16. Der Multitasker

Ein wichtiger Aspekt der FORTH-Programmierung 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.

Das volksFORTH83 besitzt einen einfachen, aber leistungsfähigen Multitasker. Er ermöglicht die Konstruktion von Druckerspoolern, Uhren, Zählern und anderen einfachen Tasks, kann aber auch für Aufgaben eingesetzt werden, die über einen Druckerspooler hinausgehen.
Als Beispiel soll gezeigt werden, wie man einen einfachen Druckerspooler konstruiert. 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. Eine 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** .

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.

Der neue Task tut nichts, bis sie aufgeweckt wird. Das geschieht durch das Wort **SPOOL** . 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
```

Somit entspricht BACKGROUND ACTIVATE gerade der Phrase 0 BACKGROUND PASS .

16.1 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 des Interpreterpointers (IP) , des Returnstackpointers (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. 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 einen 1-Byte Call auf einen Vektor 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.

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.

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 Einschränkung 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.

Der Multitasker beim volksFORTH ist gegenüber z.B. dem des polyFORTH vereinfacht, da volksFORTH ist kein Multiuser-System ist. So besitzen alle Terminal-Einheiten (wir nennen sie Tasks) gemeinsam nur ein Lexikon und einen Eingabepuffer. Es darf daher nur der OPERATOR (wir nennen ihn Main- oder Konsolen-Task) kompilieren.

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.

16.2 Semaphore und Lock

Ein Problem, daß bisher noch nicht erwähnt wurde, ist die Frage, 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:

```
Variable disp  disp off
: 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, wobei man sich noch vor Augen halten muß, 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.

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.

Das Beispiel

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

ist <u>falsch</u>, 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 <u>richtig</u>, 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 ( addr len -- )   pad place pad count type ;
```

ist multitasking-sicher, wenn ein String vom **BLOCK** geholt wird.

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

ist deshalb richtig, denn PAD ist für jeden Task verschieden.

16.3 Glossar

activate (Taddr --) -tasker.scr-

aktiviert die Task, die durch Taddr gekennzeichnet ist, und weckt sie auf.
Vergleiche **SLEEP** , **STOP** , **PASS** , **PAUSE** , **UP@** , **UP!** und **WAKE** .

lock (semaddr --) -kernel.scr-

Der Semaphor (eine **VARIABLE**), dessen Adresse auf dem Stack liegt, wird von der Task, die **LOCK** ausführt, blockiert.
Dazu prüft **LOCK** den Inhalt des Semaphors. Zeigt der Inhalt an, daß eine andere Task den Semaphor blockiert hat, so wird **PAUSE** ausgeführt, bis der Semaphor freigegeben ist. Ist der Semaphor freigegeben, so schreibt **LOCK** das Kennzeichen der Task, die **LOCK** ausführt, in den Semaphor und sperrt ihn damit für alle anderen Tasks. Den FORTH-Code zwischen **semaddr LOCK** ... und ... **semaddr UNLOCK** kann also immer nur eine Task zur Zeit ausführen.
Semaphore schützen gemeinsame Ressourcen (z.B. den Drucker) vor dem gleichzeitigen Zugriff durch verschiedene Tasks.
Vergleiche **UNLOCK** und **RENDEZVOUS** .

multitask (--) -tasker.scr-

schaltet das Multitasking ein.
Das Wort **PAUSE** ist dann keine NOOP-Funktion mehr, sondern gibt die Kontrolle über die FORTH-Maschine an eine andere Task weiter.

pass (nØ .. nr-1 Taddr r --) -tasker.scr-

aktiviert die Task, die durch Taddr gekennzeichnet ist, und weckt sie auf. r gibt die Anzahl der Parameter nØ bis nr-1 an, die vom Stack der **PASS** ausführenden Task auf den Stack der durch Taddr gekennzeichneten Task übergeben werden. Die Parameter nØ bis nr-1 stehen dieser Task dann in der gleichen Reihenfolge auf ihrem Stack zur weiteren Verarbeitung zur Verfügung.
Vergleiche **ACTIVATE** .

pause (--) -kernel.scr-

ist eine NOOP-Funktion, wenn der Singletask-Betrieb eingeschaltet ist; bewirkt jedoch, nach Ausführung von **MULTITASK** , daß die Task, die **PAUSE** ausführt, die Kontrolle über die FORTH-Maschine an eine andere Task abgibt.
Existiert nur eine Task, oder schlafen alle anderen Tasks, so wird die Kontrolle unverzüglich an die Task zurückgegeben, die **PAUSE** ausführte. Ist mindestens eine andere Task aktiv, so wird die Kontrolle von dieser übernommen und erst bei Ausführung von **PAUSE** oder **STOP** in dieser Task an eine andere Task weitergegeben. Da die Tasks ringförmig miteinander verkettet sind, erhält die Task, die zuerst **PAUSE** ausführte, irgendwann die Kontrolle zurück. Eine Fehlerbedingung liegt vor, wenn eine Task weder **PAUSE** noch **STOP** ausführt.
Vergleiche **STOP** , **MULTITASK** und **SINGLETASK** .

rendezvous (semaddr --) -tasker.scr-

gibt den Semaphor (die **VARIABLE**) mit der Adresse semaddr frei (siehe **UNLOCK**) und führt **PAUSE** aus, um anderen Tasks den Zugriff auf das, durch diesen Semaphor geschützte, Gerät zu ermöglichen. Anschließend wird **LOCK** ausgeführt, um das Gerät zurück zu erhalten. Dies ist eine Methode, eine zweite Task nur an einer genau benannten Stelle laufen zu lassen.

singletask (--) -tasker.scr-

schaltet das Multitasking aus.
PAUSE ist nach Ausführung von **SINGLETASK** eine NOOP-Funktion.
Eine Fehlerbedingung besteht, wenn eine Hintergrund-Task **SINGLETASK** ohne anschließendes **MULTITASK** ausführt, da die Main- oder Terminal-Task dann nie mehr die Kontrolle bekommt.
Vergleiche **UP@** und **UP!** .

sleep (Taddr --) -tasker.scr-

bringt die Task, die durch Taddr gekennzeichnet ist, zum Schlafen.
SLEEP hat den gleichen Effekt, wie die Ausführung von **STOP** durch die Task selbst. Der Unterschied ist, daß **STOP** in der Regel am Ende des Jobs der Task ausgeführt wird. **SLEEP** trifft die Task zu einem nicht vorhersehbaren Zeitpunkt, so daß die laufende Arbeit der Task abgebrochen wird.
Vergleiche **WAKE** .

stop (--) -tasker.scr-

bewirkt, daß die Task, die **STOP** ausführt, sich schlafen legt.
Wichtige Zeiger der FORTH-Maschine, die den Zustand der Task kennzeichnen, werden gerettet, dann wird die Kontrolle an die nächste Task abgegeben, deren Zeiger wieder der FORTH-Maschine übergeben werden, so daß diese Task ihre Arbeit an der alten Stelle aufnehmen kann. **PAUSE** führt diese Aktionen ebenfalls aus, der Unterschied zu **STOP** ist, daß die ausführende Task bei **PAUSE** aktiv, bei **STOP** hingegen schlafend hinterlassen wird.
Vergleiche **PAUSE** , **WAKE** und **SLEEP** .

Task (rlen slen <name> --) -tasker.scr-

ist ein definierendes Wort, das in der Form:

```
rlen slen Task <name>
```

benutzt wird. **TASK** erzeugt einen Arbeitsbereich für einen weiteren Job, der gleichzeitig zu schon laufenden Jobs ausgeführt werden soll.
Die Task erhält den Namen <name>, hat einen Stack-Bereich der Länge slen und einen Returnstack-Bereich der Länge rlen.
Im Stack Bereich liegen das Task-eigene Dictionary einschließlich **PAD** , das in Richtung zu höheren Adressen wächst, und der Daten-Stack, der zu niedrigen Adressen wächst. Im Returnstack-Bereich befinden sich die Task-eigene USER-Area (wächst zu höheren Adressen) und der Returnstack, der gegen kleinere Adressen wächst.
Eine Task ist ein verkleinertes Abbild des FORTH-Systems, allerdings ohne den Blockpuffer-Bereich, der von allen Tasks gemeinsam benutzt wird.
Zur Zeit ist es nicht zugelassen, daß Jobs einer Hintergrundtask kompilieren. Die Task ist nur der Arbeitsbereich für einen Hintergrund-Job, nicht jedoch der Job selbst.
Die Ausführung von <name> in einer beliebigen Task hinterläßt die gleiche Adresse, die die Task <name> selbst mit **UP@** erzeugt und ist zugleich die typische Adresse, die von **LOCK** , **UNLOCK** und **RENDEZVOUS** im Zusammenhang mit Semaphoren verwendet wird, bzw. von **ACTIVATE** , **PASS** , **SLEEP** und **WAKE** erwartet wird.

unlock (semaddr --) -kernel.scr-

gibt den Semaphor (die VARIABLE), dessen Adresse auf dem Stack ist, für alle Tasks frei. Ist der Semaphor im Besitz einer anderen Task, so wartet UNLOCK mit PAUSE auf die Freigabe. Vergleiche LOCK und die Beschreibung des Taskers.

up@ (-- Taddr) -kernel.scr- "u-p-fetch"
liefert die Adresse Taddr des ersten Bytes der USER-Area der Task, die **UP@** ausführt (siehe **TASK**). In der USER-Area sind Variablen und andere Datenstrukturen hinterlegt, die jede Task für sich haben muß.
Vergleiche **UP!** .

up! (addr --) -kernel.scr- "u-p-store"
richtet den **UP** (User Pointer) der FORTH-Maschine auf addr.
Vorsicht ist bei der Verwendung von **UP!** in Hintergrund-Tasks geboten.
Vergleiche **UP@** .

wake (Taddr --) -kernel.scr-
weckt die Task, die durch Taddr gekennzeichnet ist, auf.
Die Task führt ihren Job dort weiter aus, wo sie durch **SLEEP** angehalten wurde oder wo sie sich selbst durch **STOP** beendet hat (Vorsicht!).
Vergleiche **SLEEP** , **STOP** , **ACTIVATE** und **PASS** .

User (--) 83
ist ein definierendes Wort, benutzt in der Form:

```
User <name>
```

USER erzeugt einen Kopf für <name> im Dictionary und hält 2 Byte in der Userarea frei (siehe **UALLOT**).
Diese 2 Byte werden für den Inhalt der Uservariablen benutzt und werden nicht initialisiert. Im Parameterfeld der Uservariablen im Dictionary wird nur ein Offset zum Beginn der Userarea abgelegt. Wird <name> ausgeführt, so wird die Adresse des Wertes der Uservariablen in der Userarea auf den Stack gegeben. Uservariablen werden statt normaler Variablen z.B. dann benutzt, wenn der Einsatz des Multitaskers geplant ist und mindestens eine Task die Variable unbeeinflußt von anderen Tasks benötigt (Jede Task hat ihre eigene Userarea).

forget (--) 83
Wird in folgender Form benutzt:

```
FORGET <name>
```

Falls <name> in der Suchreihenfolge gefunden wird, so werden <name> und alle danach definierten Worte aus dem Dictionary entfernt. Wird <name> nicht gefunden, so wird eine Fehlerbehandlung eingeleitet. Liegt <name> in dem durch **SAVE** geschützten Bereich, so wird ebenfalls eine Fehlerbehandlung eingeleitet. Es wurden Vorkehrungen getroffen, die es ermöglichen, aktive Tasks und Vokabulare, die in der Suchreihenfolge auftreten, zu vergessen.

block (u -- addr) 83

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

BLOCK prüft den Pufferbereich auf die Existenz des Blocks Nummer u. Befindet sich der Block u in keinem der Blockpuffer, so wird er vom Massenspeicher in einen an ihn vergebenen Blockpuffer geladen. Falls der Block in diesem Puffer als **UPDATE**d markiert ist, wird er auf den Massenspeicher gesichert, bevor der Blockpuffer an den Block u vergeben wird. Nur die Daten im letzten Puffer, der über **BLOCK** oder **BUFFER** angesprochen wurde, sind sicher zugreifbar. Alle anderen Blockpuffer dürfen nicht mehr als gültig angenommen werden (möglicherweise existiert nur 1 Blockpuffer).

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

17. 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 Fehler-suche nicht erschöpfend behandeln. Da aber Anfänger häufig typische Fehler machen, kann man gerade für diese Gruppe brauchbare Hilfsmittel angeben.

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 mehr als 2 - 3 Zeilen lang sein !
- Übergabe von Parametern auf dem Stack statt in Variablen, wo immer es möglich ist.

Guter Stil in FORTH ist nicht schwer, erleichtert aber sehr die Fehlersuche. Ein Buch, das auch diesen Aspekt behandelt, sei unbedingt empfohlen : "In FORTH denken" von Leo Brodie, Hanser Verlag 1986.

Sind die genannten Bedingungen erfüllt, ist es meist möglich, die Worte interaktiv zu testen. Damit ist gemeint, daß man bestimmte Parameter auf den Stack legt und anschließend das zu testende Wort aufruft. Anhand der hinterlassenen 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".

17.1 Der Tracer

Allgemein wird man zuerst das fehlerhafte Wort isoliert testen wollen. Dazu legt man ausgewählte Werte als Parameter für dieses Wort auf den Stack und beobachtet die Verarbeitung dieser Werte innerhablb des Wortes.

Angenommen, Sie wollen das Wort -TRAILING auf Fehler untersuchen. Die Funktion dieses Wortes ist das "Abschneiden" von Leerzeichen am Ende einer Zeichenkette:

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

Dabei könnten Sie dem Wort schon im Editor ein anderes Erscheinungsbild geben, indem Sie z.B. die Schleifenanweisungen ?DO und LOOP bündig untereinander schreiben, durch das Einfügen von Leerzeichen eine Gliederung schaffen oder die IF...THEN-Anweisung auf einer eigenen Zeile isolieren.

Zum Testen des Wortes wird ein String benötigt. Sie definieren bitte:

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

Dabei haben Sie absichtlich einige zusätzliche Leerzeichen eingefügt.

Um die Werte auf dem Stack, meist Adressen, besser interpretieren zu können, schalten Sie mit HEX die Zahlenbasis um und geben nun ein (Antwort des Computers unterstrichen) :

```
hex ok
teststring .s 8686 ok
count      .s 1B 8687 ok
-trailing  .s 1B 8687 ok
```

Der Aufruf von **TESTSTRING** liefert Ihnen die Stringadresse (hexadezimal) 8686 und das Wort **COUNT** berechnet Ihnen daraus die Länge der Zeichenkette und ihren Beginn.: 1B 8687 (auch hexadezimal)
Beim Aufruf von **-TRAILING** stellen Sie an Hand der unveränderten Adressen 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 dazu, ob es das Wort **TOOLS** im FORTH-Vokabular gibt, dann ist der Tracer vorhanden. Denn der Tracer gehört zum Vokabular **Tools** , dessen Quelltexte Sie auf Ihrer Diskette finden.

Mit dem Tracer können Sie Worte, die mit dem : definiert wurden, schrittweise testen. Der Tracer läßt sich mit dem Wort **TRACE** starten, das seinerseits ein zu tracendes FORTH-Wort erwartet. **TRACE** schaltet im Gegensatz zu **DEBUG** nach Durchlauf des Wortes den Tracer automatisch mit **END-TRACE** wieder ab.

Um den Tracer zu benutzen, geben Sie nun folgendes ein:

```
teststring count
.s 1B 8687  ok
tools  trace -trailing
```

Es erscheint dieses Bild. wenn Sie nach dem Erscheinen einer jeden Zeile solange die <CR>Taste drücken, bis wieder ok erscheint:

```
8658  66A  2DUP              1B 8687
865A  AB2  BOUNDS            1B 8687 1B 8687
```

```
865C  A80  (?DO            8687 86A2 1B 8687
8660  66A  2DUP            1B 8687
8662  6E2  +               1B 8687 1B 8687
8664  789  1-              86A2 1B 8687
8666  469  C@              86A1 1B 8687
8668 2224  BL              20 1B 8687
866A  91B  =               20 20 1B 8687
866C  B34  ?BRANCH         FFFF 1B 8687
8670  CFB  LEAVE           1B 8687
8678  40D  UNNEST          1B 8687
```

Betrachten Sie zuerst die Syntax von **TRACE** :

```
trace <name1>
```

Hierbei ist <name1> das zu tracende Wort, wobei die vom Tracer zurückgelieferte Information so aussieht:

```
addr1 addr2 <name2>  Werte
```

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

Wie deuten Sie nun das eben erhaltene Bild ? In der ersten Zeile, die der Tracer beim Abarbeiten des zu untersuchenden Wortes **-TRAILING** angezeigt hat, finden Sie:

```
8658  66A  2DUP            1B 8687
```

Hier ist 1B 8687 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.
66A ist die Kompilationsadresse von **2DUP** , 8658 die Position von **2DUP** in **-TRAILING** . Auch diese Adressen können sich bei Ihnen geändert haben! Diese Zahlen werden mit ausgegeben, so daß auch im Falle mehrerer Worte mit gleichem Namen eine Identifizierung möglich ist.

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 66A 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** kompi-

liertes Wort, das zusätzlich zur Kompilationsadresse noch einen 16Bit-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 16Bit-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. Damit ist das Wort -TRAILING auch beendet. Das hier gelistete Wort UNNEST ist nicht zu verwechseln mit dem UNNEST des Tracers, siehe unten.m

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 auch - statt = NOT eingesetzt werden kann. Tip: Der IF-Zweig wird nicht ausgeführt, wenn der oberste Stackwert FALSE , also gleich NULL ist.

Der volksFORTH83-Tracer gestattet es, jederzeit Befehle einzu-geben, die vor dem Abarbeiten des nächsten Trace-Kommandos ausgeführt werden. Das System wartet nach einem Step auf eine Eingabe von der Tastatur, bevor der nächste Step ausgeführt wird.
Endet eine Zeile mit einem Leerzeichen vor dem <CR> , so erlaubt der Tracer die Eingabe und Verarbeitung einer vollständigen Kommandozeile. Die Anforderung wird wiederholt, wenn in dieser Zeile ein Eingabe-Fehler auftauchen sollte. Damit ist jetzt ein etwas stressfreieres Ändern des Code bzw. des Stack während des Tracens möglich.
So kann man z. B. Stack-Werte verändern oder das Tracen abbrechen. Ändern Sie probehalber beim nächsten Trace-Lauf von -TRAILING durch Eingabe von NOT das TRUE-Flag ($FFFF) auf dem Stack. bevor ?BRANCH ausgeführt wird 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 FORTHSystems 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 ComputerNirwana entschwinden wird.

17.2 Debug

Beim Tracen mit **TRACE** - nomen est omen - haben wir das fehlerhafte Wort isoliert zusammen mit ausgesuchten Parametern analysiert.

Im Gegensatz dazu bietet sich die Möglichkeit an, ein Wort in seiner Umgebung, also in der Definition, in der es fehlerhaft arbeitet, zu untersuchen. Dazu dient der Befehl **DEBUG** :

```
debug <name>
```

Hierbei ist <name> das zu tracende Wort.

17.2.1 Beispiel: EXAMPLE

Bleiben wir bei unserem Beispiel, geben Sie bitte eine Definition mit **-TRAILING** ein. Dabei bietet sich zunächst die gleiche Sequenz wie oben an. Bitte definieren Sie dieses Beispiel :

```
: example  ( -- )
    teststring count
    -trailing  clearstack  ;
```

Nun aktivieren Sie den Tracer mit :

```
debug -trailing
```

Zunächst geschieht noch gar nichts. **DEBUG** hat nur den Tracer "scharf" gemacht. Rufen Sie nun mit **EXAMPLE** ein Wort auf, das <name> enthält, unterbricht der Tracer die Ausführung, wenn er auf <name> stößt. Es erscheint folgendes, Ihnen schon bekanntes Bild:

```
example

8658  66A  2DUP              1B 8687
865A  AB2  BOUNDS            1B 8687 1B 8687
865C  A80  (?DO              8687 86A2 1B 8687
8660  66A  2DUP              1B 8687
8662  6E2  +                 1B 8687 1B 8687
8664  789  1-                86A2 1B 8687
...
```

Die Parameter sind unter diesen Bedingungen natürlich wieder die, die von **TESTSTRING** und **COUNT** auf den Stack gelegt werden. Interessant ist aber hier die Kontrolle, ob **-TRAILING** vielleicht falsche Parameter übergeben bekommt.

17.2.2 NEST und UNNEST

Nützlich ist auch die Möglichkeit, das Wort, das als nächstes zur Ausführung ansteht, seinerseits zu tracen, bis es ins aufrufende Wort zurückkehrt.

Dafür ist das Wort **nest** vorgesehen. Allerdings beschränkt sich dieses Möglichkeit auf Worte, die in Highlevel-FORTH geschrieben sind. Assemblercode läßt sich damit nicht debuggen und Sie erhalten eine Meldung:

```
<name> can't be debugged
```

Bitte tracen Sie unser Beispiel wieder mit **TRACE EXAMPLE** :

Wenn Sie nun wissen wollen, was **-TRAILING** innerhalb des Beispiels macht, so geben Sie bitte **NEST** ein, wenn **-TRAILING** als nächstes auszuführendes Wort angezeigt wird.

Sie erhalten dann:

```
trace example

86B0 8684 TESTSTRING
86B2  EEF COUNT              8686
86B4 8656 -TRAILING          1B 8687  nest
8658  66A 2DUP               1B 8687
865A  AB2 BOUNDS             1B 8687 1B 8687
865C  A80 ?DO                8687 86A2 1B 8687
8660  66A 2DUP               1B 8687
8662  6E2 +                  1B 8687 1B 8687
8664  789 1-                 86A2 1B 8687
8666  469 C@                 86A1 1B 8687
8668 2224 BL                 20 1B 8687
866A  91B =                  20 20 1B 8687
866C  B34 ?BRANCH            FFFF 1B 8687
8670  CFB LEAVE              1B 8687
8678  40D UNNEST             1B 8687
86B6 162E CLEARSTACK         1B 8687
86B8  40D UNNEST             ok
```

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 ReturnStack manipulieren, mit **NEST** nicht tracen lassen, da der Tracer selbst Gebrauch vom ReturnStack 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 Tracelauf 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 mel-

det sich erst wieder, wenn er nach dem Wort angekommen ist, bei dem ENDLOOP eingegeben wurde.
Sollte statt dessen die Meldung: ENDLOOP COMPILE ONLY zu sehen sein, so ist das Vokabular TOOLS nicht in der Suchreihenfolge . 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; außer dem verringert sich bei eingeschaltetem Tracer die Geschwindigkeit des Systems.
Beachten Sie bitte auch, daß Sie für die Worte DEBUG und TRACE das Vokabular TOOLS mit in die Suchreihenfolge aufgenommen haben. Sie sollten also nach hoffentlich erfolgreichem Tracelauf die Suchordnung wieder umschalten, weil der Befehl RESTART zwar den Tracer abschaltet, aber die Suchreihenfolge nicht zurückschaltet.

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.

17.3 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 errorhandlernest
    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:
E32  */MOD 1D8A EXECUTE
1E24 PARSER
20CD INTERPRET
20FA 'QUIT / division overflow
```

'QUIT INTERPRET PARSER und EXECUTE rühren vom Textinterpreter her. Interessant wird es bei */MOD . Wir wissen, 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 ( addr 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:

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

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

17.4 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 <CR> 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.

17.5 Dump

Einen Speicherdump 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, dann folgen 16 Byte, die in der Mitte zur besseren Übersicht getrennt sind und dann die Ascii-Darstellung. Dabei werden nur Zeichen im Bereich zwischen $20 und $7F ausgegeben. 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 Bild-schirm als Ausgabegerät gesetzt wird.

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

ldump (laddr n --)

erwartet eine - doppelt lange - absolute Speicheradresse und wie oben die Anzahl der auszugebenden Bytes. Die Ausgabe auf einen Drucker geschieht genau so wie oben beschrieben.

17.6 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 volks FORTH83 ist das anders, und zwar aus zwei Gründen: Sie haben sämtliche Quelltexte vorliegen, und es gibt die VIEW-Funktion. Letztere ist normalerweise sinn voller 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 leichter aufzufinden.
Dennoch gibt es natürlich auch im volksFORTH einen Dekompiler, zum einen als zuladbares Werkzeug und zum anderen in einfacher von Hand zu bedienender Form.

Der zuladbare Dekompiler **SEE** wird mit

```
include see.scr
```

geladen und in der Form

```
see <name>
```

benutzt. Daraufhin wird das Wort dekompilert.

Als ständig verfügbares Dekompiler-Werkzeug stehen im Vokabular TOOLS folgende Worte zur Verfügung:

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

L literal (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 FORTH-Wort 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 Daten strukturen anzusehen.
Siehe auch **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 80B0:  856  CLIT
c 80B2:   12
n 80B3:  92A  =
n 80B5:  B43  ?BRANCH
b 80B7:   1B  80D2
n 80B9: 2BE9  CR
n 80BB: 127E  (."
s 80BD:   14  Die Zahl ist zwölf !
n 80D2:  41C  UNNEST
```

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 programmieren.

17.7 Glossar

restart (--)
ermöglicht gegenüber den Versionen auf dem ATARI und dem C64 auf Grund der Konstruktion ausschließlich den Ausstieg aus dem Single-Step-Trace-Modus .

end-trace (--)
dient lediglich dazu, um einen DEBUG-Befehl rückgängig zu machen. Schaltet den Tracer ab, der durch "patchen" der Next-Routine arbeitet. Die Ausführung des aufrufenden Wortes wird fortgesetzt.

next-link (-- addr)
addr ist die Adresse einer Uservariablen, die auf die Liste aller Nextroutinen zeigt. Der Assembler erzeugt bei Verwendung des Wortes NEXT Code, für den ein Zeiger in diese Liste eingetragen wird. Die so entstandene Liste wird vom Tracer benutzt, um alle im System vorhandenen Kopien des Codes für NEXT zu modifizieren.

18. Begriffe

18.1 Entscheidungskriterien

Bei Konflikten läßt sich das Standardteam von folgenden Kriterien in Reihenfolge ihrer Wichtigkeit leiten:

1. Korrekte Funktion - bekannte Einschränkungen, Eindeutigkeit.
2. Transportabilität - wiederholbare Ergebnisse, wenn Programme zwischen Standardsystemen portiert werden.
3. Einfachheit
4. Klare, eindeutige Namen - die Benutzung beschreiben - der statt funktionaler Namen, z.B. **[COMPILE]** statt 'c, und **ALLOT** statt **dp+!** .
5. Allgemeinheit
6. Ausführungsgeschwindigkeit
7. Kompaktheit
8. Kompilationsgeschwindigkeit
9. Historische Kontinuität
10. Aussprechbarkeit
11. Verständlichkeit - es muß einfach gelehrt werden können

18.2 Definition der Begriffe

Es werden im allgemeinen die amerikanischen Begriffe beibehalten, es sei denn, der Begriff ist bereits im Deutschen geläufig. Wird ein deutscher Begriff verwendet, so wird in Klammern der engl. Originalbegriff beigefügt; wird der Originalbegriff beibehalten, so wird in Klammern eine möglichst treffende Übersetzung angegeben.

Adresse, Byte (address, byte)
: Eine 16bit Zahl ohne Vorzeichen, die den Ort eines 8bit Bytes im Bereich <0...65.535> angibt. Adressen werden wie Zahlen ohne Vorzeichen manipuliert.
Siehe: "Arithmetik, 2er-komplement"

Adresse, Kompilation (address, compilation)
: Der Zahlenwert, der zur Identifikation eines FORTH Wortes kompiliert wird. Der Adressinterpreter benutzt diesen Wert, um den zu jedem Wort gehörigen Maschinencode aufzufinden.

Adresse, Natürliche (address, native machine)
Die vorgegebene Adressdarstellung der Computerhardware.

Adresse, Parameterfeld (address, parameter field) "pfa"
Die Adresse des ersten Bytes jedes Wortes, das für das Ablegen von Kompilationsadressen (bei :-definitionen) oder numerischen Daten bzw. Textstrings usw. benutzt wird.

anzeigen (display)
Der Prozess, ein oder mehrere Zeichen zum aktuellen Ausgabegerät zu senden. Diese Zeichen werden normalerweise auf einem Monitor angezeigt bzw. auf einem Drucker gedruckt.

Arithmetik, 2er-komplement (arithmetic, two's complement)
Die Arithmetik arbeitet mit Zahlen in 2er-Komplementdarstellung; diese Zahlen sind, je nach Operation, 16bit oder 32bit weit. Addition und Subtraktion von 2er-Komplementzahlen ignorieren Überlaufsituationen. Dadurch ist es möglich, daß die gleichen Operatoren benutzt werden können, gleichgültig, ob man die Zahl mit Vorzeichen (-32,768...32,767 bei 16-Bit) oder ohne Vorzeichen (Ø...65,535 bei 16-Bit) benutzt.

Block (block)
Die 1Ø24 Byte Daten des Massenspeichers, auf die über Blocknummern im Bereich Ø...Anzahl_existenter_Blöcke-1 zugegriffen wird. Die exakte Anzahl der Bytes, die je Zugriff auf den Massenspeicher übertragen werden, und die Übersetzung von Blocknummern in die zugehörige Adresse des Laufwerks und des physikalischen Satzes, sind rechnerabhängig.
Siehe: "Blockpuffer" und "Massenspeicher"

Blockpuffer (block buffer)
Ein 1Ø24 Byte langer Hauptspeicherbereich, in dem ein Block vorübergehend benutzbar ist. Ein Block ist in höchstens einem Blockpuffer enthalten.

Byte (byte)
Eine Einheit von 8bit. Bei Speichern ist es die Speicherkapazität von 8bits.

Kompilation (compilation)

Der Prozess, den Quelltext in eine interne Form umzuwandeln, die später ausgeführt werden kann. Wenn sich das System im Kompilationszustand befindet, werden die Kompilationsadressen von Worten im Dictionary abgelegt, so daß sie später vom Adresseninterpreter ausgeführt werden können. Zahlen werden so kompiliert, daß sie bei Ausführung auf den Stack gelegt werden. Zahlen werden aus dem Quelltext ohne oder mit negativem Vorzeichen akzeptiert und gemäß dem Wert von **BASE** umgewandelt.

Siehe: "Zahl", "Zahlenumwandlung", "Interpreter, Text" und "Zustand"

Definition (Definition)

Siehe: "Wortdefinition"

Dictionary (Wörterbuch)

Eine Struktur von Wortdefinitionen, die im Hauptspeicher des Rechners angelegt ist. Sie ist erweiterbar und wächst in Richtung höherer Speicheradressen. Einträge sind in Vokabularen organisiert, so daß die Benutzung von Synonymen möglich ist, d.h. gleiche Namen können, in verschiedenen Vokabularen enthalten, vollkommen verschiedene Funktionen auslösen.

Siehe: "Suchreihenfolge"

Division, floored (division, floored)

Ganzzahlige Division, bei der der Rest das gleiche Vorzeichen hat wie der Divisor oder gleich Null ist; der Quotient wird gegen die nächstkleinere ganze Zahl gerundet. Bemerkung: Ausgenommen von Fehlern durch Überlauf gilt: `N1 N2 SWAP OVER /MOD ROT * +` ist identisch mit `N1` .

Siehe: "floor, arithmetisch"

Beispiele:	Dividend	Divisor	Rest	Quotient
	10	7	3	1
	-10	7	4	-2
	10	-7	-4	-2
	-10	-7	-3	1

Empfangen receive)

Der Prozess, der darin besteht, ein Zeichen von der aktuellen Eingabeeinheit zu empfangen. Die Anwahl einer Einheit ist rechnerabhängig.

Falsch (false)

Die Zahl Null repräsentiert den "Falschzustand" eines Flags.

Fehlerbedingung (error condition)

Eine Ausnahmesituation, in der ein Systemverhalten erfolgt, das nicht mit der erwarteten Funktion übereinstimmt. Im der Beschreibung der einzelnen Worte sind die möglichen Fehlerbedingungen und das dazugehörige Systemverhalten beschrieben.

Flag (logischer Wert)

Eine Zahl, die eine von zwei möglichen Werten hat, falsch oder wahr.
Siehe: "Falsch" , "Wahr"

Floor, arithmetic

Z sei eine reelle Zahl. Dann ist der Floor von Z die größte ganze Zahl, die kleiner oder gleich Z ist.

Der Floor von +Ø,6 ist Ø
Der Floor von -Ø,4 ist -1

Glossar (glossary)

Eine umgangssprachliche Beschreibung, die die zu einer Wortdefinition gehörende Aktion des Computers beschreibt - die Beschreibung der Semantik des Wortes.

Interpreter, Adressen (interpreter, address)

Die Maschinencodeinstruktionen, die die kompilierten Wort-definitionen ausführen, die aus Kompilationsadressen bestehen.

Interpreter, Text (interpreter, text)

Eine Wortdefinition, die immer wieder einen Wortnamen aus dem Quelltext holt, die zugehörige Kompilationsadresse bestimmt und diese durch den Adressinterpreter ausführen läßt. Quelltext, der als Zahl interpretiert wird, hinter-läßt den entsprechenden Wert auf dem Stack.
Siehe: "Zahlenumwandlung"

Kontrollstrukturen (structure, control)

Eine Gruppe von Worten, die, wenn sie ausgeführt werden, den Programmfluß verändern. Beispiele von Kontrollstrukturen sind:

DO ... LOOP
BEGIN ... WHILE ... REPEAT
IF ... ELSE ... THEN

laden (load)

Das Umschalten des Quelltextes zum Massenspeicher. Dies ist die übliche Methode, dem Dictionary neue Definitionen hinzuzufügen.

Massenspeicher (mass storage)
Speicher, der ausserhalb des durch FORTH adressierbaren Bereiches liegen kann. Auf den Massenspeicher wird in Form von 1024 Byte großen Blökken zugegriffen. Auf einen Block kann innerhalb des FORTH-Adressbereichs in einem Blockpuffer zugegriffen werden. Wenn ein Block als verändert (**UPDATE**) gekennzeichnet ist, wird er letztendlich wieder auf den Massenspeicher zurückgeschrieben.

Programm (program)
Eine vollständige Ablaufbeschreibung in FORTH-Quelltext, um eine bestimmte Funktion zu realisieren.

Quelltext (input stream)
Eine Folge von Zeichen, die dem System zur Bearbeitung durch den Textinterpreter zugeführt wird. Der Quelltext kommt üblicherweise von der aktuellen Eingabeeinheit (über den Texteingabepuffer) oder dem Massenspeicher (über einen Blockpuffer). **BLK** , **>IN** , **TIB** und **#TIB** charakterisieren den Quelltext. Worte, die **BLK** , **>IN** , **TIB** oder **#TIB** benutzen und/oder verändern, sind dafür verantwortlich, die Kontrolle des Quelltextes aufrechtzuerhalten oder wiederherzustellen. Der Quelltext reicht von der Stelle, die durch den Relativzeiger **>IN** angegeben wird, bis zum Ende. Wenn **BLK** Null ist, so befindet sich der Quelltext an der Stelle, die durch **TIB** adressiert wird, und er ist **#TIB** Bytes lang. Wenn **BLK** ungleich Null ist, so ist der Quelltext der Inhalt des Blockpuffers, der durch BLK angegeben ist, und er ist 1024 Byte lang.

Rekursion (recursion)
Der Prozess der direkten oder indirekten Selbstreferenz.

Screen (Bildschirm)
Ein Screen sind Textdaten, die zum Editieren aufbereitet sind. Nach Konvention besteht ein Screen aus 16 Zeilen zu je 64 Zeichen. Die Zeilen werden von 0 bis 15 durchnumeriert. Screens enthalten normalerweise Quelltext. können jedoch auch dazu benutzt werden, um Massenspeicherdaten zu betrachten. Das erste Byte eines Screens ist gleichzeitig das erste Byte eines Massenspeicherblocks; dies ist auch der Anfangspunkt für Quelltextinterpretation während des Ladens eines Blocks.

Suchreihenfolge (search order)

Eine Spezifikation der Reihenfolge, in der ausgewählte Vokabulare im Dictionary durchsucht werden. Die Suchreihenfolge besteht aus einem auswechselbaren und einem festen Teil, wobei der auswechselbare Teil immer als erstes durchsucht wird. Die Ausführung eines Vokabularnamens macht es zum ersten Vokabular in der Suchreihenfolge, wobei das Vokabular, das vorher als erstes durchsucht worden war, verdrängt wird. Auf dieses erste Vokabular folgt, soweit spezifiziert, der feste Teil der Suchreihen-folge, der danach durchsucht wird. Die Ausführung von ALSO übernimmt das Vokabular im auswechselbaren Teil in den festen Teil der Suchreihenfolge. Das Dictionary wird immer dann durchsucht, wenn ein Wort durch seinen Namen aufge-funden werden soll.

stack, data (Datenstapel)

Eine "Zuletzt-rein, Zuerst-raus" (last-in, first-out) Struktur, die aus einzelnen 16bit Daten besteht. Dieser Stack wird hauptsächlich zum Ablegen von Zwischenergebnissen während des Ausführens von Wortdefinitionen benutzt. Daten auf dem Stack können Zahlen, Zeichen, Adressen, Boole'sche Werte usw. sein. Wenn der Begriff "Stapel" oder "Stack" ohne Zusatz benutzt wird, so ist immer der Datenstack gemeint.

stack, return (Rücksprungstapel)

Eine "Zuletzt-rein, Zuerst-raus" Struktur, die hauptsächlich Adressen von Wortdefinitionen enthält, deren Ausführung durch den Adressinterpreter noch nicht beendet ist. Wenn eine Wortdefinition eine andere Wortdefinition aufruft, so wird die Rücksprungadresse auf dem Returnstack abgelegt. Der Returnstack kann zeitweise auch für die Ablage anderer Daten benutzt werden.

String, counted (abgezählte Zeichenkette)

Eine Hintereinanderfolge von 8bit Daten, die im Speicher durch ihre niedrigste Adresse charakterisiert wird. Das Byte an dieser Adresse enthält einen Zahlenwert im Bereich <Ø...255>, der die Anzahl der zu diesem String gehörigen Bytes angibt, die unmittelbar auf das Countbyte folgen. Die Anzahl beinhaltet nicht das Countbyte selber. Counted Strings enthalten normalerweise ASCII-Zeichen.

String, Text (Zeichenkette)

Eine Hintereinanderfolge von 8bit Daten, die im Speicher durch ihre niedrigste Adresse und ihre Länge in Bytes charakterisiert ist. Strings enthalten normalerweise ASCII-Zeichen. Wenn der Begriff "String" alleine oder in Verbindung mit anderen Begriffen benutzt wird, so sind Textstrings gemeint.

Userarea (Benutzerbereich)
Ein Gebiet im Speicher, das zum Ablegen der Uservariablen benutzt wird und für jeden einzelnen Prozeß/Benutzer getrennt vorhanden ist.

Uservariable (Benutzervariable)
Eine Variable, deren Datenbereich sich in der Userarea befindet. Einige Systemvariablen werden in der Userarea gehalten, sodaß die Worte, die diese benutzen, für mehrere Prozesse/Benutzer gleichzeitig verwendbar sind.

Vokabular (vocabulary)
Eine geordnete Liste von Wortdefinitionen. Vokabulare werden vorteilhaft benutzt, um Worte voneinander zu unterscheiden, die gleiche Namen haben (Synonyme). In einem Vokabular können mehrere Definitionen mit dem gleichen Namen existieren; diesen Vorgang nennt man redefinieren. Wird das Vokabular nach einem Namen durchsucht, so wird die jüngste Redefinition gefunden.

Vokabular, Kompilation (vocabulary, compilation)
Das Vokabular, in das neue Wortdefinitionen eingetragen werden.

Wahr (true)
Ein Wert, der nicht Null ist, wird als "wahr" interpre-tiert. Wahrheits-werte, die von Standard-FORTH-Worten errechnet werden, sind 16bit Zahlen, bei denen alle 16 Stellen auf "1" gesetzt sind, so daß diese zum Maskieren benutzt werden können.

Wort, Definierendes (defining word)
Ein Wort, das bei Ausführung einen neuen Dictionary-Eintrag im Kompi-lationsvokabular erzeugt. Der Name des neuen Wortes wird dem Quelltext entnommen. Wenn der Quelltext erschöpft ist, bevor der neue Name er-zeugt wurde, so wird die Fehlermeldung "ungültiger Name" ausgegeben. Beispiele von definierenden Worten sind: : CONSTANT CREATE

Wort, immediate (immediate word)
Ein Wort, das ausgeführt wird, wenn es während der Kompilation oder Interpretation aufgefunden wird. Immediate Worte behandeln Sondersitua-tionen während der Kompilation.
Siehe z.B. IF LITERAL ." usw.

Wortdefinition (word definition)

Eine mit einem Namen versehene, ausführbare FORTH-Proze-dur, die ins Dictionary kompiliert wurde. Sie kann durch Maschinencode, als eine Folge von Kompilationsadressen oder durch sonstige kompilierte Worte spezifiziert sein. Wenn der Begriff "Wort" ohne Zusatz benutzt wird, so ist im allgemeinen eine Wortdefinition gemeint.

Wortname (word name)

Der Name einer Wortdefinition. Wortnamen sind maximal 31 Zeichen lang und enthalten kein Leerzeichen. Haben zwei Definitionen verschiedene Namen innerhalb desselben Voka-bulars, so sind sie eindeutig auffindbar, wenn das Vokabular durchsucht wird.

Siehe: "Vokabular"

Zahl (number)

Wenn Werte innerhalb eines größeren Feldes existieren, so sind die höherwertigen Bits auf Null gesetzt. 16bit Zahlen sind im Speicher so abgelegt, dass sie in zwei benachbar-ten Byteadressen enthalten sind. Die Bytereihenfolge ist rechnerabhängig. Doppeltgenaue Zahlen (32bit) werden auf dem Stack so abgelegt, daß die höherwertigen 16bit (mit dem Vorzeichenbit) oben liegen. Die Adresse der niederwer-tigen 16bit ist um zwei größer als die Adresse der höherwertigen 16bit, wenn die Zahl im Speicher abgelegt ist.

Siehe: "Arithmetik, 2er-komplement" und "Zahlentypen"

Zahlenausgabe, bildhaft (pictured numeric output)

Durch die Benutzung elementarer Worte für die Zahlenaus-gabe (z.B. **<# # #s #>**) werden Zahlenwerte in Textstrings umgewandelt. Diese Definitionen werden in einer Folge benutzt, die ein symbolisches Bild des gewünschten Ausga-beformates darstellen. Die Umwandlung schreitet von der niedrigstwertigen zur höchstwertigen Ziffer fort und die umgewandelten Zeichen werden von höheren gegen niedrigere Speicheradressen abgelegt.

Zahlenausgabe, freiformatiert (free field format)

Zahlen werden in Abhängigkeit von BASE umgewandelt und ohne führende Nullen, aber mit einem folgenden Leerzei-chen, angezeigt. Die Anzahl von Stellen, die angezeigt werden, ist die Minimalanzahl von Stellen - mindestens eine - die notwendig sind, um die Zahl eindeutig darzustellen.

Siehe: "Zahlenumwandlung"

Zahlentypen (number types)

Alle Zahlentypen bestehen aus einer spezifischen Anzahl von Bits. Zahlen mit oder ohne Vorzeichen bestehen aus bewerteten Bits. Bewertete Bits innerhalb einer Zahl haben den Zahlenwert einer Zweierpotenz, wobei das am weitesten rechts stehende Bit (das niedrigstwertige) einen Wert von zwei hoch null hat. Diese Bewertung setzt sich bis zum am weitesten links stehenden Bit fort, wobei sich die Bewertung für jedes Bit um eine Zweierpotenz erhöht. Für eine Zahl ohne Vorzeichen ist das am weitesten links stehende Bit in diese Bewertung eingeschlossen, sodaß für eine solche 16bit Zahl das ganz linke Bit den Wert 32768 hat. Für Zahlen mit Vorzeichen wird die Bewertung des ganz linken Bits negiert, sodaß es bei einer 16bit Zahl den Wert -32768 hat. Diese Art der Wertung für Zahlen mit Vorzeichen wird 2er-komplementdarstellung genannt.Nicht spezifizierte, bewertete Zahlen sind mit oder ohne Vorzeichen; der Programmkontext bestimmt, wie die Zahl zu interpretieren ist.

Zahlenumwandlung (number conversion)

Zahlen werden intern als Binärzahlen geführt und extern durch graphische Zeichen des ASCII Zeichensatzes darge-stellt. Die Umwandlung zwischen der internen und externen Form wird unter Beachtung des Wertes von **BASE** durchge-führt, um die Ziffern einer Zahl zu bestimmen. Eine Ziffer liegt im Bereich von Null bis **BASE**-1. Die Ziffer mit dem Wert Null wird durch das ASCII-zeichen "Ø" (Position 3/Ø, dezimalwert 48) dargestellt. Diese Zifferndarstellung geht den ASCII-code weiter aufwärts bis zum Zeichen "9", das dem dezimalen Wert neun entspricht. Werte, die jenseits von neun liegen, werden durch die ASCII-zeichen beginnend mit "A", entsprechend dem Wert zehn usw. bis zum ASCII-zeichen "~", entsprechend einundsiebzig, dargestellt.Bei einer negativen Zahl wird das ASCII-Zeichen "-" den Ziffern vorangestellt. Bei der Zahleneingabe kann der aktuelle Wert von **BASE** für die gerade umzuwandelnde Zahl dadurch umgangen werden, daß den Ziffern ein "Zahlenbasisprefix" vorangestellt wird. Dabei wird durch das Zeichen "%" die Basis vorübergehend auf den Wert zwei gesetzt, durch "&" auf den Wert zehn und durch "$" oder "h" auf den Wert sechzehn. Bei negativen Zahlen folgt das Zahlenbasisprefix dem Minuszeichen. Enthält die Zahl ein Komma oder einen Punkt, so wird sie als 32bit Zahl umgewandelt.

Zeichen (character)

Eine 8bit Zahl, deren Bedeutung durch den ASCII-Standard festgelegt ist. Wenn es in einem größeren Feld gespeichert ist, so sind die höherwertigen Bits auf Null gesetzt.

Zustand (mode)

Der Textinterpreter kann sich in zwei möglichen Zuständen befinden: dem interpretierenden oder dem kompilierenden Zustand. Die Variable STATUS wird vom System entsprechend gesetzt, und zwar enthält sie bei der Interpretation eine False-flag, bei der Kompilation eine True-flag.
Siehe: "Interpreter, Text" und "Kompilation"

Indexverzeichnis