimmten Stelle im Programm eine bestimmte Anzahl von Werten auf dem Stack liegt, so können Sie das ebenfalls sicherstellen:

```
: is-depth  ( n -- )  depth 1-  - abort" wrong depth" ;
```

IS-DEPTH bricht das Programm ab, wenn die Zahl der Werte auf dem Stack nicht n ist, wobei n natürlich nicht mitgezählt wird. Es wird analog zu ARGUMENTS benutzt.
Mit diesen Worten lassen sich Stackfehler oft feststellen und lokalisieren.

7.4) Aufrufgeschichte

Möchte man wissen, was geschah, bevor ein Fehler auftrat und nicht nur, wo er auftrat (denn nur diese Information liefert UNRAVEL), so kann man einen modifizierten Tracer verwenden, bei dem man nicht nach jeder Zeile <RETURN> drücken muß:

```
: : :
    Does>  cr  rdepth 2* spaces
           dup 2- >name .name  >r ;
```

Hierbei wird ein neues Wort mit dem Namen : definiert, das zunächst das alte Wort : aufruft und so ein Wort im Dictionary erzeugt. Das Laufzeitverhalten dieses Wortes wird aber so geändert, daß es sich jedesmal wieder ausdruckt, wenn es aufgerufen wird. Alle Worte, die nach der Redefinition (so nennt man das erneute Definieren eines schon bekannten Wortes) des : definiert wurden, weisen dieses Verhalten auf.

Beispiel :

```
: / / ;
: rechne ( -- n)  1 2 3 / / ;

RECHNE
  /
  /   RECHNE  division overflow
```

Wir sehen also, daß erst bei der zweiten Division der Fehler auftrat. Das ist auch logisch, denn 2 3 / ergibt 0 .

Sie sind sicher in der Lage, die Grundidee dieses zweiten Tracers zu verfeinern. Ideen wären z.B. :

- Ausgabe der Werte auf dem Stack bei Aufruf eines Wortes
- Die Möglichkeit, Teile eines Tracelaufs komfortabel zu unterdrücken.

7.5) Speicherdump

Ein Dump des Speichers benötigt man beim Programmieren sehr oft, mindestens dann, wenn man eigene Datenstrukturen anschauen will. Oft ist es dann hinderlich, eigene Worte zur (womöglich gar formatierten) Ausgabe der Datenstrukturen schreiben zu müssen. In diesen Fällen benötigt man ein Wort, das einen Speicherdump ausgibt. Das volksFORTH besitzt zwei Worte zum Dumpen von Speicherblöcken sowie einen Dekompiler, der auch für Datenstrukturen verwendet werden kann.

```
DUMP    ( addr n -- )
```

Ab addr werden n Bytes formatiert ausgegeben. Dabei steht am Anfang einer Zeile die Adresse (abgerundet auf das nächste Vielfache von $10) , dann folgen 16 Byte, in der Mitte zur besseren Übersicht getrennt und dann die Ascii-Darstellung. Dabei werden nur Zeichen im Bereich zwischen $20 und $7F ausgegeben, alle anderen Werte werden als Punkt angezeigt. Die Ausgabe läßt sich jederzeit mit einer beliebigen Taste unterbrechen oder mit <Esc> abbrechen. Mit PRINT läßt sich die Ausgabe auf einen Drucker umleiten. Beispiel:

```
print  pad 40 dump  display
```

Das abschließende DISPLAY sorgt dafür, daß wieder der Bildschirm als Ausgabegerät gesetzt wird.

Möchte man Speicherbereich außerhalb des FORTH-Systems dumpen, gibt es dafür das Wort

```
LDUMP    ( laddr n -- )
```

laddr muß eine - doppelt lange - absolute Speicheradresse sein, n gibt wie oben die Anzahl der Bytes an. Da das FORTH-System immer im sogenannten Supervisormodus arbeitet, können alle Speicherbereiche einschließlich der Trap-Vektoren usw. angezeigt werden. Die Ausgabe auf einen Drucker geschieht genau so wie oben beschrieben.

7.6) Der Dekompiler

Ein Dekompiler gehört so zu sagen zum guten Ton eines FORTH-Systems, war er bisher doch die einzige Möglichkeit, wenigstens ungefähr den Aufbau eines Systems zu durchschauen. Bei volksFORTH-83 ist das anders, und zwar aus zwei Gründen:

-) Sie haben sämtliche Quelltexte vorliegen, und es gibt die VIEW-Funktion. Letztere ist normalerweise sinnvoller als der beste Dekompiler, da kein Dekompiler in der Lage ist, z.B. Stackkommentare zu rekonstruieren.

-) Der Tracer ist beim Debugging sehr viel hilfreicher als ein Dekompiler, da er auch die Verarbeitung von Stackwerten erkennen läßt. Damit sind Fehler leicht aufzufinden.

Dennoch gibt es natürlich auch im volksFORTH einen Dekompiler, allerdings in einfacher, von Hand zu bedienender, Form. Er befindet sich, wie der Tracer, im Vokabular TOOLS. Folgende Worte stehen zur Verfügung.

N (name) (addr -- addr')
druckt den Namen des bei addr kompilierten Wortes aus und setzt addr auf das nächste Wort.

K (konstante) (addr -- addr')
wird nach LIT benutzt und druckt den Inhalt von addr als Zahl aus. Es wird also nicht versucht, den Inhalt, wie bei N, als Forthwort zu interpretieren.

S (string) (addr -- addr')
wird nach (ABORT" (" (." und allen anderen Worten benutzt, auf die ein String folgt. Der String wird ausgedruckt und addr entsprechend erhöht, sodaß sie hinter den String zeigt.

C (character) (addr -- addr')
druckt den Inhalt von addr als Ascii-Zeichen aus und geht ein Byte weiter. Damit kann man eigene Datenstrukturen ansehen.

B (branch) (addr -- addr')
wird nach BRANCH oder ?BRANCH benutzt und druckt den Inhalt einer Adresse als Sprungoffset und Sprungziel aus.

D (dump) (addr n -- addr')
Dumped n Bytes. Wird benutzt, um selbstdefinierte Datenstrukturen anzusehen. (s.a. DUMP und LDUMP)

Sehen wir uns nun ein Beispiel zur Benutzung des Dekompilers an. Geben Sie bitte folgende Definition ein:

```
: test   ( n -- )
   12 = IF  cr ." Die Zahl ist zwölf !"  THEN ;
```

Rufen Sie das Vokabular TOOLS - durch Nennen seines Namens auf und ermitteln Sie die Adresse des ersten in TEST kompilierten Wortes:

```
' test >body
```

Jetzt können Sie TEST nach folgendem Muster dekompilieren

```
n A988:    B54  LIT  ok
k A98A:     12  ok
n A98C:    C52  =  ok
n A98E:   E74  ?BRANCH  ok
b A990:    1A  A9AA  ok
n A992:   32BC  CR  ok
n A994:   1776  (."  ok
s A996:   12    Die Zahl ist zwölf  ok
n A99A:    416  UNNEST
drop
```

Die erste Adresse ist die, an der im Wort TEST die anderen Worte kompiliert sind. Die zweite ist jeweils die Kompilationsadresse der Worte, danach folgen die sonstigen Ausgaben des Dekompilers.

Probieren Sie dieses Beispiel auch mit dem Tracer aus:

```
20 trace' test
```

und achten Sie auf die Unterschiede. Sie werden sehen, daß der Tracer aussagefähiger und dazu noch einfacher zu bedienen ist.

Wenn Sie sich die Ratschläge und Tips zu Herzen genommen haben und noch etwas den Umgang mit den Hilfsmitteln üben, werden Sie sich sicher nicht mehr vorstellen können, wie Sie jemals in anderen Sprachen ohne diese Hilfsmittel ausgekommen sind. Bedenken Sie bitte auch : Diese Mittel sind kein Heiligtum ; oft wird Sie eine Modifikation schneller zum Ziel führen. Scheuen Sie sich nicht, sich vor dem Bearbeiten einer umfangreichen Aufgabe erst die geeigneten Hilfsmittel zu verschaffen.