bigFORTH+MINOΣ
Dokumentation

Bernd Paysan

Einleitung

1. "Real Programmers don't Read Manuals"

Die Lobreden auf das Produkt, die häufig an dieser Stelle stehen, wollen wir uns ersparen, Sie haben sich ja aus gutem Grunde für bigFORTH entschieden. Ebenso selbstverständlich ist, daß Sie natürlich gleich loslegen wollen. Tun Sie sich also keinen Zwang an, starten Sie BIGFORTH.PRG mit einem Doppelklick und spielen Sie einfach ein bißchen mit dem Programm. Falls Sie dann noch Fragen haben, lassen die sich sicherlich durch Lektüre dieses Handbuchs klären.

Unsichere Anwender sollten sich auf alle Fälle das Handbuch vornehmen. Es wurde bewußt so aufgebaut, daß sich sowohl Einsteiger als auch Profis darin zurechtfinden. Da bigFORTH ein sehr komplexes System ist, werden auch FORTH-Experten aus der Lektüre Vorteile ziehen.

Deshalb empfehlen wir, die Dokumentation vollständig durchzulesen und die Beispiele dabei auszuprobieren. Falls dann noch Fragen sind, lesen Sie alles noch einmal in Ruhe durch, meistens sind dann alle Unklarheiten beseitigt.

Es wird vorausgesetzt, daß Sie den ST zumindest soweit beherrschen, wie es nach dem Studium des Atari-Handbuchs möglich ist.

Dieses Handbuch kann und will kein allgemeines FORTH-Buch ersetzen. Kapitel 3 enthält einen knapp gehaltener FORTH-Kurs; die Lektüre sei vor allem Anfängern ans Herz gelegt.

2. Entstehungsgeschichte

bigFORTH basiert auf einem 32-Bit-Ansatz der volksFORTH-83-Autoren. Man kann wohl schon erahnen, woher das System kommt, und welchen Weg es gegangen ist. volksFORTH-83 ist ein Public Domain System, das es schon zu den Anfangszeiten des STs gegeben hat. Leider ist es nur eine 16-Bit-Implementation mit den damit verbundenen Einschränkungen, wie dem nur 64 KByte großen direkt ansprechbaren Adreßraum. volksFORTH wiederum basiert auf dem Perry/Laxen-F83 von der FORTH Interest Group. Damit ist eine weitgehende Erfüllung des FORTH-83-Standards gewährleistet.

Die volksFORTH-Autoren haben Ende 1987 angefangen, das System auf 32 Bit umzuschreiben und den Compiler darauf umzustellen, echten 68000er-Maschinencode zu erzeugen. An einige engagierte volksFORTH-User wurden auch schon lauffähige Prototypen verteilt. Das System hieß damals turboFORTH und sollte auch kommerziell vertrieben werden, aber die Sache verlief dann leider im Sande.

Allerdings hat einer dieser engagierten volksFORTH-User den Prototyp weiterentwikkelt, so können wir Ihnen nach zwei Jahren intensiver Entwicklungsarbeit "bigFORTH" vorstellen.

3. Danksagungen

An dieser Stelle möchte ich den volksFORTH-Autoren D. Weineck, G. Rehfeld und K. Schleisiek danken, denn ohne ihre Vorarbeit und ohne ihr volksFORTH gäbe es bigFORTH gar nicht. Insbesondere danke ich dabei B. Pennemann, dessen Unterstützung

alle noch offene Fragen (vor allem bezüglich des Vertriebes) geklärt hat und dessen Anregungen die Fertigstellung des Systems erst möglich machten. Zudem stammt von ihm (aus seiner Liste mit Vorschlägen) der Name. Außerdem sei auch den β-Testern gedankt (vor allem B. Forstbauer), deren Anregungen sehr wertvoll waren und N. Heusler, der so freundlich war, das Handbuch zu redigieren.

4. Zielsetzung

FORTH ist traditionell eine Public Domain-Sprache, die nicht wie COBOL, Fortran oder C von der Industrie, wie ADA vom DoD (Department of Defence, das amerikanische Verteidigungsministerium) oder wie Pascal bzw. Modula II von den Universitäten getragen wird. FORTHs Entwicklung liegt hauptsächlich in den Händen engagierter Anwender, die die Systeme in ihrer Freizeit entwickeln und selten genügend Zeit und Geduld aufbringen, etwas mit kommerziellen Systemen vergleichbares auf die Beine zu stellen.

bigFORTH soll diese Lücke auffüllen und eine Entwicklungsumgebung zur Verfügung stellen, die vor großen Problemen nicht kapituliert und die einen Code erzeugt, der mit dem anderer Hochsprachen wie C oder Modula II durchaus konkurrieren kann. Ich hoffe, daß möglichst viele bigFORTH-User die Früchte ihrer Arbeit einem breiteren Publikum zur Verfügung stellen; sei es als Public Domain, oder kommerziell.

5. bigFORTH in Stichpunkten

bigFORTH ist: Ein 32-Bit-FORTH; ein Native-Code-Compiler mit Peephole-Optimierung; ein umfangreiches Entwicklungssystem mit Assembler, Disassembler, Debugger, Decompiler, Editor, Multitasker und Target-Compiler. Libraries: GEM-AES, GEM-VDI, GEMDOS, BIOS, XBIOS, Line-A, Memory Management, Filesystem, Floating Point Arithmetik, High-Level-GEM, Streamfilezugriff und Turtle-Graphic. bigFORTH läuft: Auf allen STs und Kombatiblen mit mindestens 512 KByte RAM und einem doppelseitigen Disketten-Laufwerk.

6. Urheberrecht

7. Haftung und Garantie

Selbstverständlich wurde sowohl bei den Programmen als auch beim Handbuch mit größtmöglicher Sorgfalt gearbeitet. Dennoch lassen sich Fehler nie ganz ausschließen. Wir möchten darauf hinweisen, daß weder eine Garantie für Fehlerfreiheit gegeben wird, noch

eine Haftung für irgendwelche Folgen übernommen werden kann, gleich ob durch Fehler im Handbuch oder der Software verursacht. Für die Mitteilung eventueller Fehler sind wir jederzeit dankbar.

Mir bleibt nur noch, Ihnen viel Vergnügen und Erfolg bei der Arbeit mit diesem komplexen und ausgereiften Entwicklungssystem zu wünschen. Genießen Sie die Vorzüge, die bigFORTH Ihnen bietet!

München, im Mai 1990

Bernd Paysan

Inhaltsverzeichnis

1 Installation

1. Installation auf Linux

BigFORTH kommt in mehren tar-Bällen, von denen nur einer — der Source-tar-Ball — wirklich notwendig ist. Die anderen Teile sind nette Gimmicks, oder im Fall der Dokumentation, etwas zum Lesen für Sie.

Das Packet ist in sechs Teile aufgeteilt. Sie brauchen die Sourcen und die Dokumentation um zu starten, die anderen Teile sind entweder optional oder Komfort. Alle Packete werden vom selben Verzeichnis entpackt, sie landen alle im Unterverzeichnis `bigforth`. Wechseln Sie mit `cd` dorthin., und geben Sie `make` ein, um den Rest zu generieren. `make install` installiert bigFORTH systemweit. Das configure-Script kann man mit `./configure --prefix=`⟨***your path***⟩ dazu bringen, bigFORTH in einem anderen Pfad zu installieren. Es ist eine gute Idee, die Datei `xbigforth.cnf` ins Home-Verzeichnis zu kopieren, und die Editor-ID zu verändern. Es gibt da auch noch anderes zu verändern, und irgendwann gibt es sicher einen Dialog, der diese Optionen bequem editiert.

Source `bigforth-`⟨***version***⟩`.tar.bz2`: Das ist ausreichend für eine minimale Installation, und auch für häufigere Updates. Einfach über die alten Sourcen auspacken, und `make` eingeben, um die neuen Sachen zu kompilieren (`make install` zum Installieren).

Pattern `bigforth-pattern-`⟨***version***⟩`.tar.bz2`: Nette Pattern-Pixmaps. Man braucht sie nicht jedesmal zu holen, da sie sich selten ändern.

Wood style `bigforth-edata-wood-`⟨***version***⟩`.tar.bz2`: Icons und Povray-Sourcen für den "wood" Enlightenment Stil. Wer nicht vorhat, diesen Stil zu benutzen, braucht das nicht.

ShinyMetal style `bigforth-edata-ShinyMetal-`⟨***version***⟩`.tar.bz2`: Icons für den ShinyMetal Enlightenment Stil. Wer nicht vorhat, diesen Stil zu benutzen, braucht das nicht.

2. Installation auf Windows

Für diejenigen, die sich noch nicht dazu entschlossen haben, auf Linux zu wechseln: Die Windows-Version ist ein selbst-installierendes Program, `bigforth-`⟨***version***⟩`.exe`, und enthält alles, was man auf Windows braucht. Installiert sich mit einem Doppelklick (keine Angst: es gibt keine Registry-Einträge). Bitte deinstallieren, bevor die nächste Version installiert wird.

Während der Installation werden Teile des Systems compiliert; das reduziert die Download-Zeit. Außerdem werden die Konfigurationsdateien `bigforth.cnf` und `xbigforth.cnf` modifiziert. Hier gibt's einen Dialog, mit dem man den Suchpfad für bigFORTH Sourcen und die Editor-ID ändern kann.

3. Editor-Kommandos

Es gibt verschiedene Wege, in den Editor zu kommen. Der einfachste ist `ed` ⟨***file***⟩ `[`⟨***screen/line***⟩ `[`⟨***cursor***⟩`]]`. Man kann das Kommando aufteilen in `use` ⟨***file***⟩, und dann den Editor mit `v` an der aktuellen Position öffnen, oder an einem spezifischen Screen oder Zeile mit ⟨***screen/line***⟩ `l`.

4. Notation von Befehlen

Befehle können in bigFORTH groß oder klein geschrieben werden, das System macht keinen Unterschied. Bei Dateinamen ist darauf zu achten, daß Linux einen Unterschied macht, Windows aber nicht.

Abgegrenzt werden Befehle durch Leerzeichen. Andere Zeichen wirken nur in besonderen Situationen als Abgrenzung, dann kann zwar auf das Leerzeichen verzichtet werden, es ist aber schlechter Stil. Führende Leerzeichen werden vor Befehlen überlesen.

Wundern Sie sich nicht, wenn ein Befehl mit einer Klammer beginnt, die nicht geschlossen wird, oder mit einem Anführungszeichen endet. Diese Zeichen gehören zum Wort, sie haben in der FORTH-Terminologie eine besondere Bedeutung.

Im Handbuch werden Befehle teilweise in einer abgewandelten BNF (Backus Naur Form) notiert, vor allem Direkteingaben, weil diese Notation ziemlich flexibel ist:

Teile in spitzen Klammern (⟨ und ⟩) werden sinngemäß ersetzt;

Teile in eckigen Klammern können weggelassen werden, so bedeutet z. B. "DIR [⟨*Directory*⟩]", daß man das gewünschte Directory (z. B. A:\GEM) angeben kann, dann lautet der Befehl "DIR A:\GEM", daß man es auch weglassen kann, dann bewirkt DIR aber etwas anderes (gibt das aktuelle Directory aus).

Mehrere Möglichkeiten werden durch einen senkrechten Strich "|" abgetrennt.

Teile in geschweiften Klammern können beliebig oft wiederholt (oder weggelassen) werden, z. B. PATH ⟨*Pfad*⟩;⟨*Pfad*⟩

Ansonsten wird die FORTH-übliche Notation verwendet:

Befehl::= ⟨*Name*⟩ (⟨*In*⟩ −− ⟨*Out*⟩) (⟨*Stackname*⟩ ⟨*In*⟩ −− ⟨*Out*⟩) ⟨*Inputstring*⟩[⟨*Begrenzer*⟩][immediate][restrict][:⟨*Befehl*⟩]

Stackname::=RS|VS|FS|$S

In::= ⟨*Parameter*⟩ / ⟨*Parameter*⟩

Out::= ⟨*Parameter*⟩ / ⟨*Parameter*⟩

Der Inputstring wird von einem Leerzeichen begrenzt, wenn keine andere Angabe gemacht wird. Dann dürfen auch beliebig viele Leerzeichen zwischen Befehl und Inputstring liegen. Ist ein Begrenzerzeichen angegeben, so trennt nur ein Leerzeichen Name und String, alle weiteren Leerzeichen gehören bereits zum Inputstring.

Hat das Wort einen Effekt auf einen anderen Stack als den Parameterstack, so wird dieser Stackeffekt in (einer) weiteren Klammer(n) angezeigt. Die Klammern enthalten dann auch den Namen des Stacks: RS=Returnstack; VS=Vocabulary Stack; FS=Floating Point Stack; $S=String Stack.

Die Eigenschaften immediate und restrict werden zum Schluß angegeben.

Erzeugt der Befehl einen weiteren (Defining Word), so steht nach einem Doppelpunkt die Notation für diesen weiteren Befehl.

Bei den Parametern schreibt man den letzten Parameter auf dem Stack (den Top of Stack) ganz rechts und die anderen links davon, also auch in der Reihenfolge, in der sie übergeben werden. Parameter mit der selben Nummer oder großgeschriebene mit dem

selben Namen sind identisch, werden also durch den Befehl nicht verändert. Alternativen in der Parameterliste werden durch einen Slash "/" abgetrennt. Es bedeutet:

n	⟶ 32-Bit-Zahl
d	⟶ 64-Bit-Zahl, wird durch ein Paar 32-Bit-Zahlen gebildet.
flag	⟶ true (-1) oder false (0).
f	⟶ false
t	⟶ true
8b	⟶ Zeichen (nur 8 Bit werden ausgewertet)
16b	⟶ Nur 16 Bit werden ausgewertet
xxb	⟶ Entsprechend werden xx Bits ausgewertet
/	⟶ Steht zwischen Alternativen, so bedeutet (.. −− n t / f), daß entweder eine Zahl (n) und **t**rue auf dem Stack liegt, oder **f**alse.
n1 n2 .. nx x	⟶ Die Punkte ersetzen eine nicht genau bestimmbare Anzahl Stackparameter, in diesem Fall liegen x Werte auf dem Stack und x selbst, damit das Wort auch auswerten kann, wieviele Parameter auf dem Stack liegen.
u	⟶ Vorzeichenlos (unsigned), auch als Prefix, "ud" bedeutet vorzeichenlose 64-Bit-Zahl

Andere Namen (z. B. "addr" oder "count") bezeichnen das Stackelement nach seiner Funktion. Meistens handelt es sich dann um eine 32-Bit-Zahl. Grundsätzlich führen zusätzliche Bits zu keiner Fehlfunktion. Bei doppelt genauen Zahlen ist zu beachten, daß sie aus zwei Stackelementen gebildet werden.

Der Inputstring wird, wenn nötig, in der oben beschriebenen abgewandelten BNF dargestellt.

Das mag nun ziemlich kompliziert und formalistisch wirken, viele Beispiele dazu finden Sie ab Kapitel 4. Sie werden noch sehen, wieviel Ihnen diese Informationen geben.

Direkt einzugebende Befehle erkennen Sie im Handbuch an der geänderten Schriftart, sie werden unproportional und unterstrichen dargestellt:

`Direkteingabe`

5. Zahleneingaben

FORTH versucht zuerst, ein eingegebenes Wort zwischen zwei Leerzeichen als Befehl zu interpretieren. Schlägt dies fehl, wird versucht, es als Zahl aufzufassen. Erst wenn auch dies scheitert, wird eine Fehlermeldung ausgegeben.

Reine Ziffernfolgen werden in 32-Bit-Zahlen umgewandelt. Dabei wird die aktuelle Zahlenbasis benutzt, um die Wertigkeit der Stellen zu ermitteln. Zugelassen sind nur Ziffern bis Basis-1. Ziffern mit einem Wert von 10 (dezimal) oder mehr werden durch Buchstaben von A aufwärts dargestellt.

Setzt man vor die Ziffernfolge ein "%", "&" oder "$", so kann man während der Umwandlung eine andere Zahlenbasis wählen: "%" für binär, "&" für dezimal und "$" für hexadezimal.

Negative Zahlen beginnen mit einen "-". Es wird dann nach der Umwandlung das Zweierkomplement (Negation) gebildet.

Zahlen, die entweder einen Punkt oder ein Komma enthalten, werden als doppelt genaue Zahlen (64 Bit) interpretiert. Dabei darf zwischen zwei Ziffern oder am Ende höchstens ein Zeichen (Punkt oder Komma) stehen.

Format in BNF:

Zahl::= [-][%|&|$]⟨***Ziffer***⟩[,|.]⟨***Ziffer***⟩[,|.]

Da gültige Ziffern von der Basis abhängen, können sie nicht einfach mit BNF dargestellt werden.

6. Notation von Sondertasten

Die Cursortasten werden als Pfeile in die entsprechende Richtung (⟨←⟩, ⟨→⟩, ⟨↑⟩ und ⟨↓⟩) dargestellt. Tasten, die in Verbindung mit der Control-Taste gedrückt werden müssen, haben ein "^"-Zeichen davor, ⟨Ctrl⟩⟨N⟩ bedeutet also, daß Sie erst die Control-Taste und dann (ohne Control loszulassen) ⟨N⟩ drücken. Vor Zeichen, die mit der Shift-Taste gedrückt werden, steht vorne ein ⟨⇧⟩◊. ⟨⇧⟩⟨UNDO⟩ bedeutet demnach: Gleichzeitig Shift-Taste und UNDO-Taste drücken. Vor Zeichen, die zusammen mit der Alternate-Taste gedrückt werden, steht ⟨Alt⟩◊.

Die weiteren Tasten haben folgende Bezeichnung:

⟨Esc⟩ : Esc-Taste
⟨RET⟩ : Return- oder Enter-Taste
⟨TAB⟩ : Tab-Taste
⟨DEL⟩ : Delete-Taste
⟨BS⟩ : Backspace-Taste
⟨UNDO⟩ : Undo-Taste
⟨HOME⟩ : Clr Home-Taste
⟨HELP⟩ : Help-Taste
⟨F1⟩-⟨F10⟩ : Funktionstasten

2 Tutorial

1. Starten des Systems

Das üblicherweise gestartete Programm ist `xbigforth`, das die GUI-Library enthält. `bigforth` nutzt das Text-Terminal als Benutzerinterface, und `forthker` ist der spartanische Kernel.

bigFORTH besteht aus zwei Teilen, einem ausführbaren Loader, und einer Image-Datei. Diese Datei hat normalerweise den gleichen Basis-Namen wie der Loader, und die Endung `.fi`. Normalerweise wird bigFORTH wie folgt aufgerufen:

```
xbigforth [file | -e forth-code] ...
```

Dabei werden die Dateien und der Forth-Code in der angegeben Reihenfolge interpretiert. Ich habe versucht, bigFORTH soweit sinnvoll mit Gforth kompatibel zu halten.

Grundsätzlich sieht die Kommandozeile wie folgt aus:

```
[x]bigforth [loader options] [image options]
```

Die Loader-Optionen müssen vor dem Rest der Kommandozeile stehen. Sie sind:

`--image-file` ⟨***file***⟩
`-i` ⟨***file***⟩ Lädt das Forth-Image ⟨***file***⟩ anstatt des Defaults ⟨***loader***⟩`.fi`.

`--dictionary-size` ⟨***size***⟩
`-d` ⟨***size***⟩ Belegt ⟨***size***⟩ Bytes für das Forth-Dictionary (alle gerade offenen Module) statt dem Default, der im Modul selbst spezifiziert ist (typisch 256k). Die ⟨***size***⟩ Spezifikation dafür und für folgende Optionen besteht aus einer Zahl und einer Einheit (z. B. `4M`). Die Einheit kann eine von `b` (bytes), `k` (kilobytes), `M` (Megabytes) und `G` (Gigabytes) sein. Wenn keine Unit angegeben ist, wird `b` verwendet.

`--mem-size` ⟨***size***⟩
`-m` ⟨***size***⟩ Belegt ⟨***size***⟩ Bytes für die dynamische Speicherverwaltung (typisch 16M).

`--stack-size` ⟨***size***⟩
`-s` ⟨***size***⟩ Belegt ⟨***size***⟩ Bytes für beide Stacks zusammen (typisch 64k).

`--verbose`
`-v` Erhöht die Geschwätzigkeit des Loaders (Debugging).

Nachdem es gestartet wurde, begrüßt bigFORTH den User mit einer Meldung in der zweiten Zeile:

```
ANS bigFORTH 386-Linux rev. x.xx
```

wobei x.xx die Versionsnummer ist, und "ANS" anzeigt, daß es sich um ein American National Standard Forth handelt.

2. Verlassen des Systems

Geben Sie BYE⟨ret⟩ ein, um das System zu verlassen.

3. Dateiendungen

Dateiendungen sind in der Regel nur Konventionen. Die Endungen werden von mir so genutzt:

`.fs` Forth Sourcen als Stream, können auch mit anderen Editoren bearbeitet werden.

`.fb` Forth Sourcen als Blöcke, können normal nur mit dem Forth-eigenen Editor bearbeitet werden (Der Gforth-Mode von Emacs versteht auch Blöcke).

`.fi` Forth Image, wird vom Loader geladen und mit SAVESYSTEM gespeichert.

`.m` MINOS-Sourcen, für den GUI-Editor Theseus.

4. Der Zeileneditor

- Sie können Text im Einfügemodus eingeben.
- Mit den Cursortasten ⟨←⟩ und ⟨→⟩ wandern Sie in der Zeile nach links und rechts.
- Mit ⟨BS⟩ löschen Sie einzelne Zeichen links vom Cursor, der Cursor geht dann eine Schreibstelle nach links und zieht den Rest der Zeile nach.
- Mit ⟨DEL⟩ löschen Sie das Zeichen unter dem Cursor. Der Rest der Zeile rückt nach, der Cursor bleibt an seiner Position.
- Vor der Eingabe einer Zeile können Sie sich mit der ⟨UNDO⟩-Taste die vorherige Zeile in den Puffer holen, sie ggf. modifizieren und erneut ausführen. ⟨UNDO⟩ funktioniert nur, wenn der Cursor ganz links steht, egal ob Sie vorher etwas eingegeben haben, oder nicht. Durch eine Eingabe kann die geänderte Stelle nicht wieder in den ursprünglichen Zustand versetzt werden, es erscheint dort der neue Text.
- Eine Eingabe wird durch Drücken von ⟨RET⟩ (Return- oder Enter-Taste) beendet. Der FORTH-Interpreter führt dann diese Zeile aus, schreibt dabei eventuell etwas auf den Schirm und signalisiert zuletzt mit einem " ok", daß alles ordnungsgemäß ausgeführt wurde. Ist der Compiler eingeschaltet, wird statt " ok" " compiling" ausgegeben.
- Die Funktionstasten ("Hot Keys") sind mit einigen häufig benötigten Befehlen vorbelegt. Diese Erweiterung ist keine Kernelfunktion, läuft also nur mit BIGFORTH.PRG und noch nicht mit FORTHKER.PRG.

  Die Belegung:

  ⟨F1⟩: .S

  ⟨F2⟩: ORDER

  ⟨F3⟩: WORDS

  ⟨F4⟩: FILE?

  ⟨F5⟩: FILES

  ⟨F6⟩: DIR

  ⟨F7⟩: PATH

⟨F8⟩: FREE?

⟨F9⟩: nicht belegt

⟨F10⟩: V

Bei Fehleingaben (oder bei internen Fehlern, z. B. Meldungen des Betriebssystems) erscheint eine entsprechende Meldung.

5. Fehlermeldungen

Sollte die Eingabezeile nicht korrekt ausführbar sein, so erscheint eine Fehlermeldung. Meistens wird ein Tippfehler Ursache der Fehlermeldung sein. Es erscheint dann eine Alertbox mit dem fehlerhaften Wort und ein "Hä?" in der nächsten Zeile. In den darauffolgenden zwei Zeilen wird versucht, den Weg zurückzuverfolgen, über den die Fehlermeldung aufgerufen wurde. Dies wird mit "Ebene:" bezeichnet. Beispiel: Geben Sie

hallo⟨RET⟩ `don't know hallo`

(und ⟨RET⟩) ein. Es erscheint eine Fehlermeldung.

Die Ebenen sind nützlich, wenn man verfolgen will, welches Wort die Fehlermeldung ausgelöst hat. QUIT und (QUIT (Klammer gehört zum Wort!) sind die Hauptschleife zur Befehlsausführung des FORTH-Systems. Der eigentliche Verursacher, der Kommandointerpreter, ist ein unsichtbares Wort und wird nur durch ein Fragezeichen dargestellt. NO.EXTENSIONS gibt die Fehlermeldung dann aus. Da Alertboxen nur maximal 30 Zeichen pro Zeile haben dürfen, wird im zweiten QUIT umgebrochen, der Einfachheit halber direkt am Ende der Zeile.

Diese Alertbox läßt sich nun mit einem Mausklick auf den OK-Knopf oder mit RET quittieren. In FORTHKER.PRG wird übrigens nur das Wort, dessen Ausführung (bzw. der Versuch, es auszuführen) den Fehler hervorrief, und die Fehlermeldung selbst (Hier: "Hä?") ausgegeben, es erscheint keine Alertbox.

Falls der Fehler beim Laden einer Datei auftrat, wird in der letzten Zeile der Alertbox (oder hinter der Fehlermeldung in FORTHKER.PRG) noch "Blk " und die Nummer des Blocks, in dem der Fehler auftrat, ausgegeben. Statt dem OK-Knopf gibt es zwei, einer mit "Editor", der andere mit "Cancel" beschriftet. Wie der Name vermuten läßt, führt der "Editor"-Knopf in den Editor, der Cursor steht hinter dem nicht ausführbaren Wort. Mit "Cancel" können Sie weiterarbeiten, ohne den Fehler zu korrigieren. Falls Sie sich es anders überlegen, können Sie jederzeit mit V an dieselbe Stelle im Editor gelangen, die Sie mit dem "Editor"-Knopf erreicht hätten. Auch mit RET kommen Sie von der Alertbox in den Editor.

Die gängigsten Fehler sind:

Hä?: Dieses Wort ist nicht definiert. Oder: Diese Zahl läßt sich nicht umwandeln. Häufigste Ursache sind Tippfehler oder ein Wort, das einem Vokabular definiert ist, auf das im Moment nicht zugegriffen werden kann.

unstructured: Das Programm ist nicht wohlstrukturiert. Es steht ein THEN ohne ein IF oder umgekehrt, ein BEGIN wird nicht von einem REPEAT oder UNTIL beendet, oder es fehlt eben das BEGIN. FORTH sagt hier nicht, was fehlt, sondern welches Strukturwort als erstes nicht richtig stand.

compile only: Dieses Wort darf nur compiliert werden, es kann nicht im Interpreter ausgeführt werden. Beispiel: Returnstackmanipulationen, Strukturwörter.

Stack empty: Der Stack ist leer, d. h. das zuletzt aufgerufene Wort hat mehr Stackelemente verbraucht, als vorhanden waren.

6. Der Editor

Sourcecodes schreiben Sie mit dem Editor. Es besteht zwar die Möglichkeit, einen Fremdeditor wie Tempus zu verwenden, aufgrund des FORTH-eigenen Formats ist es aber nicht möglich, die mitgelieferten Quelltexte ohne Umwandlung in diesen Editor einzulesen. Siehe dazu Kapitel 17..

Die Massenspeicherverwaltung wurde in FORTH anders gelöst als bei anderen Sprachen. So geht FORTH von blockorientierten Massenspeichern aus. Diese Vermutung erweist sich auch physikalisch als richtig: Massenspeicher sind in der Tat blockorientiert. Die meisten anderen Systeme betrachten Massenspeicher buchstaben-, wort- oder zeilenorientiert.

Also werden die Zeichen der Reihe nach in die Datei geschrieben und an den Zeilenenden steht ein (oder zwei) Umbruchzeichen. Der Compiler muß sich dann die Zeilen herauspicken.

In FORTH, wie gesagt, läuft das nicht so. Ein Block umfaßt hier ein KByte (1024 Bytes). Dieser Block wird vom Interpreter/Compiler als eine Zeile betrachtet. Um dem Menschen das Schreiben zu erleichtern, daß sie nicht den Überblick verlieren, teilt man den Block in 16 Zeilen zu je 64 Zeichen auf: Ein “Screen”. Hier kann man zweidimensional arbeiten, nach Lust und Laune formatieren, auch wenn es dem FORTH-System eigentlich egal ist, wie Sie die Wörter auf dem Screen verteilen. Nutzen Sie diese Freiheit aus, um übersichtlich zu programmieren!

Prinzipiell soll ein Screen eine kleine Einheit bilden, in dem einige Befehle (FORTH-Worte) definiert sind, die eng zusammengehören. Die erste Zeile eines Screens ist für einen Titel definiert, in dem nicht nur die Namen der Befehle stehen (bei Libraries kann das sinnvoll sein), sondern der die Definitionen auf einen “gemeinsamen Nenner” bringt.

Der Vorteil dieser Screens ist, daß man beliebig lange Dateien editieren kann, ohne sie ganz im Speicher halten zu müssen. Dieses Konzept der “virtuellen Speicherverwaltung” wird im Kapitel 3.10 noch genauer erklärt. Der unvermeidliche Nachteil liegt einerseits in dem begrenzten Format von 64*16 Zeichen, andererseits wird durch Leerzeilen oder nicht bis zum rechten Rand vollgeschriebenen Zeilen viel Speicherplatz verschwendet. Und drittens macht es einige Schwierigkeiten, eine Zeile einzufügen, wenn ein Screen bereits voll ist.

Leider gibt es sonst keinen Editor, der dieses Format erzeugt (Diskmonitoren natürlich ausgenommen, die sind auch blockorientiert), also bleibt nichts anderes übrig, als den FORTH-eigenen Editor zu benutzen. Den könnte man übrigens auch eingeschränkt als Diskmonitor mißbrauchen.

Gestartet wird der Editor entweder mit V oder mit ⟨*Screen#*⟩ L. Dabei wird die gerade aktuelle Datei editiert.

7. Starten des Editors

Legen Sie die graue Systemdiskette in Laufwerk A. Achten Sie darauf, daß der Schreibschutzschieber geöffnet ist, um die Diskette vor ungewollten Schreibzugriffen zu schützen. Geben Sie ein:

A:␣USE␣FORTH.SCR␣1␣L⟨RET⟩

Die Aufforderung in der Dialogbox, Ihre ID einzugeben ("Enter your ID") quittieren Sie nur mit RET oder einem Klick auf "OK". Das aktuelle Datum und Ihre ID werden vom System übernommen. Nur wenn Sie damit nicht zufrieden sind, können Sie noch Änderungen vornehmen:

Abbildung 2.1: Id eingeben

Den Hinweis über das falsche Datum und Uhrzeit ignorieren Sie zunächst mit "Abbruch", es erscheint dann allerdings ein falsches Datum in der ID (und oben in der rechten Ecke nach wie vor eine falsche Uhrzeit).

Der eigentliche Editoraufruf steckt im Wort L, mit dem man beim angegebenen Screen mit dem Editieren anfängt. Man kann den Editor auch mit V aufrufen und landet dann dort, wo man ihn das letzte Mal verlassen hat oder an der Stelle, an der bei der Compiliation ein Fehler aufgetreten ist.

Woher weiß eigentlich der Editor, daß Sie das Datum nicht gestellt haben? Und warum erscheint diese Feststellung?

Das dient zur Fehlerbehebung. Ich nenne diesen Fehler den "06feb86"-error, weil dies das Datum ist, das bei allen STs älterer Bauart (mit dem allerersten ROM-TOS) nach dem Start eingestellt ist. Dieses Datum taucht dann in allen Screens auf. So läßt sich leicht feststellen, wann der Screen verändert wurde. Man wird dann den Eindruck nicht los, als hätte hier jemand das Programm an einem Tag erschaffen...

Da Atari neue Betriebssystemversionen bekanntlich sehr schnell (und sehr oft) ausliefert, halte ich es für ausgeschlossen, daß jemand eine TOS-Version in kürzerer Zeit als eine Woche nach dem "Erstellungsdatum" in die Hände bekommt und so wird der Benutzer eben dann genervt, wenn diese 7 Tage noch nicht vergangen sind - ein untrüglicher Hinweis auf eine nicht gestellte Uhr. Außerdem werden falsche Daten wie der 31. Februar oder der 1.13. nicht durchgelassen, ebensowenig wie ein 31 Uhr 05 oder ähnliches. Geht das Datum über den Wertebereich hinaus, kann FORTH den Fehler nicht mehr feststellen (35. Mai wird eben zum 3. Juni).

Falls sich bigFORTH Ihrer Meinung nach doch irrt, so können Sie über "Abbruch" in der Alertbox Ihre Datumsvorstellung durchsetzen.

8. Das Fenster

Auf dem Schirm öffnet sich ein Fenster, die Menüzeile erscheint ganz oben. In der Titelzeile des Fensters steht "FORTH.SCR", also die editierte Datei. In der Infozeile darunter steht "Scr # 001 not updated", d. h. Sie befinden sich im Screen 1 und haben noch nichts verändert. Hätten Sie schon etwas eingegeben, stünde da "Scr # 001 updated". Der Cursor (ein nicht blinkendes, schwarzes Rechteck) steht oben in der linken Ecke.

Der untere Schieber zeigt die Position des Screens in der Datei, und das Verhältnis von Screen zu Dateilänge. Hier ist der Schieber quadratisch, d. h. die Datei ist sehr lang. Der Schieber steht ziemlich weit links, also am Dateianfang. Anschaulich kann man sich vorstellen, daß die verschiedenen Screens einer Datei nebeneinander angeordnet sind und das Fenster einen Ausblick auf gerade einen Screen gibt.

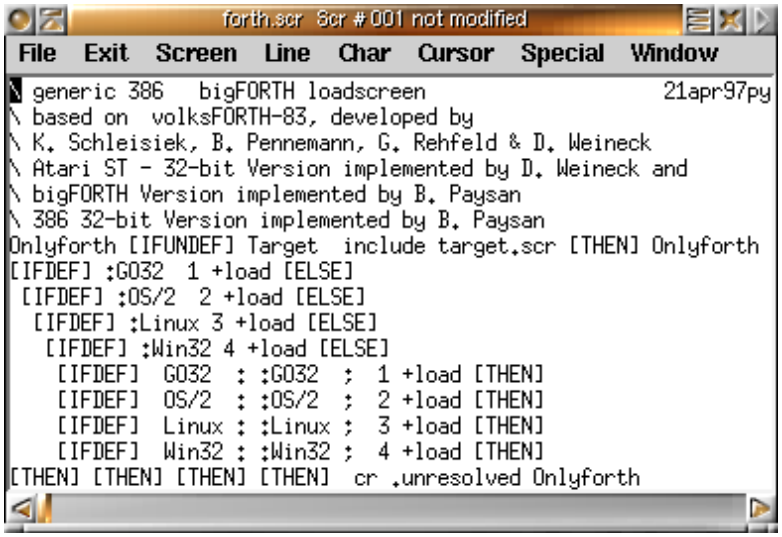


Abbildung 2.2: Editor-Fenster

Sie werden bemerken, daß links vom Schieber noch ein wenig Platz ist, also scheint Screen 1 nicht der erste Screen der Datei zu sein. Klicken Sie den linken Scrollpfeil einmal an. Tatsächlich: Es gibt noch einen Screen 0. In dieser Datei ist hier ein Inhaltsverzeichnis untergebracht. Das ist historisch zu verstehen, deshalb soll hier nicht näher darauf eingegangen werden. Da durch Änderungen auf diesem Screen keine Fehlfunktionen ausgelöst werden, eignet er sich hervorragend zum Üben.

Spielen Sie erstmal noch mit dem Fenster. Verkleinern Sie es, Sie werden sehen, es geht nur senkrecht. Waagrecht bleibt das Fenster immer 64 Zeichen breit. Scrollen Sie senkrecht durch, Sie brauchen dazu nur die Maustaste festzuhalten. Das geht übrigens auch waagrecht, aber kehren Sie nach Ihrer Reise am besten wieder zu Screen 0 zurück.

Sie können das Fenster natürlich auch verschieben, aber nur bis zum Schirmrand.

Klicken Sie einfach an irgendeine Stelle im Fenster. Der Cursor springt an diese Stelle. Der Mauscursor nimmt deshalb auch im Fenster eine andere Form an, die bedeuten soll, daß man einen Cursor positionieren kann. Diese Form entspricht damit zwar den GEM-Normen, ist aber unglücklicherweise im Farbmodus doppelt so hoch wie ein Zeichen. Der Aktionspunkt liegt in der Mitte des senkrechten Strichs, und wer dann immer noch nicht “trifft”, kann auf den Pfeil umschalten (aber dazu später).

9. Die Tastatur

Sie können mit den Cursortasten den Cursor bewegen, einfach drauflostippen, mit ⟨RET⟩ in eine neue Zeile gehen, mit ⟨BS⟩ und ⟨DEL⟩ in der üblichen Weise löschen und mit UNDO einen “verhauten” Screen zurückholen, probieren Sie es nur aus.

Im Moment befindet sich der Editor im Insert-Modus. Eingegebene Zeichen werden eingefügt, die restlichen Zeichen der Zeile weitergeschoben. Dieser Modus ist am angenehmsten und nicht fehlerträchtig. Ansonsten gibt es noch einen Overwrite-Modus, in dem das Zeichen unter dem Cursor einfach überschrieben wird. Hier kann natürlich etwas verloren gehen. Von den Modi sind nur die druckbaren Eingaben betroffen, alles andere funktioniert genau gleich. Der Overwrite-Modus wird mit ⟨Ctrl⟩⟨O⟩ eingeschaltet, zurück in den Insertmodus kommt man mit ⟨Ctrl⟩⟨I⟩.

Mit ⟨BS⟩ wird das Zeichen links vom Cursor gelöscht. Dieser und die restliche Zeile rücken nach links nach. Mit ⟨DEL⟩ dagegen löscht man das Zeichen unter dem Cursor. ⟨BS⟩ läßt sich durch ein ⟨←⟩ und ⟨DEL⟩ ersetzen.

Mit ⟨⇧⟩⟨DEL⟩ löscht man die Zeile, in der man gerade ist, mit ⟨⇧⟩⟨BS⟩ die vorherige (äquivalent zu ⟨BS⟩ und ⟨DEL⟩, nur daß statt Zeichen Zeilen gelöscht werden).

⟨INS⟩ fügt ein Leerzeichen ein, der Cursor bleibt aber an derselben Stelle stehen. ⟨INS⟩ ist die Zusammensetzung aus SPACE und ⟨←⟩. ⟨⇧⟩⟨INS⟩ fügt entsprechend eine ganze Zeile ein. Ein ⟨DEL⟩ hebt ein vorangegangenes ⟨INS⟩ auf, wie ein ⟨BS⟩ eine vorherige Eingabe zurücknimmt.

Mit ⟨Ctrl⟩⟨E⟩ wird die aktuelle Zeile gelöscht, ohne daß der Rest des Bildschirms nachrückt. ⟨Ctrl⟩⟨DEL⟩ löscht ab dem Cursor bis zum rechten Rand des Screens. Mit ⟨RET⟩ gelangt man an den Anfang der nächsten Zeile. ⟨⇧⟩⟨RET⟩ bricht an der Stelle um, an der der Cursor stand und schreibt den Rest der Zeile an den Anfang der nächsten. Der Cursor steht wie bei ⟨RET⟩. ⟨Ctrl⟩⟨RET⟩ dagegen fügt an der Cursorposition 64 Leerzeichen ein, alles rechts vom Cursor wird also eine Zeile nach unten geschoben, auch der Cursor wandert nach unten.

Einen versehentlich editierten Screen setzt man mit ⟨UNDO⟩ in den ursprünglichen, gespeicherten Zustand zurück. Da man auch die UNDO-Taste versehentlich betätigen kann, wird mit ⟨⇧⟩⟨UNDO⟩ dieser (oft fatale) Befehl zurückgenommen, zumindest einmal, und dann nur in dem Screen, den man zuletzt zurückgesetzt hat. ⟨⇧⟩⟨UNDO⟩ funktioniert auch nur, solange der Screen nicht wieder verändert wurde.

Der Cursor läßt sich auch mit ⟨TAB⟩ bewegen. Vorwärts kommt man dabei beim 1., 17., 33. und 49. Zeichen zum Stehen, also in 1/4-Zeilen-Schritten. Rückwärts (mit ⟨⇧⟩⟨TAB⟩) ist alle 1/8-Zeile ein Tabstop. Mit ⟨HOME⟩ wird der Cursor in die linke obere Ecke gesetzt, mit ⟨⇧⟩⟨HOME⟩ an das Textende des Screens.

Um Textteile frei zu verschieben oder zu kopieren gibt es statt der üblichen Textblöcke einen Zeichen- und einen Zeilenstack. Das funktioniert wie folgt: Mit ⟨⇧⟩⟨←⟩ nimmt man ein Zeichen auf den Stack, der Rest der Zeile rückt wie bei ⟨DEL⟩ auf. Das Zeichen ist aber nicht verloren, mit ⟨⇧⟩⟨→⟩ wird es wieder zurückgegeben, auch an einer beliebigen anderen Stelle. Das Zeichen wird dabei an der Cursorposition eingefügt, der Cursor bleibt auf seinem Platz. Mit ⟨Ctrl⟩⟨→⟩ kopiert man das Zeichen in den Puffer, ohne es zu löschen, der Cursor geht dabei eine Stelle weiter. ⟨⇧⟩⟨←⟩ funktioniert also so wie ⟨Ctrl⟩⟨→⟩ und ⟨BS⟩.

Damit man nicht jedes Zeichen einzeln schieben oder kopieren muß, können bis zu 128 Zeichen nacheinander aufgenommen werden. Dabei gilt, daß das zuletzt aufgenommene Zeichen auch zuerst abgegeben wird (LIFO- oder Stack-Prinzip). Ein Wort wird genauso herausgeschoben, wie vorher hinein.

Dasselbe gilt mit ⟨⇧⟩⟨↑⟩ und ⟨⇧⟩⟨↓⟩ bzw. ⟨Ctrl⟩⟨↓⟩ auch für Zeilen. Der Zeilenstack ist nur durch den Speicherplatz zwischen Pad und Stack begrenzt. Der Stack ist übrigens global, d. h. bei der Benutzung mehrerer Fenster kann man über den Stack Text auch in ein anderes Fenster verschieben.

10. Weitere Funktionen

Diese elementaren Funktionen reichen vielleicht aus, wenn man einen Screen editieren will. Um vernünftig zu arbeiten, braucht man noch einige weitere Funktionen. Diese erreicht man zum größten Teil mit der Control-Taste, ein paar wenige mit der Alternate-Taste.

Zuerst die einfachen Befehle, die auf ganze Screens wirken:

Mit ⟨Ctrl⟩⟨N⟩ (N für Next) geht man in den nächsten Screen, mit ⟨Ctrl⟩⟨B⟩ (Back) in den vorherigen. Versucht man, mit ⟨Ctrl⟩⟨N⟩ über das Dateiende hinauszugehen, erscheint zuerst eine Alertbox, die fragt: "Einen Screen anhängen?" Mit ⟨RET⟩ wählen Sie "NEIN", die Dateilänge bleibt beim Alten. Klicken Sie "JA", wird an die Datei ein leerer Screen angehängt, den Sie dann editieren können.

Abbildung 2.3: Nächster Screen

⟨Ctrl⟩⟨J⟩ läßt Sie an einen beliebigen Screen springen. Eine Dialogbox erscheint, in der die Screennummer erfragt wird. Mit ⟨Ctrl⟩⟨A⟩ gelangen Sie zur letzten mit ⟨Ctrl⟩⟨M⟩ gesetzten Marke, die Stelle, die Sie damit verlassen haben, wird zur neuen Marke, Sie können also mit ⟨Ctrl⟩⟨A⟩ wieder zurückspringen.

Abbildung 2.4: Springe zu Screen

Manche Dateien des Systems sind mit Shadowscreens kommentiert. Jedem Source-Screen wird ein Kommentarscreen (Shadowscreen) zugeordnet. Die Datei wird dazu in zwei Teile zerlegt, in der ersten Hälfte steht das eigentliche Programm, in der zweiten die Shadowscreens. Mit ⟨Ctrl⟩⟨W⟩ erreicht man vom Source-Screen den dazugehörigen Shadowscreen, bzw. kommt vom Shadowscreen wieder zurück. Bei Dateien mit ungerader Screenanzahl ist Screen 0 sein eigener Shadowscreen.

Um ein in einer Datei definiertes Wort zu finden, können Sie entweder den Editor mit VIEW ⟨*Wort*⟩ aufrufen, oder im Editor ⟨Ctrl⟩⟨V⟩ tippen und dann das gesuchte Wort eingeben. Falls Sie die Dialogbox nicht über "Cancel" verlassen und die Funktion dadurch abbrechen, können Sie sie entweder (mit ⟨RET⟩) über "Mark" verlassen und die aktuelle Cursorposition als Marke speichern, oder über "OK", dann wird die Marke nicht verändert:

Abbildung 2.5: Springe zu Wort

Natürlich können Sie auch suchen (und ersetzen) lassen. Rufen Sie dazu mit ⟨Ctrl⟩⟨F⟩ eine Dialogbox auf:

Abbildung 2.6: Suche nach Text

Gehen wir die Funktionen zeilenweise durch. In der ersten Zeile können Sie den ersten und letzten Screen wählen, in dem gesucht werden soll. Normalerweise sind das der Screen 1 und der letzte Screen der Datei. Ferner können Sie mit den zwei Pfeilen in der Mitte die Suchrichtung bestimmen. Ist der linke Pfeil selektiert, wird in Richtung Dateianfang gesucht, beim rechten Pfeil entsprechend zum Dateiende. Die Suche wird immer von der aktuellen Cursorposition im aktuellen Screen Richtung Screenende angefangen, und bei Vorwärtssuche höchstens bis zum letzten Screen weitergeführt. Bei Rückwärtssuche entsprechend bis zum ersten Screen, d. h. die Screengrenzen haben nur auf das Ende der Suche einen Einfluß, nicht auf den Anfang.

Im Feld “Case:” können Sie wählen, ob Groß- und Kleinschreibung unterschieden werden soll (Button “Match” selectiert) oder nicht (Button “Ignore”). Daneben können Sie im Feld “Replace:” bestimmen, ob alle (Button “All”) oder nur manche Texte (Button “Some”) ersetzt werden sollen. “Manche” bedeutet hier nicht, daß bigFORTH willkürlich entscheidet, was es ersetzt und was nicht, sondern daß Sie bei jeder Fundstelle selbst wählen dürfen, ob ersetzt werden soll.

Im nächsten Kasten stehen die drei möglichen Ausgänge, “Cancel”, “Find:” und “Replace with:”. “Cancel” bricht den Suchdialog ab. Mit “Find:” wird der im Textfeld daneben stehende Text gesucht. “Find:” ist auch der Default-Ausgang, den man auch über ⟨RET⟩ erreicht. Bei geglückter Suche steht der Cursor hinter dem Suchbegriff. Hinter “Replace with:” können Sie einen Austauschtext eingeben. Verlassen Sie die Dialogbox dazu über den Button “Replace with:”.

Geben Sie nun zum Beispiel für den Suchbegriff “FORTH” ein, und für den Austauschbegriff “BASIC”. Lassen Sie die Optionen so, wie sie bei Systemstart sind, also auf “Case: Ignore” und “Replace: Some”. Im Screen 1 wird das System in “bigFORTH” fündig. Es erscheint eine kleine Dialogbox, die mit einem schwarzen Pfeil auf den Anfang von “FORTH” zeigt und fragt “Replace?”. Zur Auswahl stehen “Yes”, “No” und “Cancel”. Drücken Sie ⟨RET⟩, das entspricht einem Klick auf “Yes”, und aus dem “bigFORTH” wird “bigBASIC”:

Abbildung 2.7: Replace word

Die Dialogbox erscheint sogleich ein paar Zeilen weiter unten und deutet auf das "FORTH" von "volksFORTH". Klicken Sie nun "Nein" an, "volksFORTH" bleibt "volksFORTH" und die Dialogbox deutet schon wieder ein paar Zeilen weiter auf "bigFORTH". Um dem Spuk ein Ende zu bereiten, klicken Sie "Cancel".

Falls es Ihnen Spaß gemacht hat, können Sie ja noch weitere "FORTH"s durch "BASIC"s ("Modula", "C", oder was immer Ihnen gefällt) ersetzen, und wenn Sie genug haben, wieder zurück durch "FORTH". Ist der Ersatz-String länger oder kürzer als der Such-String, so werden die folgenden Zeichen verschoben.

Wird über mehrere Screens hinweg gesucht, so steht der horizontale Schieber des Fensters immer an der Position des Screens, der gerade durchsucht wird. Wird ein Wort gefunden, so wird der betreffende Screen ganz aufgebaut. Zwischen den Screens kann man den Suchvorgang mit einem beliebigen Tastendruck aufhalten (mit einem weiteren fortsetzen) und mit Esc oder ⟨Ctrl⟩⟨C⟩ stoppen. Die Suche fortsetzen kann man mit ⟨Ctrl⟩⟨R⟩. Gelangt man an das Ende des Suchbereichs, so wird die Suchrichtung automatisch umgedreht.

Ganze Screens werden mit ⟨Alt⟩⟨C⟩ gelöscht. In Dateien fügt man mit ⟨Alt⟩⟨I⟩ einen Screen ein, der aktuelle Screen (mit allen weiteren) wird um eine Position nach hinten geschoben. Falls der letzte Screen nicht leer war, wird an die Datei ein weiterer Screen angehängt. Mit ⟨Alt⟩⟨D⟩ löscht man den aktuellen Screen und holt die weiteren eine Position nach vorn. Der letzte Screen wird dabei gelöscht, die Datei kann aber nicht verkürzt werden.

11. Spezialitäten

Die ID-Box, die wir am Anfang gesehen haben, kann man natürlich auch vom Editor aus aufrufen, und zwar mit ⟨Ctrl⟩⟨G⟩. Die ID erscheint bei einem editierten Screen rechts oben in der Ecke, aber nur, wenn man den Screen wechselt oder den Editor verläßt. Während des Editierens erscheint sie noch nicht.

Sinn der ID ist es, festzustellen, wer der Autor des Screens war und wann die letzte Änderung durchgeführt wurde. Will man vermeiden, seinen Stempel aufzusetzen (z. B. weil man nur Bagatelländerungen (sei es im Format) durchführen will), kann man entweder den ganzen ID-Text löschen oder die Box über "No ID" verlassen. Gefällt die Änderung der ID nicht, so verläßt man die Dialogbox über Cancel, es wird dann die alte ID beibehalten. Beim Start des Editors wird in diesem Fall übrigens der Vorschlag des Systems übernommen.

12. Dateien und Disketten

Da bigFORTH nicht nur direkt auf Disketten zugreifen kann, sondern ein DateiSystem hat, muß man auch die Datei wechseln können. Mit ⟨Alt⟩⟨U⟩ wählt man in der Fileselectorbox eine andere Datei aus. Die bisherige Position wird zur Marke. Screen 1 der neuen Datei wird aktueller Screen (falls sie überhaupt so groß ist, in ganz kleinen Dateien landet man im Screen 0).

Natürlich kann man auch Dateien erzeugen. Drückt man ⟨Alt⟩⟨M⟩, so erscheint eine Fileselectorbox, in der man den Namen eingibt. Läßt man den Dateisuffix weg, so wird .SCR angehängt. Die erzeugte Datei ist zwei Blöcke lang, um sie zu editieren, muß man sie mit ⟨Alt⟩⟨U⟩ auswählen.

Schließlich und endlich lassen sich mit ⟨Alt⟩⟨K⟩ Dateien löschen. Auch hier wird die zu löschende Datei mit der Fileselectorbox ausgewählt.

Über das Menü erreicht man auch eine Funktion zum Anlegen von Ordnern, aber dazu später.

13. Mehrere Fenster

Ein einziges Fenster, das mag ausreichen, aber der ST bietet schließlich eine grafische Benutzerumgebung, die mehrere Fenster möglich macht. So öffnet man in diesem Editor mit ⟨Alt⟩⟨O⟩ ein neues Fenster. Das entspricht einem Verdoppeln des gerade benutzten, man steht anfangs in derselben Datei an derselben Position, hat denselben Marker, kann sich aber davon lösen. Alles, außer dem Zeichen- und Zeilenstack und dem UNDO-Puffer ist unabhängig vom Fenster, man kann also (mit ⟨Alt⟩⟨U⟩) in eine andere Datei gehen und tun und lassen, was man will. Über die Closebox schließt man das Fenster wieder.

Um Dateien leichter mit Shadowscreens zu versehen, erzeugt man mit ⟨Alt⟩⟨S⟩ auch ein abhängiges Fenster, in dem der Shadowscreen zu sehen ist. Diese beiden Fenster verhalten sich wie siamesische Zwillinge, d. h. das andere Fenster zeigt immer den korrespondierenden Screen, gleich ob man Dateien wechselt oder sonstige Funktionen ausführt. Auch Marker werden in beiden Fenstern an der entsprechenden Stelle gesetzt. Ist das Shadowwindow schon geöffnet, holt man es mit ⟨Alt⟩⟨S⟩ nach oben und kann dort editieren, die Wirkung ist dann vergleichbar mit ⟨Ctrl⟩⟨W⟩.

Zur besseren Übersicht gibt es auf dem S/W-Monitor noch die Möglichkeit, den halbhohen Zeichensatz (⟨Alt⟩⟨8⟩) zu benutzen, man kann damit zwei Fenster auf einmal betrachten. Mit ⟨Alt⟩⟨L⟩ kommt man zurück zum großen Zeichensatz. Auf dem Farbmonitor ist das leider nicht möglich, da hier bereits der halbhohe Zeichensatz benutzt wird, um 25 Textzeilen darzustellen.

14. Verlassen des Editors

Die naheliegendste Möglichkeit: Sie schließen alle Fenster. Dann kommen Sie wieder in den Direktmodus zurück. Angenommen, Sie waren im Screen 0. Auf dem Bildschirm steht unter der Zeile, in der Sie den Editor aufgerufen haben: “Scr # 0 closed”. Wenn Sie den Editor mit V wieder aufrufen, landen Sie wieder genau im selben Screen, an derselben Position, es hat sich nichts geändert. Auch wurden keine Daten gesichert.

Sie hätten den Editor auch mit ⟨Ctrl⟩⟨U⟩ verlassen können (gleiche Wirkung). Ob Sie den Screen verändert haben oder nicht, steht nun in der Meldung zu lesen: “Scr # 0 updated” oder “Scr # 0 not updated”. Auch hier wird nichts auf Diskette zurückgesichert, sondern nur im Speicher gemerkt, ob etwas verändert wurde oder nicht. Der Unterschied zum einfachen Schließen ist, daß alle Fenster der Reihe nach geschlossen werden und sich die Meldung auf das oberste Fenster bezieht.

Normalerweise wird man den Editor mit ⟨Ctrl⟩⟨S⟩ verlassen, da hier die Veränderungen auf Diskette gespeichert werden (Meldung: “Scr # 0 saved”). Schlägt das Sichern fehl, weil z. B. die Diskette schreibgeschützt oder defekt ist, so bleibt man im Editor. Man kann ihn immer noch über ⟨Ctrl⟩⟨U⟩ verlassen.

Bricht man den Editor mit Esc ab (Meldung: “Scr # 0 canceled”), so wird der gerade editierte Screen aufgegeben (ähnlich wie bei UNDO). Da er aber unwiederbringlich verschwindet, wird man vorher mit einer Alertbox gewarnt.

Schließlich kann man auch mit ⟨Ctrl⟩⟨L⟩ aus dem Editor aussteigen, es wird dann gesichert und der Screen, der gerade editiert wurde, ab der Cursorposition geladen (Meldung: “Scr # 0 loaded”).

Beim Wiederaufruf des Editors mit V oder L wird das Fenster an derselben Stelle geöffnet, an der vorher das oberste lag.

Löschen Sie am Ende Ihrer Trainingssitzung die Blockpuffer und verhindern so, daß versucht wird, die veränderten Screens zu sichern:

EMPTY-BUFFERS⟨RET⟩

Der Editor ist immer nur ein zeitweiliges Werkzeug. Die Steuerung findet im Dialog mit FORTH statt, von hier aus rufen Sie ja auch den Editor auf. Der Dialog bleibt stets der Hintergrund des Geschehens, anstelle des einheitsgrauen (-grünen) Desktops, den man sonst in GEM-Programmen findet. Die verschiedenen Screens des Editors liegen als Blätter davor. Daher ist es auch sinnvoll, daß man nach dem Schließen des letzen "Blattes" (Fensters) wieder zurück zum FORTH-Dialog kommt.

15. Die Menüs

Natürlich lassen sich fast alle über Tastatur zugänglichen Funktionen auch per Menü erreichen. Da aber beim Editieren wohl doch die Tastatur die wichtigere Rolle spielen dürfte, steht dieser Teil im Handbuch am Schluß, er bietet zudem Gelegenheit zu einem systematischen Überblick.

Die Menüs sind vor allem für untrainierte Benutzer gedacht. Es gibt fast nichts, was man nicht schneller und bequemer über die Tastatur erreicht. Die Ausnahme bestätigt die Regel, so kann man per Maus und Fenster schneller zwischen Screens blättern (wenn die Wiederholfunktion benutzt wird) als mit ⟨Ctrl⟩⟨N⟩ und ⟨Ctrl⟩⟨B⟩, vor allem, wenn ein Shadowscreen gleichzeitig offen ist. Hier gibt nämlich die Window-Library dem reinen Blättern Vorrang und stellt erst nach dem Loslassen des Mausknopfes den gewünschten Endzustand her.

Und zweitens: Neue Ordner kann man auch nur von der Menüleiste aus erzeugen, denn diese Funktion braucht man so selten, daß ein Tastaturkürzel unnötig erschien. Außerdem läßt sich diese Funktion leicht mit MAKEDIR ⟨*Ordner*⟩ aus dem FORTH-Dialog erreichen.

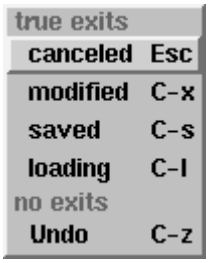
Dafür bietet das Menüsystem eine Hilfefunktion. Nach dem Aufruf eines Menüpunktes erscheint bei eingeschalteter Hilfe eine Alertbox, die erklärt, was man angewählt hat, und die auch noch einen Ausstieg ermöglicht ("NEIN" anklicken). Mit ⟨RET⟩ (oder "JA" anklicken) kann man die Aktion ausführen lassen. Die Hilfefunktion ist standardmäßig angeschaltet, nur wenn BFHELP.RSC nicht gefunden wird, ist sie ausgeschaltet.

Nun die Auflistung im einzelnen (Titel sind fett gedruckt und unterstrichen, nicht anwählbare Punkte hell; drei Punkte hinter dem Befehl zeigen an, daß weitere Werte in einer Dialogbox abgefragt werden):

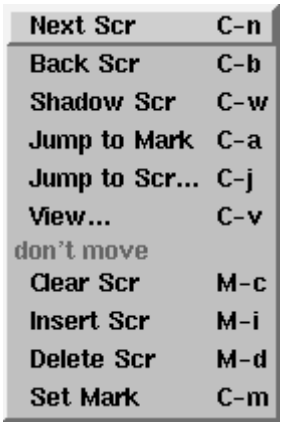
Dateinamen über Fileselektorbox anwählen:
Datei editieren (als aktuelle Datei wählen)
Datei erzeugen (Block 0 und Block 1)
Datei löschen

Ordner über Fileselectorbox erzeugen

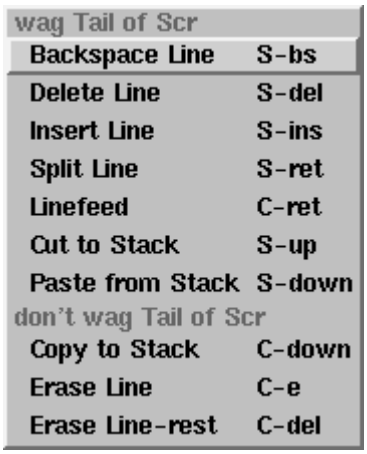
bigFORTH verlassen, Sources werden gesichert.



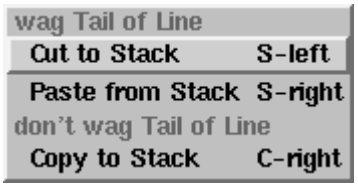
Editor wird verlassen, eine Meldung erscheint
Der aktuelle Screen wird nicht gespeichert
Alle Screens bleiben im Puffer stehen
Veränderte Screens werden gesichert
⟨Ctrl⟩⟨S⟩ und laden ab der Cursorposition
bleibt im Editor, betrifft aktuellen Screen:
Screen wird von Diskette neu geladen



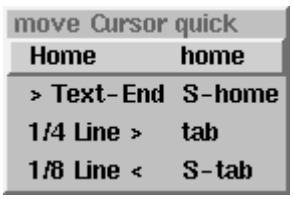
Zum nächsten Screen gehen
Zum vorherigen Screen gehen
Zum Shadowscreen springen
Zur Marke springen
Zum Screen n springen
Zum Screen springen, in dem ¡Wort¿ definiert ist
Hier bleibt der aktuelle Screen erhalten
Einen Screen einfügen, der Rest rückt weiter
Aktuellen Screen löschen
Aktuellen Screen löschen, der Rest rückt auf
Marke setzen



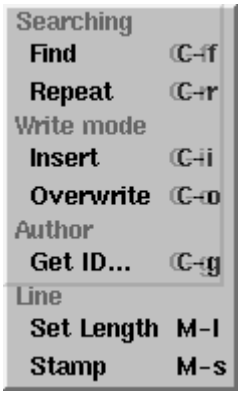
Letzte Zeile wird verschoben
Zeile über dem Cursor löschen, Rest rutscht hoch
Aktuelle Zeile löschen, Rest rutscht hoch
Eine Zeile einfügen
Zeile an der Cursorposition auftrennen
Rest der Zeile rückt eine Zeile tiefer
Zeile auf den Stack schieben, Rest rutscht hoch
Zeile vom Stack schieben, Rest rutscht runter
Letzte Zeile wird nicht verschoben
Zeile auf den Stack kopieren
Zeile löschen
Zeile ab Cursorposition löschen



Zeilenende wird verändert
Zeichen auf Stack schieben, Rest rutscht nach
Zeichen vom Stack schieben, Rest rutscht weiter
Zeilenende wird nicht verändert
Zeichen auf Stack kopieren



Die Cursortasten funktionieren auch...
Cursor nach links oben
Cursor ans Text-Ende setzen
Vorwärtstabulator alle viertel Zeile
Rückwärtstabulator alle achtel Zeile



Suche/Ersetze-Funktionen
Im Dialog suchen oder ersetzen lassen
Die letzte Suche wiederholen
Schreibmodus:
Einfügen
Überschreiben

ID eingeben oder verändern

Setze Zeilenlänge für den Streamfile-Editor
Setze einen Stamp im Streamfile-Editor

Ein neues Fenster wird geöffnet
Fenster "verdoppeln", neues ist unabhängig
Abhängiges Shadowfenster erzeugen

16. Fehlermeldungen des Editors

Wahrscheinlich haben Sie es schon bemerkt: Der Editor meldet sich, wenn er Ihren Wünschen nicht entsprechen kann. Ein Signal ertönt, und die Fehlermeldung erscheint in der rechten Hälfte der Infozeile des obersten Fensters. Falls sich Fehler wiederholen, ertönt die Glocke nur einmal, es wird aber immer die letzte Fehlermeldung ausgegeben.

Die Ausgabe erfolgt über die übliche Umleitung von ABORT" $\langle Meldung \rangle$", d. h. es können auch Fehlermeldungen aus dem System ausgegeben werden. Hier eine Auflistung aller Fehlermeldungen, die vom Editor selbst erzeugt werden:

"Border!": Sie sind mit dem Cursor oben oder unten am Screen angelangt, es geht hier nicht weiter.

"Out of range !": Vor dem ersten und hinter dem letzten Screen einer Datei ist kein Zugriff möglich.

"Not for direct access!": Einfügen eines Screens oder Löschen eines solchen (wobei die anderen vorgerückt werden) geht nicht im Direktzugriff auf eine Diskette.

"What?": Ein Steuerzeichen löst keine Funktion aus, der Editor weiß nicht, wie er darauf reagieren soll.

"Dann eben nicht !!": Eigentlich keine Fehlermeldung, es wird nur bestätigt, daß man einen Menüaufruf in der folgenden Hilfebox mit "NEIN" abgebrochen hat.

"marked !": Auch keine Fehlermeldung, es wird bestätigt, daß man den Screen markiert hat.

"Line buffer empty": Der Zeilenstack ist leer, es kann keine Zeile mehr herausgeschoben werden.

"Line buffer full": Es kann keine Zeile mehr aufgenommen werden. Der Fehler deutet eher auf einen zu geringen Platz zwischen PAD und den Stack an, als auf zu viele Zeilen im Puffer.

"You would lose a line": Der Screen ist bis unten voll. Sie können keine Zeile mehr nachschieben, die unterste würde aus dem Screen "herausfallen".

“Char buffer empty”: Zeichenstack leer.

“Char buffer full”: Der Zeichenstack ist voll. Er hat eine begrenzte Kapazität von 128 Zeichen (2 Zeilen).

“You would lose a char”: Sie können kein Zeichen mehr nachschieben, da die Zeile bis zum Rand voll ist. Automatischen Umbruch gibt es nicht, da er das Format des Screens nachhaltig stören würde.

“not found”: Die Suche nach ⟨Ctrl⟩⟨F⟩ oder ⟨Ctrl⟩⟨R⟩ ist erfolglos verlaufen oder wurde mit ⟨Esc⟩ bzw. ⟨Ctrl⟩⟨C⟩ abgebrochen.

“not enough room”: Das zu ersetzende Word kann an dieser Stelle nicht ausgetauscht werden, da das andere zu lang ist und in dieser Zeile nicht genug Platz dafür ist.

“use find first”: Der Suche-Puffer ist leer, ⟨Ctrl⟩⟨R⟩ kann nicht arbeiten. Rufen Sie die Findbox mit ⟨Ctrl⟩⟨F⟩ auf.

17. Externe Editoren

Trotz all dieser Features wird es sicher immer noch Leute geben, die lieber einen Editor ihrer Wahl benutzen wollen. Das Hauptproblem dabei ist das Format. Doch auch dieses Problem kann man umgehen. In STREAM.SCR auf der blauen Diskette ist eine Utility, die normale Text-Dateien (“Streamfiles”) einladen kann. Zuerst laden Sie also diese Datei mit dem Befehl

INCLUDE␣STREAM.SCR⟨RET⟩

Dann lassen sich Dateien eines Ascii-Editors mit #INCLUDE ⟨*Datei*⟩ laden oder mit #LIST ⟨*Datei*⟩ auflisten. Der Editor muß folgendes Format erzeugen:

- Text mit Ascii-Zeichensatz codiert.
- Zeilenende mit CR und LF ($0D und $0A) markiert, steht nur eines von beiden, so wird das auch akzeptiert.
- Maximale Zeilenlänge: 255 Zeichen.
- Steuerzeichen (Ascii-Code<$20) werden in Leerzeichen ($20) gewandelt.

Dieses Format erzeugen alle bekannten Programmeditoren, es dürfte also keinerlei Schwierigkeiten geben.

Bedenken Sie aber, daß der Editor nicht direkt in das FORTH-System eingebunden ist und damit ein (zeitraubender) Wechsel zwischen FORTH und Editor kaum zu vermeiden ist. Zwei Möglichkeiten, diesen Wechsel zu umgehen oder erleichtern, werden hier vorgestellt, sie sind aber beide (vor allem die letzere) nicht ohne weiteres für Anfänger verwirklichbar.

Erstens: Sie starten bigFORTH als Accessory (s. u.). Von einem GEM-Editor aus können Sie dann das bigFORTH-Fenster öffnen und profitieren von dem eingeschränktem Multitasking unter GEM. Sie brauchen nur die Fenster zu wechseln und sind einmal im Editor und gleich darauf im FORTH. Der Nachteil: Sie können das FORTH nur eingeschränkt nutzen, denn Sie müssen sich an die Spielregeln für Accessories halten. Und Sie müssen selbst darauf achten, daß die Dateien des Editors auch vom bigFORTH geladen werden können, sie müssen also vorher gesichert werden. Schließlich gibt es (zumindest bei

den bekannten Editoren) keine Möglichkeit der Kommunikation zwischen bigFORTH und Editor.

Da bleibt noch die Alternative: Sie rufen den Editor von bigFORTH auf, aber das ist noch komplizierter. Zuerst müssen Sie entscheiden, wieviel Platz der Editor braucht. bigFORTH läßt normalerweise nach dem Start gerade 64 KBytes übrig, die es zum Teil selbst wieder mit den Resourcedateien belegt. 256 KBytes müßten es schon sein, damit man einigermaßen vernünftig arbeiten kann.

Laden Sie dazu RELOCATE.SCR (mit INCLUDE, versteht sich) und geben

$40000␣RESERVE␣BIGFORTH.PRG⟨RET⟩

ein. Sie haben nun nach einem erneuten Systemstart $40000 Bytes=256 KBytes frei, von denen noch etwas für die Resourcedateien verloren geht. Diese veränderte Systemdatei müssen Sie dann starten, denn nur hier kann der Editor nachgeladen werden.

Aufgerufen wird der Editor dann mit RUN“ [⟨*Datei*⟩]” ⟨*Editor*⟩.PRG. Dieses Wort benutzt den GEMDOS-Befehl pexec und ist nicht nur in STREAM.SCR, sondern auch in DOS.SCR definiert.

18. bigFORTH als Accessory

Ja, auch das geht, bigFORTH in eines jener “kleinen” nützlichen Utilities verwandeln, die einen Eintrag in das Desk-Menü schreiben und beim Aufruf eine Dialogbox oder ein Fenster öffnen und dort nützliche Funktionen zur Verfügung stellen. Um bigFORTH so aufzurufen, muß man es lediglich in BIGFORTH.ACC umbenannt in das Wurzelverzeichnis der Bootdiskette bzw. der Bootpartition der Festplatte kopieren (die Resourcedateien müssen auch da sein) und den Rechner neu starten.

Es erscheint ein Eintrag im Desk-Menü (“ bigFORTH”), mit dem wie gewohnt das Accessory aufgerufen wird. Sie haben sämtliche Funktionen von bigFORTH, einschließlich Editor zur Verfügung. Dieser muß leider ohne die Menüleiste auskommen, da Accessories keine solche anmelden dürfen. Aber da man praktisch alle Befehle des Editors auch von der Tastatur aus erreicht, bedeutet dies keinen wesentlichen Nachteil.

Und wozu das Ganze? bigFORTH ist schließlich kein kleiner Brocken, es belegt einen großen Teil des Speichers, mehr als 256 KBytes. Allerdings kann man damit die fehlende Möglichkeit eines ständig vorhandenen, leistungsfähigen (weil programmierbaren) Kommandointerpreters ersetzen. Auf jeden Fall kommt ein bißchen Multifinder-Feeling auf und tröstet uns darüber hinweg, daß das TOS durchaus noch seinen Meister finden kann, was die Bedienerfreundlichkeit angeht.

19. Die Notbremse

Auch Spitzenprogrammierer übersehen manchmal, daß sie z. B. eine Endlosschleife geschrieben haben. Das Programm hängt dann, der Rechner reagiert auf Tastendrücke gerade noch mit dem obligatorischen Signalton, sonst passiert nichts. Hier hilft (außer dem Griff zu Sicherung, Netzstecker, Ein/Ausschalter) nur der Resetknopf. Danach wird zwar neu gebootet, das Programm geht vielleicht verloren, aber der Rechner läuft wenigstens wieder.

Lassen Sie sich nicht abhalten: Bei bigFORTH passiert etwas anderes. Nach dem Reset erscheint der alte Bildschirm wieder, es ertönt die Glocke, “You hit reset!” steht in einer neuen Zeile, eine Sekunde später ertönt noch einmal die Glocke, “ ok” wird ausgegeben und man befindet sich wieder im FORTH-Dialog und kann unverzüglich weiterarbeiten - sofern

man wirklich eine Endlosschleife abgebrochen hat, die nur FORTH-Befehle aufgerufen hat und der Prozessor während des Resets im FORTH-System beschäftigt war.

Anderenfalls gibt es unterschiedliche Fehlermöglichkeiten: Aus Teilen des BIOS, XBIOS und GEMDOS kann problemlos ausgestiegen werden (Bildschirmausgaben, einfache Abfragen, die nur Systemvariablen auslesen), während Diskettenoperationen kann es aber Schwierigkeiten geben. Line-A ist völlig reentrant (wiedereintrittsfähig), auch GEM-VDI kann wieder aufgerufen werden. Nur GEM-AES ist nicht reentrant: Beim nächsten AES-Aufruf gäbe es mit an Sicherheit grenzender Wahrscheinlichkeit einen Absturz.

Und dann? Es gibt eine Möglichkeit, einen echten Reset durchzuführen. Während der einen Sekunde bis zum zweiten "Bing" hat sich bigFORTH aus dem Resetvektor ausgehängt. Drückt man also gleich wieder auf Reset, so springt das TOS die ursprüngliche Resetroutine (bzw. gar keine) an und es wird neu gebootet.

Manchmal ist auch nur ein Teil des FORTH-Systems zerstört, und es würde ausreichen, COLD aufzurufen, um weiterzuarbeiten, aber es geht nicht. Drückt man dann gleich nach dem Reset auf ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩, so führt bigFORTH nach dem zweiten Glockenzeichen COLD aus, die neu definierten Wörter werden vergessen, die User Area wird neu geschrieben und die Stacks werden geleert.

Wird beim Reset ein anderer Task des FORTH-Systems abgebrochen, so ist er nachher angehalten, was ja bei einer Endlosschleife auch sinnvoll erscheint.

20. Exception-Trapping

Kommt es zu einer sogenannten Prozessor-Exception, wirft das TOS, seinen anarchistischen Trieben folgend, Bomben. Da dies sehr oft vorkommt, vor allem bei der Programmentwicklung und gerade, wenn man eine systemnahe Programmiersprache wie FORTH benutzt, wäre diese wenig aussagekräftige Fehlermeldung nicht gerade das Richtige. Zudem wird dabei das gerade laufende Programm beendet.

Unter bigFORTH wird stattdessen ein kompletter Post-Mortem-Dump aufgelistet. Folgende Fehler können auftreten:

Bus Error: Es wird auf einen Bereich zugegriffen, der nicht belegt ist oder auf den im Moment kein Zugriff möglich ist. Der Busfehler muß von einem Periferiebaustein (der MMU) ausgelöst werden.

Address Error: Wörter und Langwörter (16 und 32 Bit) können beim 68000er nur an geraden Adressen gelesen werden, es ist also versucht worden, auf eine ungerade Adresse zuzugreifen.

Illegal Instruction: Dieses Wort kann vom Prozessor nicht als Befehl interpretiert werden.

Sämtliche Parameter, die der Prozessor bei der Ausnahmebehandlung auf den Stack legt, werden ausgewertet und angezeigt. Bei Bus und Address Error wird in der ersten Zeile angegeben, ob der Fehler beim Lese- oder Schreibzugriff auftrat ("Read"/"Write"), ob während der Ausführung einer Instruktion, die weitergeführt werden kann oder nicht ("(No_)Instruction") (für den 68000er belanglos), ob der Prozessor im Supervisor- oder im User-Mode war und ob er beim Lesen von Daten ("Data") oder Programm geschah.

In der zweiten Zeile steht dann noch die Adresse und der Befehl, der die Probleme bereitet.

Darunter findet man dann auch bei Illegal Instruction folgende Daten:

Statusregister (SR) und Programmzähler (PC), die 8 Datenregister und die 8 Adreßregister, den Inhalt des Returnstacks und des Parameterstacks.

Schließlich wird mit ABORT“ noch die Art des Fehlers ausgegeben, es erscheint also die übliche Alertbox, nur mit “Address Error !”, “Bus Error !” oder “Illegal Instruction !” als Meldung. Diese Angaben dürften genügen, um diese Fehler leichter aufzufinden.

Vom Kernel wird nur die Art des Fehlers selbst gemeldet, die erweiterte Ausgabe wird mit EXCEPT.SCR dazugeladen und ist erst in BIGFORTH.PRG vorhanden.

Probleme kann es hier genauso wie beim Reset dann geben, wenn die Exception nicht im FORTH selbst, sondern in anderen Teilen des Betriebssystems auftrat. Meistens ist dann der Speicherinhalt schon so weit zerstört, daß nur ein wirklicher Neustart hilft.

21. Der externe Relocater

TOS verlangt, daß Programme relokatibel sind, also an einer beliebigen Adresse lauffähig. bigFORTH stellt zwei Möglichkeiten zur Verfügung, zum einen der interne Relocater (siehe Kapitel 4.24), zum anderen einen externen. Diese doppelte Lösung hat ihren Grund darin, daß die eine unpraktisch und die andere inkompatibel und fehlerträchtig ist. Wenden wir uns hier der unpraktischen zu.

Es gibt eine sichere Methode, Adressen auf die Spur zu kommen: Man compiliert das System zweimal an unterschiedlichen Adressen. Diese beiden Systeme vergleicht man und kann so problemlos Adressen, Daten und Befehle unterscheiden. Die so gewonnenen Informationen kann man entweder im TOS-Format oder im Format des internen Relocaters abspeichern — das TOS-Format hat den Vorteil, daß es nur auf dem Massenspeicher Platz verbraucht und dort zudem meist noch weniger als das bigFORTH-Format.

Das Tool, das diese Aufgabe löst, heißt RELOCATE.PRG und befindet sich auf der blauen Diskette. Es soll aber erst angewendet werden, wenn das Programm bereits fehlerfrei compilierbar ist. Zum Austesten muß es ja noch nicht relokatibel sein. Nach dem Start erscheint folgende Dialogbox:

In der Zeile nach “Loadfile:” gibt man ein, wie das Programm heißen soll, das man laden will (z. B. FORTHKER.PRG), in der Zeile “Save as :”, unter welchem Namen man das fertige System sichert (z. B. als MYFORTH.PRG). Klickt man diese Felder an, so erscheint eine Fileselectorbox, mit der man die Datei und den Pfad auswählen kann.

Im Editierfeld darunter steht die Kommandozeile. Es ist deshalb auch mit “Command line:” überschrieben. Hinter diese Kommandozeile wird auf alle Fälle der Befehl GOODBYE gesetzt, der das System so verläßt, daß es auch wieder gestartet werden kann.

In der letzten Zeile hat man noch die Möglichkeit, die Relocaterinfo im TOS- oder bigFORTH-Format abzuspeichern. Der Relocater darf auch seinen Kommentar abgeben, wenn der Knopf “RELOCATE.INF” angewählt ist. Hier wird ein Bitstring abgespeichert, in dem jedes Bit für zwei Bytes des Systems steht. Jede Adresse, die fälschlicherweise nicht im der internen Relocaterinfo markiert ist und jede Zahl (außer 0, die für die Adresse nil steht), die als Adresse markiert ist, wird mit einem gesetzten Bit angezeigt.

Über “Cancel” kann man die Dialogbox und das Programm RELOCATE.PRG ohne weitere Auswirkungen verlassen.

Nach einem Klick auf “OK” wird das Programm nach “Loadfile:” zweimal gestartet, es wird ihm dabei jedesmal die Kommandozeile übergeben. Das Programm wird dabei mit dem GEMDOS-Befehl Pexec #3 (“nur laden”) geladen, damit bleibt der eigentliche Programmspeicher “Eigentum” des aufrufenden Prozesses (also von RELOCATE.PRG).

Gestartet wird dann mit Pexec #4 (“nur starten”). Atari selbst warnt vor der Verwendung dieser Modi, obwohl sie ausgezeichnet funktionieren (und für diese Anwendung den einzigen Weg darstellen).

Nach der Rückkehr zu RELOCATE.PRG wird der Programmspeicherbereich auf das Minimum geschrumpft. Damit liegen die beiden Systeme um eine Distanz auseinander, die ihrer eigenen Länge (+ Basepage) entspricht. Wie man leicht erraten kann, benötigt RELOCATE.PRG eine Menge freien Speicherplatz. Auf einem ST mit 512 KBytes dürfte es da schon problematisch werden. Hier hilft nur eine Speichererweiterung.

Danach sucht RELOCATE.PRG erstmal, wo Adressen stehen könnten. Der Bitstring des Systems (falls vorhanden) wird dann auch noch zu Rate gezogen, allerdings dürfen nur 0-Adressen (Zeiger auf nil) zusätzlich markiert sein. Schließlich wird das zuerst geladene System auf 0 reloziert und gesichert.

Der Relocater-Bitstring wird je nach Einstellung so gesichert, wie er ist oder in die TOS-Relocater-Info umgewandelt. Bei der zweiten Einstellung werden nur die tatsächlich als relokatible Adressen gefundene Stellen reloziert und gesichert, da das TOS keine Ausnahmebehandlung für 0-Adressen besitzt. Außerdem kann das System nach dem Laden nicht mehr mit SAVESYSTEM gesichert werden, da der dazu notwendige Bitstring fehlt.

Fehler bei der Ausführung werden als TOS-Fehlernummer an den aufrufenden Prozeß (also meistens wohl der Desktop) weitergeleitet. Der Desktop reagiert auf einige Nummer mit einer Alertbox, in der die bekannten Atari-amtsdeutschen Meldungen wie “Die Anwendung kann das angesprochene Objekt nicht finden” oder ähnliche hilfreiche Nachrichten stehen. Da normalerweise nichts schiefgehen sollte, sind diese Meldungen ausreichend.

Tritt beim Laden in bigFORTH ein Fehler auf, so erscheint eine normale Fehlermeldung von bigFORTH. Es muß dann mit BADBYE verlassen werden! Nur dann kann RELOCATE.PRG erkennen, daß etwas schiefgegangen ist.

RELOCATE.PRG kann durch einfaches Umbenennen in RELOCATE.ACC auch als Accessory gestartet werden. Dazu muß es sich natürlich im Root-Directory des Boot-Devices (Diskettenlaufwerk A: bzw. Festplatte) befinden und gebootet werden. Im Desk-Menü steht dann ein Punkt “ bigFORTH→rel”. Klickt man ihn an, so erscheint die Dialogbox und alles funktioniert wie oben beschrieben.

Da natürlich auch hier eine Menge Platz vorhanden sein muß, lohnt sich der Einsatz (außer auf einem Mega ST 4) nur vom Desktop aus - sinnvoll ist das höchstens dann, wenn man nicht nach der Diskette suchen will, auf der RELOCATE.PRG abgespeichert ist.

3 FORTH-Kurs

1. Ziel des Kurses

Dieser Kurs soll kein detailliertes FORTH-Buch ersetzen, sondern einen kurzen Einstieg ermöglichen. Deshalb wurde auf viele Beispiele und weitergehende Erklärungen verzichtet. Alle Konzepte von FORTH, die von anderen Programmiersprachen abweichen, werden erklärt. Dem Umsteiger mag dieser Kurs ausreichen, sich in FORTH einzuarbeiten; falls nach der Lektüre noch Fragen offen bleiben, seien Sie auf ein Lehrbuch in FORTH verwiesen.

Die Beispiele in diesem Kurs sollen gleich eingegeben werden. Spielen Sie ruhig ein bißchen mit ihnen herum, das vermittelt Ihnen ein Gefühl für die Eigenheiten von FORTH.

2. Was ist FORTH?

FORTH wurde Ende der 60er Jahre von Charles Moore entwickelt. Er verwendete es zur Steuerung eines Observatoriums. Ursprünglich hatte er seinem Produkt den Namen "Fourth Generation Language" („Sprache der vierten Generation“) gegeben, da sein System aber nur Namen mit bis zu fünf Zeichen zuließ, wurde daraus eben FORTH. Dieses Wort heißt eigentlich „nach vorne“. Von der Aussprache her ist es gleich wie "fourth" („vierte“) und sehr ähnlich wie "force" („Kraft“). Daher sei Charles Moore gedankt, daß er der Sprache einen recht vieldeutigen Namen gegeben hat, die ihren Charakter viel besser widerspiegelt als z. B. "FGL".

Einige Zeit lang blieb FORTH ein Geheimtip. Mit FORTH konnte man einen Rechner von der Leistungsfähigkeit des weitverbreiteten C64, allerdings mit Festplatte zur Steuerung eines kompletten Observatoriums einschließlich Meßwertauswertung verwenden — in einer Zeit, in der es solche Rechner noch nicht im Supermarkt gab, war das ein nicht zu unterschätzender Vorteil. Für diesen Zweck muß das Betriebssystem realtimefähig („echtzeitfähig“) sein. Solche Systeme waren damals ohnehin noch dünn gesät.

So verbreitete sich FORTH vorerst kaum, anscheinend hatte das DoD (Department of Defence, das amerikanische Verteidigungsministerium) bei den Konferenzen, die zu Ada geführt haben, überhaupt keinen Wind davon bekommen.

Ende der 70er Jahre wurde es dann Public Domain, die Forth Interest Group (fig) wurde gegründet. '78 erschien mit figFORTH ihr erstes System. '79 wurde der erste Standard definiert, '83 folgte der zweite, heute noch gültige. Eine ANSI–Norm wird zur Zeit diskutiert, es sollen dabei auch 32–Bit–Systeme in die Norm einbezogen werden. Die Entwicklung ist dem F83–Standard davongelaufen, bigFORTH ist (leider) ein Beispiel, da es an einigen Stellen ganz deutlich vom Standard abweicht. Trotzdem baut es auf dem Sourcecode der fig auf, die von Perry und Laxen zum F83–Standard geschrieben wurde, es sind also alle F83–Wörter vorhanden.

Was bedeutet „Sprache der vierten Generation“? Die Bezeichnung steht für die geschichtliche Entwicklung der Computerprogrammierung. Zuerst wurden Computer direkt mit gestanzten Lochstreifen programmiert, deren Lochung der Prozessor als Befehle interpretierte. Gestanzt wurde binär oder mit einfachen Stanzmaschinen hexadezimal. Das waren die Sprachen der ersten Generation.

Da dieses System für den Menschen nur schwer zu merken und anzuwenden war, erfand man nach kurzer Zeit “Mnemonics”, meist dreibuchstabige Kürzel für die Befehle, deren Bedeutung man leicht erraten und sich noch leichter merken konnte. Diese Kürzel wurden von einem „Assembler“ direkt in die binären Prozessorbefehle übersetzt, am Wesen der Programmierung änderte sich nichts, nur an der Form der Notation. Trotzdem bezeichnet man Assembler schon als Sprache der zweiten Generation.

Wieder ein paar Jahre später waren die Computer viel größer (leistungsmäßig, vom Platzbedarf her kleiner), es gab die ersten Plattenlaufwerke, und mehr und mehr relativ computerunerfahrene Wissenschaftler benutzten Rechner. Sie wollten ihre Formeln in gewohnter Weise eingeben und sich nicht mit der Assemblerprogrammierung herumschlagen. So entstand FORTRAN (und wenig später COBOL für kaufmännische Anwendungen), aus dem ein Compiler ein Assemblerprogramm erzeugte, das schließlich in Maschinensprache übersetzt wurde und erst dann lief. Diese Sprachen nennt man Sprachen der dritten Generation.

Die Sprachen, die später entwickelt wurden, kann man nicht mehr so leicht einordnen. Die meisten Compilersprachen stammen mehr oder weniger direkt von Algol–68 ab. Algol–68 wurde von einer Informatikerkonferenz entwickelt und war der erste Versuch, aus der damals schon drohenden „Softwarekatastrophe“ zu entrinnen. Man nutzte konsequent das Prinzip der „strukturierten Programmierung“. Die meisten später entstandenen Sprachen sind Algol–68 so ähnlich, daß sie sich mit einem modifizierten Algol–68–Compiler verarbeiten lassen (dies vor allem deshalb, weil Algol–68 eine Monstersprache geworden ist — vergleichbar etwa mit PL/1 und ADA).

Gerade um die Zeit ’rum (also ’68) wurde das Terminal erfunden und es gab erste “time–sharing”–Systeme. Man konnte erstmals direkt am Computer arbeiten, ohne daß ein Operator zwischengeschaltet war. Diese Technik schrie nach neuen Arbeitsmethoden — mit klassischen Programmiersprachen blieb der Arbeitsablauf ja derselbe, außer, daß der Programmierer auch noch Operator spielen mußte. Also erfand man die Interaktivität. Basic war dann eine der ersten interaktiven Programmiersprachen überhaupt, aber gerade deshalb rechnet man Basic üblicherweise zu den Sprachen der 3. Generation.

Der Computer wurde damals Werkzeug von Nicht–Programmierern, die eigentlich nur auf Datenbanken zugreifen wollten, oder Texte mit einem Editor schreiben. Auch sie entdeckten die Mächtigkeit von Programmen — allerdings sauber in ihre gewohnte Umgebung eingebunden und leichter zu durchschauen, weil die Programmiersprachen von Editor und Datenbank interaktiv arbeiten und speziell auf ihren Einsatzzweck zugeschnitten sind — im Gegensatz zu den „eierlegenden Wollmilchsäuen“, den Sprachen der 3. Generation.

FORTH als Sprache ließe sich nahtlos in die 3. Generation einfügen — FORTH kann von Haus aus kaum mehr, als ein Programm erzeugen; bringt man aber genügend Geduld auf, so kann man FORTH alles beibringen — es ist also eine „eierlegene Wollmilchsau“. Andererseits ist FORTH interaktiv, und eine in FORTH definierte applikationsspezifische Sprache ist absolut keine große Sache. Es ist dies sogar der übliche Weg, FORTH zu programmieren.

Da die Interaktivität auch vor dem Compiler nicht haltmacht, läßt sich FORTH formen, wie keine andere Sprache (und bleibt doch FORTH, denn alle diese Erweiterungen werden direkt in die FORTH–Ebene eingefügt). KI–Elemente wie Listen oder Backtracking, Objektorientierung oder Fuzzy Logic sind in FORTH leicht zu implementieren (von Neuronalen Netzen will ich gar nicht reden, die gehen auch mit Pascal).

Zu allem Überfluß bringt FORTH auch ein Betriebssystem mit. Terminal und Massenspeicher werden von FORTH so verwaltet, daß man es mit geringem Aufwand auch auf einem „nackten“ Rechner implementieren kann. 8 KByte FORTH–Code reichen im

allgemeinen aus, um einen kleinen Rechner wachzuküssen — einschließlich Massenspeicherverwaltung, Terminal, Compiler und Interpreter.

Schließlich ist FORTH noch eine Philosophie. Jede Sprache (ob eine natürliche oder eine Computersprache) beeinflußt das Denken des Sprechers, denn in jeder Sprache sind Denkmuster eingeprägt, die dann in das Gehirn des Benutzers der Sprache überwandern. Die Philosophie von FORTH ist die des Wortes. Jedes Wort in FORTH hat seine eigene Kraft, seine eigene Bedeutung — es ist von allen anderen Wörtern nicht abhängig. FORTH lehrt, wie man ein Problem in kleine, unabhängige Komponenten zerlegt. Oft kommt einen erst dann die Erkenntnis, daß das Problem gar nicht so komplex und verzahnt war, wie ursprünglich angenommen.

Von einem Programmierer, der mit Basic, Pascal, Modula 2 oder C groß geworden ist, fordert FORTH ein drastisches Umdenken. Der Mächtigkeit der Sprache auf der einen Seite steht eine meist spartanische Ausstattung auf der anderen gegenüber — da man an dieser Ausstattung erst dann etwas ändern kann, wenn man das Konzept verstanden hat, und man sich durch die völlig ungewohnte Notation durchgekämpft hat, ist die Hemmschwelle von FORTH doch leider recht hoch.

3. Stapel, Zahlen, oder: wie man mit FORTH rechnet

Das bekannteste ungewöhnliche Konzept von FORTH ist der Stack. Er ist das zentrale Medium in FORTH, auf dem Daten bearbeitet werden. „Stack“ bedeutet „Stapel“. Stellen Sie sich einen Stapel Papier vor, auf den Sie Blätter legen können, wieder herunternehmen, aus dem Stapel herausziehen, nach oben legen und wieder in den Stapel hineinschieben. Genau so funktioniert das auch mit dem Stack in FORTH. Geben Sie einmal ein paar Zahlen ein und drücken Sie RET (Ihre Eingaben werden unterstrichen dargestellt, die Antworten des Computers nicht):

```
2␣5⟨RET⟩  ok
```

FORTH hat Ihre Eingabe akzeptiert („ok“). Anscheinend hat die Eingabe keine sichtbare Wirkung. Tatsächlich aber sind die Zahlen der Reihe nach auf dem Stack gelandet. Geben Sie .S ein, dieses Wort gibt den Stack aus, ohne dessen Inhalt zu löschen.

```
.S⟨RET⟩ 5 2  ok
```

.S gibt das oberste Element zuerst aus, deshalb zeigt es die beiden Zahlen in vertauschter Reihenfolge an. Die FORTH-Befehle erwarten ihre Argumente auf dem Stack. Geben Sie + ein:

```
+␣.⟨RET⟩ 7  ok
```

5+2 ergibt 7. Der Punkt gibt das oberste Stackelement aus, es wird dabei vom Stack genommen. .S zeigt, daß der Stack wieder leer ist, denn auch + hat seine Argumente verbraucht:

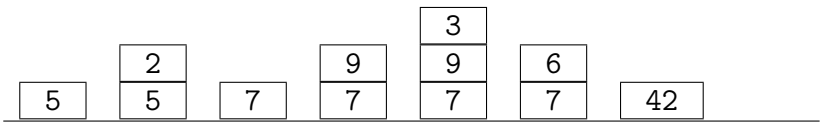
```
.S⟨RET⟩  ok
```

Sie hätten die Zeile auch auf einmal eingeben und andere Grundrechenarten (−, ∗ oder /) verwenden können:

```
5␣2␣+␣9␣3␣-␣*␣.⟨RET⟩ 42  ok
```

Was passiert dabei auf dem Stack? Wir können uns das ja mal ansehen:

```
5  2  +  9  3  -  *  .  42  ok
```



Die 7, das Ergebnis der ersten Berechnung, liegt auf dem Stack solange unter den neuen Werten, bis sie gebraucht wird.

Diese Formel–Notation nennt man Postfixnotation oder umgekehrt polnische Notation (UPN) nach dem polnischen Mathematiker Jan Lukasievicz. Sie kommt ohne Klammern aus und heißt auch Zeitfolgennotation, weil die Operationen in der Reihenfolge ihres Auftretens ausgeführt werden.

Um einen solchen Ausdruck in die gewohnte Schreibweise (Infixnotation) umzuformen, gehen Sie von hinten vor:

1. `5 2 + 9 3 - *`
2. `5 2 +` $(9-3)$ `*`
3. $(5+2)$ $(9-3)$ `*`
4. $(5+2)*(9-3)$ $(=42)$

Fertig!

Hier könnte man noch unnötige Klammern weglassen, die durch Vorrangregeln oder Ausführungsreihenfolge (von links nach rechts, was übrigens keine notwendige Eigenschaft dieser Darstellungsform ist) unnötig sind.

Umgekehrt verfahren Sie bei der (wohl wichtigeren) Konvertierung von Infix- in Postfixnotation. Klammern Sie dabei jede Operation so, daß Sie die zugehörigen Paare genau erkennen können, also bis der ganze Term von einer Klammer umgeben ist. Lösen Sie anschließend diese Klammer auf, indem Sie den zugehörigen Operator hinter die Klammer schreiben. Beispiel:

1. $6+5*(2+3)-7$ $(=24)$
2. $6+(5*(2+3))-7$
3. $(6+(5*(2+3)))-7$
4. $((6+(5*(2+3)))-7)$
5. $(6+(5*(2+3)))$ `7 -`
6. `6` $(5*(2+3))$ `+ 7 -`
7. `6 5` $(2+3)$ `* + 7 -`
8. `6 5 2 3 + * + 7 -`

6␣5␣2␣3␣+␣␣+␣7␣-␣.* ⟨RET⟩ `24 ok`

Die Vorteile der UPN sind vielleicht nicht sofort ersichtlich. Sicher, der Compiler hat es leichter. Aber das war vielleicht bei kleinen Systemen mit ein paar KByte Hautspeicher von Belang, bei großen Systemen wie bigFORTH spielt es kaum eine Rolle. Aber das Programm wird in derselben Reihenfolge ausgeführt, wie man es liest, nämlich von links nach rechts. Das ist sehr wohl von Bedeutung; in klassischen Sprachen gibt es hierzu keine definitiven Regelungen.

Zudem umgeht man mit dem Stack das lästige Problem, wie man mehrere Werte zurückgeben soll. In anderen Sprachen wird das mit referenzierten Variablen oder Arrays gemacht, mit beiden Rückgabearten kann man aber nicht direkt weiterrechnen. In FORTH kommen die Werte einfach auf den Stack, wo sie für den weiteren Gebrauch gleich richtig liegen.

4. Stackbefehle

Stellen Sie sich vor, Ihr Programm soll folgende Funktion berechnen: $x * (x + 4)$. Als Übergabeparameter bekommt es den Wert von x. Zurück gibt es den Funktionswert. Wandeln wir die Formel einmal in UPN:

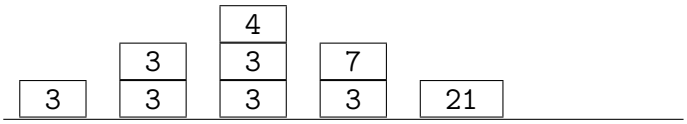
1. $x * (x + 4)$
2. `x` $(x + 4)$ `*`
3. `x x 4 + *`

Da das x zuerst auf dem Stack liegen soll, haben wir noch das Problem, wie wir es an die zweite Stelle bekommen. Das x muß also zuerst verdoppelt werden, dann hat man es in der gewünschten Form. Hierzu dient das Wort DUP (n −− n n).

Es verdoppelt das oberste Stackelement (TOS, Top of Stack). Probieren Sie es einfach mal aus:

```
3  DUP  4  +  *  . ⟨RET⟩  21  ok
```

Stackdiagramm:



Wir wollten ein Programm schreiben, das die Funktion ausrechnet. Dazu müssen Sie sich einen Namen überlegen, etwa FUNKTION1. Geben Sie ein:

```
: FUNKTION1 ( n -- f1 ) ⟨RET⟩  compiled
  DUP 4 + * ; ⟨RET⟩  ok
3 FUNKTION1 . ⟨RET⟩ 21  ok
5 FUNKTION1 . ⟨RET⟩ 45  ok
```

Sie sehen, der Aufruf von FUNKTION1 bewirkt dasselbe wie die vorherige Eingabe von `DUP 4 + *`. Während der Compilation gibt bigFORTH statt „ok“ „compiled“ aus.

Der Doppelpunkt leitet die Definition ein, der Strichpunkt beendet sie. In der Klammer steht ein Kommenter, der den Stackeffekt beschreibt. Diese Klammer wird vom Compiler ignoriert. Man sollte alle Wörter mit dem Stackeffekt kommentieren. Links vom Doppelstrich („−−“) stehen die Aufrufparameter, rechts davon die Rückgabewerte. Bei mehreren Werten steht der Top of Stack rechts, die darunterliegenden Werte gehen nach links. Man muß die Werte im aufrufenden Programm in der Reihenfolge auf den Stack legen, in der sie im Kommentar stehen.

Sie haben mit der Definition von FUNKTION1 den Wortschatz von bigFORTH erweitert. FUNKTION1 ist jetzt ein vollwertiger Befehl, die meisten Befehle in bigFORTH sind so definiert, der Anteil der Assemblerbefehle ist relativ gering. Die einzelnen Befehle heißen in FORTH Wörter (engl. words). Oft steht dieser Begriff auch für die Speichereinheit, auf die der Prozessor zugreifen kann. Beim 16/32-Bit-Prozessor 68000er ist ein Wort 16 Bit lang, ein Langwort 32 Bit. In FORTH heißt die Speichereinheit Zelle (cell).

FORTH baut auf einer Vielzahl dieser meist kurzer Wörter auf, es gibt kein großes und langes Hauptprogramm. Ein Wort muß normalerweise in einem Screen Platz haben, darf also nicht länger als 15 Zeilen sein. Man kann Wörter zwar auch über mehrere Screens definieren, aber dann geht die Übersicht verloren. Guter FORTH–Stil ist es erst, wenn die meisten Wörter höchstens zwei Zeilen lang sind. Solche stark gegliederten Programme erhöhen nicht nur die Übersichtlichkeit, sie vereinfachen auch die Programmentwicklung,

da man die Wörter oft so allgemein formulieren kann, daß eine Wiederverwendung in anderen Programmteilen möglich ist.

Ein anderes Problem: Schreiben Sie ein Programm, das jede lineare Funktion berechnen kann. Der Term hierfür lautet $y = a * x + b$. x, als Variable, soll auf dem Stack ganz unten liegen. Die Funktion (nennen wir sie LINEAR) soll folgenden Stackeffekt haben: LINEAR (x a b −− y). Dann kann man auf LINEAR aufbauend eine lineare Funktion mit konstanter Charakteristik definieren:

```
: FLINEAR1 ( x -- y ) 3 4 LINEAR ;
```

Und das Problem? Nach der Umformung kommt `a x * b +` heraus. Die drei Eingaben sollen aber hintereinander liegen. Durch noch eine Umformung könnte man der Lösung näherkommen: `b x a * +` gibt auch dasselbe. Aber wir wollen x a b als Parameter. So müssen wir b erstmal mit dem Wort –ROT (n1 n2 n3 −− n3 n1 n2) nach unten schieben. Die Lösung lautet also:

```
: LINEAR ( x a b -- a*x+b ) -ROT * + ; ⟨RET⟩  ok
```

Hätte es $a * x - b$ geheißen, wären wir noch nicht fertig. `a b -` ist nunmal nicht dasselbe wie `b a -`. Man müßte die oberen beiden Stackelemente vertauschen, dies erledigt SWAP (n1 n2 −− n2 n1):

```
: LINEAR2 ( x a b -- a*x-b ) -ROT * SWAP - ; ⟨RET⟩  ok
```

Machen wir es uns noch schwieriger: Schreiben Sie eine allgemeine Funktion, die eine Normalparabel aus ihren bekannten Nullstellen berechnet. Formel: $f(x) = (x-a)*(x-b)$

Nach der Umformung erhalten wir zuerst nur `x a - x b - *`, was leider ziemlich unbefriedigend ist. Besser ist da schon `x b x a - -ROT - *`. Die Übergabe lautet aber `x a b`, wobei ja die Reihenfolge von a und b egal ist. Wir müßten nun x zwischen a und b kopieren. Es gibt nun mehrere Lösungsmöglichkeiten:

```
: PARABEL1 ( x a b -- f ) 2 PICK SWAP - -ROT - * ; ⟨RET⟩  ok
: PARABEL2 ( x a b -- f ) >R OVER R> - -ROT - * ; ⟨RET⟩  ok
: PARABEL3 ( x a b -- f ) >R OVER >R - R> R> - * ; ⟨RET⟩  ok
```

Die erste Lösung enthält nur eine Neuigkeit: PICK (x0 .. xn n −− x0 .. xn x0) kopiert das n-te Stackelement nach oben. Dabei fängt die Zählung mit 0 an. 0 PICK wirkt also wie DUP, 1 PICK wie OVER (n1 n2 −− n1 n2 n1), der zweite neue Befehl, der das zweitoberste Stackelement (NOS, Next of Stack) nach oben kopiert.

Um die beiden weiteren Befehle (>R und R>) zu erklären, muß ich etwas weiter ausholen. Wir sprachen bisher immer nur von „dem Stack“, in Wahrheit gibt es deren zwei (manchmal auch noch mehr). Der zweite Stack ist der Returnstack, hier wird beim Aufruf einer Subroutine die Adresse des nächsten Befehls, die Returnadresse, abgelegt. Damit weiß das System am Ende eines Wortes, wo es weitermachen muß.

Diesen Returnstack kann man aber innerhalb einer Funktion eingeschränkt als Zwischenspeicher verwenden. >R (n −−) schiebt den TOS auf den Returnstack, R> (−− n) holt ihn wieder herunter. Dabei gilt auch, daß der letzte Wert zuerst wieder heruntergeholt wird. Als „Eselsbrücke“, um diese beiden Wörter nicht zu verwechseln: Der Pfeil „>“ deutet immer in die Richtung, in die geschoben wird. Bei >R deutet er auf das R (den Returnstack), bei R> deutet er von ihm weg, es wird also heruntergeschoben.

Den obersten Wert des Returnstacks kann man mit R@ (−− n) („R-fetch“) auf den „normalen“ Stack kopieren, ohne ihn vom Returnstack zu nehmen. Da der Returnstack für die Aufrufe von Wörtern und deren Subroutinen verwendet wird, kann man ihn vom FORTH–Dialog aus nicht benutzen. Es wird die Meldung „compile only“ ausgegeben. Der Returnstack wird schließlich für die Unterprogrammaufrufe und die Rückkehr aus ihnen verwendet, es darf zum Rücksprung also nur noch die Returnadresse draufliegen.

Und wenn die nicht stimmt? Genau: Es kracht im Gebälk, der Rechner wirft Bomben, bzw. eine Bus-, Address- oder Illegal Instruction–Error–Meldung, ganz nach Lust und Laune. Und genau das passiert Ihnen, wenn Sie im Programm einen Wert auf dem Returnstack vergessen haben, auch hier gibt es einen Crash. Deshalb: Der Returnstack muß vorsichtig benutzt werden.

Zählen Sie alle >R und R> in einem Wort, ihre Zahl muß auf alle Fälle gleich groß sein. Sollten innerhalb von Strukturen (IF .. ELSE .. THEN) Zugriffe auf den Returnstack sein, zählen Sie in jedem Abschnitt (dem Wahr– und dem Falsch–Zweig) alle >R und ziehen alle R> wieder ab, die Summe muß bei beiden gleich groß sein. Ganz sicher ist, wenn die Summe 0 ist. Sollte man trotzdem etwas falsch gemacht haben, so erkennt man das sehr schnell daran, daß das Wort einen Absturz verursacht.

Nun fehlen uns nur noch wenige Stackbefehle: ROT (n1 n2 n3 −− n2 n3 n1) ist der Gegenspieler von –ROT. ROT rotiert die drittunterste Zahl auf dem Stack nach oben, –ROT nach unten. Das „–“ steht für „nicht“, ein Befehl mit einem „–“ bewirkt etwas Gegenteiliges, oder gibt bei Erfolg eine false–Flag aus. –ROT kann man durch ROT ROT ersetzen, im figFORTH gab es den Befehl –ROT deshalb noch nicht.

Ähnlich, wie es zu DUP und OVER mit PICK eine Fortsetzung für beliebig viele Stackelemente gibt, so existiert das natürlich auch zu SWAP, ROT und –ROT. Die beiden Wörter (notwendigerweise braucht man deren zwei) heißen ROLL (n0 n1 .. nx x −− n1 .. nx n0) und –ROLL (n0 .. nx–1 nx x −− nx n0 .. nx–1).

Außerdem gibt es noch ein Wort namens UNDER (n1 n2 −− n2 n1 n2), das SWAP OVER ersetzt. Mit DROP (n −−) kann man das oberste Stackelement wegfallen lassen, wenn man es nicht braucht. Es gibt sogar NIP (n1 n2 −− n2), das Ihnen SWAP DROP erspart: NIP löscht den NOS.

Zu guter Letzt und ganz „klammheimlich“: Es gibt auch noch einen (destruktiven) Gegenspieler zu PICK, der kein Bestandteil des Standards ist: PIN (n0 n1 .. nx n x −− n n1 .. nx). PIN heftet den Wert n an die x-te Stelle auf dem Stack (x und n nicht mitgerechnet). Man erspart sich bei umfangreicheren Stackmanipulationen oft zeitraubende ROLL und –ROLL.

5. Und es gibt sie doch: Variablen — Hauptspeicherzugriffe

Umsteiger aus anderen Programmiersprachen wünschen es sich jetzt vielleicht Variablen, nachdem sie die Stackbefehle kennengelernt haben. Sie scheinen unentbehrlich zu sein. Geben Sei einmal ein:

```
VARIABLE␣A⟨RET⟩  ok
VARIABLE␣B⟨RET⟩  ok
VARIABLE␣C⟨RET⟩  ok
```

Anscheinend erlaubt auch FORTH die Verwendung von Variablen. Allerdings kann man einer Variablen nicht mit „A=5“ oder „A:=5“ einen Wert zuweisen. Um eine Zahl in einer Variablen zu speichern, verwendet man das Wort ! (n addr −−) (spricht “store”). Probieren Sie es aus:

```
5␣A␣!⟨RET⟩  ok
3␣B␣!⟨RET⟩  ok
```

Um aus der Variablen den Wert auszulesen, benutzt man das Wort @ (addr −− wert) (spricht "fetch").

```
A␣@␣.⟨RET⟩ 5  ok
A␣@␣B␣@␣+␣C␣!␣C␣@␣.⟨RET⟩ 8  ok
```

Vielleicht dämmert es Ihnen schon, auf welche Art FORTH nun die Variablen nachbildet, in den Stackeffektklammern steht da etwas von „addr", das heißt ja Adresse, und so ist es auch:

```
A␣.⟨RET⟩ 2220074  ok
```

(Diese Zahl ist völlig willkürlich gewählt, Sie erhalten sicher eine andere.)

Aha! Und wie unterscheidet FORTH nun eine Adresse von einer Zahl? Gar nicht! Sie selbst sind für die Interpretation der Stackelemente zuständig. So können Sie mit Adressen rechnen, wie mit normalen Zahlen auch, können sie genauso in Variablen speichern, können Zahlen als (z. B. System-)Adressen interpretieren, ohne auf ein Typenkonzept zu achten.

FORTH ist also eine typenlose Sprache. Der Compiler hat keine Möglichkeit, eine Typenüberprüfung durchzuführen. Sicher, das birgt Risiken, man vergibt eine Möglichkeit der Fehlerüberprüfung, die man in anderen Sprachen hat, spart sich aber die oft abenteuerlichen Konstruktionen, wenn man eben diese Hindernisse beiseite räumen muß.

Keine Angst, das führt nicht zu einem unsauberen Programmierstil, auch wenn @ und ! eine unverkennbare Ähnlichkeit mit PEEK und POKE haben, und sich auch so einsetzen lassen. Denn hier handelt es sich in der Tat um die einzige Verbindung zwischen Stack und Hauptspeicher (es gibt noch C@ und C!, das byteweise Zugriffe erlaubt, und bei 32–Bit–Systemen wie bigFORTH auch W@ und W! für 16–Bit–Zugriffe).

Es ist also eine ganz andere Situation als in Basic, wo sämtliche Sprachkonstrukte mit PEEK und POKE nichts zu tun haben, und diese beiden Wörter den Speicher als etwas externes, nicht zu Basic gehörendes betrachten. Der Hauptspeicher gehört zu FORTH. Der Zugriff darauf ist geordnet, eben z. B. durch dieses Variablenkonzept.

Die mit VARIABLE definierten Variablen sind global. Das erzeugte Wort legt eine Adresse auf den Stack, der Zugriff erfolgt also immer auf diese eine Adresse. Wird eine Variable an einem Ort im System geändert, so wirkt sich das auf das ganze System aus. Sie sind nicht dazu gedacht, Anfängern ihre Probleme mit dem Stack abzunehmen. Lokale Variablen liegen auf dem Stack und haben dementsprechend keinen Namen.

Auch wenn sie oft verpönt sind (ein Nachteil von Standard–Basic sind dessen globale Variablen), sie haben einen Zweck: Sie halten globale Systemzustände fest. Der Schreibzugriff zu solchen Variablen darf deshalb nur strengen Regeln folgen: Der Systemzustand muß sich geändert haben. Lesen darf man diese Variablen von überall. Ständig wechselnde Zustände, wie sie bei dynamischer Listenverwaltung auftreten können, erfordern solche Zugriffe sogar zwingend.

Solche Variablen sind Zeigervariablen, hinter deren unscheinbaren Äußerem sich viel verbirgt. Zeigervariablen haben als Inhalt wieder eine Adresse, die dann auf den eigentlichen (durch den Zeiger referenzierten) Inhalt deuten. Dieser kann weiter weisen, z. B. bei einer verkettete Liste, bei der der Zeiger auf einen weiteren Zeiger zeigt (und auf Daten, sonst wäre die Sache unsinnig). Der letzte Zeiger zeigt auf NIL (die Zahl 0), daran erkennt man das Ende der Liste. Auch Bäume kann man so aufbauen.

Sie sehen, das kleine Wort @ ist mitnichten „nur“ ein PEEK, es ersetzt die gesamte oft recht komplizierte Zeigerverwaltung in anderen Programmiersprachen. Der Klammeraffe “@” wurde von amerikanischen Händlern für das französische „à“ geschrieben, um die Preise den Waren zuzuordnen (“apples @ $1 per pound”). Man spricht das dann “at”. “At” bezeichnet aber auch einen Ort, so hat diese Abkürzung in FORTH durchaus ihren Sinn. Hier spricht man aber von “fetch”, „holen“. Der Klammeraffe hinter einem Wort bedeutet immer, daß irgendetwas geholt wird, so wie das Ausrufezeichen bedeutet, daß etwas gespeichert wird.

6. Von Schleifen, Flaggen, der Wahrheit und bedingten Anweisungen

Als Beispiel soll das Wort WORDS dienen. Sehen Sie sich das mit dem Decompiler SEE an (Einfach SEE und das zu untersuchende Wort eingeben):

```
SEE␣WORDS⟨RET⟩
: WORDS
CONTEXT @
BEGIN @ DUP STOP? NOT AND
WHILE ?CR DUP CELL+ .NAME SPACE
REPEAT DROP ;  ok
```

Zuerst wird also aus CONTEXT der Zeiger zum aktuellen Vokabular geholt, von diesem in der Schleife dann mit @ ein Element nach dem anderen. STOP? gibt true zurück, wenn ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ gedrückt wurden. Bei einem anderen Tastendruck hält es an, bis man wieder drückt und bricht hier genauso bei ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ ab. Das Symbol true wird durch die Zahl −1 ($FFFFFFFF) dargestellt, false durch den Wert 0. NOT (n −− n) führt ein bitweises not durch, macht also aus 0 −1 und aus −1 0, aber aus 2 z. B. −3 und aus −2 1.

WHILE (flag −−) springt bei der Zahl 0 auf dem Stack aus der Schleife heraus, bei jeder anderen Zahl macht es weiter. Die beiden Werte werden mit AND verknüpft, sodaß es dann zum Abbruch kommt, wenn nur eines false (also 0) ist. AND ergibt auch eine bitweise Und–Verknüpfung.

?CR beginnt eine neue Zeile, wenn der Cursor auf 16 oder weniger Zeichen an den rechten Bildschirmrand herangerückt ist, und verhindert dadurch, daß mitten im Wort umgebrochen wird. CELL+ addiert die Länge eines Stackelements in Bytes (in bigFORTH 4). Der Zeiger der verketteten Liste zeigt dann auf den Platz nach dem Verkettungszeiger, dort steht der Name des Wortes. .NAME gibt ihn aus. SPACE setzt noch ein Leerzeichen dahinter. Und REPEAT springt dorthin, wo BEGIN steht.

Und nun sind wir mittendrin im FORTH-System und verstricken uns im Gewirr . . . Wie war das mit den Flags? Nochmal ganz langsam: In FORTH wird jeder Wert als „wahr“ interpretiert, der nicht 0 ist. Probieren Sie es mal aus:

```
:␣->␣(␣Flag␣--␣)␣IF␣."␣Wahr"␣ELSE␣."␣Falsch"␣THEN␣;⟨RET⟩  ok
3␣␣␣␣␣␣␣␣->⟨RET⟩ Wahr ok
0␣␣␣␣␣␣␣␣->⟨RET⟩ Falsch ok
false␣␣␣␣->⟨RET⟩ Falsch ok
true␣␣␣␣␣->⟨RET⟩ Wahr ok
-1␣␣␣␣␣␣␣->⟨RET⟩ Wahr ok
```

```
-1␣not␣␣␣->⟨RET⟩ Falsch ok
3␣not␣␣␣␣->⟨RET⟩ Wahr ok
3␣not␣␣␣␣␣.⟨RET⟩ -4  ok
3␣0>␣␣␣␣␣->⟨RET⟩ Wahr ok
3␣4␣=␣␣␣␣->⟨RET⟩ Falsch ok
7␣7␣=␣␣␣␣->⟨RET⟩ Wahr ok
5␣7␣>␣␣␣␣->⟨RET⟩ Falsch ok
5␣7␣<␣␣␣␣->⟨RET⟩ Wahr ok
-4␣0<␣␣␣␣␣.⟨RET⟩ -1 ok
5␣3␣=␣␣␣␣␣.⟨RET⟩ 0 ok
```

TRUE und FALSE sind Konstanten mit dem Werten -1 (TRUE) und 0 (FALSE). Die Vergleichsoperatoren = > und < sind natürlich auch Postfixoperatoren. 0> ist eine Abkürzung für 0 > und testet, ob ein Wort positiv ist. Diese Operatoren legen als Ergebnis ihres Vergleichs eine Zahl (0 oder -1) auf den Stack. Da FORTH Zahlen als Flags akzeptiert, führt das zu dem etwas verwirrenden Ergebnis, daß sowohl auf 3 –> als auch auf 3 NOT –> „Wahr“ ausgegeben wird. In beiden Fällen ließt das IF einen Wert ungleich Null vom Stack und dieser wird als „wahr“ gewertet. Scheinbar fehlende Vergleichsoperationen wie <=, >= und <> („ungleich“) kann man durch > NOT, < NOT und = NOT ersetzten.

IF und THEN können auch ohne einen ELSE-Zweig benutzt werden. Die Besonderheit ist dabei, daß die Wahrheitsfindung (der Vergleich oder der Test) vor dem IF erfolgt und nicht, wie sonst üblich, zwischen IF und THEN. THEN stattdessen übernimmt die Rolle des ENDIFs, es ist etwa so gemeint:
Ist das wahr („Ist a=b“)? WENN: tu dies, SONST: tu das DANN: fahr fort.

In diesem Sinn ist die Sache verständlich. Auch die andere Form der Schleife, die BEGIN..WHILE..REPEAT-Schleife läßt sich abwandeln. Zum einen sind beliebig viele WHILEs erlaubt. Man hätte WORDS auch so formulieren können:

```
: words
  context @
  BEGIN @ dup WHILE
        stop? not WHILE
        ?cr dup cell+ .name space
  REPEAT drop ;
```

Jedes WHILE springt sofort hinter REPEAT, wenn es eine 0 auf dem Stack findet. Läßt man das WHILE ganz weg, so erhält man eine Endlosschleife. Ein sehr wichtiges Wort des Systems ist selbst eine Endlosschleife:

```
SEE␣(QUIT⟨RET⟩
: (QUIT
BEGIN .STATUS CR QUERY INTERPRET PROMPT
REPEAT ;
```

(QUIT wird von QUIT aufgerufen, es ist der Kommandointerpreter, der immer wieder mit QUERY eine Eingabe verlangt, den Prompt schreibt, .STATUS aufruft eine neue Zeile anfängt und von vorn. Bis Sie das FORTH–System verlassen, aber das geschieht durch eine Hintertür. Oder bis Sie QUIT eingeben, das startet (QUIT von neuem und löscht vorher den Returnstack. Dies geschieht auch bei jedem Fehler. Dabei wird dann kein Prompt ausgegeben.

Statt REPEAT können Sie eine Schleife auch mit UNTIL (flag −−) beenden, UNTIL springt solange zum BEGIN zurück, bis es keine 0 mehr auf den Stack vorfindet.

In Pascal heißt eine WHILE–Schleife abweisend, weil die darin enthaltenen Befehle nicht ausgeführt werden, wenn vor dem Durchlauf die Bedingung nicht erfüllt ist. Eine UNTIL–Schleife hingegen wird auf alle Fälle einmal durchlaufen. Abweisend ist die WHILE–Schleife in FORTH nur für den Teil hinter WHILE, der davor wird auf alle Fälle ausgeführt. Da das nicht nur der Wahrheitsfindung dienende Befehle sein können, sagt man besser: Mit WHILE kann man aus einer Schleife herausspringen. Jedenfalls kann man mit diesen vier Wörtern (BEGIN, WHILE, REPEAT und UNTIL) alle überhaupt möglichen bedingten Schleifen aufbauen (wobei UNTIL sogar durch die Sequenz NOT WHILE REPEAT ersetzt werden könnte).

Auch Zählschleifen sind in FORTH realisiert. Vielleicht holen wir hier nach, was angehende Basic–Programmierer schon nach kurzer Zeit können, nämlich n Sternchen ausgeben.

```
: stern Ascii * emit ; ⟨RET⟩  ok
: sterne ( n -- ) 0 ?DO stern LOOP ; ⟨RET⟩  ok
stern ⟨RET⟩ * ok
5 sterne ⟨RET⟩ ***** ok
0 sterne ⟨RET⟩  ok
```

Das Wort STERN soll uns hier nicht viel kümmern, es gibt einfach ein $*$ aus. Wenden wir uns lieber dem Wort STERNE zu, denn hier ist das Interessante. Die Schleife dreht sich also um ?DO (ende start −−) und LOOP. Wir hätten statt ?DO auch DO schreiben können, das hätte aber das unerwünschte Ergebnis, daß bei 0 STERNE der Rechner nicht mehr aufhört, Sterne auszugeben, DO zählt den Startwert einfach solange hoch, bis er den Endwert erreicht hat, und das wäre hier erst nach 4 294 967 296 (2^{32}) Sternen der Fall. ?DO dagegen bricht ab, wenn Start- und Endwert gleich sind.

Ja, FORTH ist schon etwas ungewöhnlich. Start- und Endwert werden in verkehrter Reihenfolge auf den Stack gelegt, Laufvariable gibt es anscheinend gar keine, und dann gäbe `FOR I=0 TO 5:PRINT "*";:NEXT I` ja auch nicht 5, sondern 6 Sterne! Was dahinter steckt, soll die nächste Schleife aufdecken:

```
: .INDEX ( end start -- ) ?DO I . LOOP ; ⟨RET⟩  ok
5 0 .INDEX ⟨RET⟩ 0 1 2 3 4  ok
```

Aha! Der Endwert wird gar nicht erreicht! Der Startwert wird hochgezählt, bis der Endwert erreicht ist, nicht bis der Index größer als der Endwert ist. Das ist eine Aufgabe von LOOP. Wenn also DO nicht schon gleich die Schleife überspringt, so zählt LOOP einfach stur hoch. Dabei bemerkt es nur, wenn der Endwert erreicht wurde, nicht aber, wenn er schon überschritten ist.

I ist keine Variable, sondern liest das Indexregister aus. Sie haben sich das wohl schon gedacht, sonst müßte ja mit @ darauf zugegriffen werden. In bigFORTH muß sogar ein zweites Register addiert werden, da LOOP die Bereichsüberschreitung (V–Flag) benutzt, um den Abbruch zu bemerken. In vielen FORTH–Systemen liegt das Indexregister auf dem Returnstack und I ist mit R@ identisch. Man darf also in Zählschleifen den Returnstack nicht benutzen. Auch in bigFORTH wird der alte Wert des Indexregisters auf den Returnstack gelegt. Eine eingeschränkte Benutzung ist dann zwar möglich, sollte aber aus Kompatibilitätsgründen vermieden werden.

Für Schleifen mit anderen Schrittweiten als 1 gibt es +LOOP (n −−), damit lassen sich sogar Schleifen mit während der Laufzeit wechselnden Schrittweiten realisieren — in

anderen Sprachen begibt man sich da oft in „Fallstrike“ nicht dokumentierter Eigenschaften.

7. Skalierte und doppelt genaue Zahlen

FORTH benutzt standardmäßig nur Ganzzahlen (Integer). Für bigFORTH gibt es auch eine Library, die die Anwendung von Fließkommazahlen erlaubt. Diese Library ist im Kapitel 10 dokumentiert, da sie auch nicht zum FORTH–Standard gehört. Ganzzahlen scheinen einen entscheidenden Nachteil zu haben: Brüche lassen sich damit nicht ausdrükken. Probieren Sie es aus:

```
1␣3␣/␣.⟨RET⟩ 0  ok
2␣3␣/␣.⟨RET⟩ 0  ok
3␣3␣/␣.⟨RET⟩ 1  ok
4␣3␣/␣.⟨RET⟩ 1  ok
5␣3␣/␣.⟨RET⟩ 1  ok
6␣3␣/␣.⟨RET⟩ 2  ok
```

1/3 ist also 0. 2/3 auch. 3/3 dann 1, genauso wie 4/3 und 5/3. Erst 6/3 ist 2, es wird also abgerundet. Aus diesem Grund wurde die Operation / erst hier zur Sprache gebracht. Es wird also wie in der Grundschule ganzzahlig geteilt. Da müßte dann auch ein Rest herauskommen. Probieren wir es aus:

```
1␣3␣/mod␣.␣.⟨RET⟩ 0 1  ok
2␣3␣/mod␣.␣.⟨RET⟩ 0 2  ok
3␣3␣/mod␣.␣.⟨RET⟩ 1 0  ok
4␣3␣/mod␣.␣.⟨RET⟩ 1 1  ok
5␣3␣/mod␣.␣.⟨RET⟩ 1 2  ok
6␣3␣/mod␣.␣.⟨RET⟩ 2 0  ok
```

Aha. Hier werden die ganzzahlige Division und der Modulowert zurückgegeben. Vielleicht noch der mathematische Unterschied zwischen Modulo- und Restwert:

```
-1␣3␣/mod␣.␣.⟨RET⟩ -1 2  ok
```

Der Modulowert ist immer positiv. Beim Restwert käme hier 0 Rest -1 heraus. Das Ergebnis von / ist also der ganzzahlig abgerundete Quotient. Was aber, wenn man keinen Rest brauchen kann, sondern wirklich zumindest mit einer Näherung von 1/3 arbeiten will?

Die Antwort ist relativ einfach: Stellen Sie sich vor, Sie haben einen Taschenrechner, mit dem Sie im Supermarkt Ihre Waren zusammenzählen. Die ganzen Zahlen verwenden Sie für die Mark, Pfennige sind Nachkommastellen. Wenn Sie das auf Ganzzahlen umstellen, stünden Sie zuerst auch vor Problemen, hätten aber sicher bald eine Idee: Sie rechnen einfach alles in Pfennigen. Und schon ist das Problem gelöst. Sie arbeiten dann mit skalierten Zahlen, mit denen sich Brüche und gemischte Zahlen wie $7\frac{1}{4}$ darstellen lassen. FORTH unterstützt diese Zahlen durch einen besonderen Operator: $*/$ (n1 n2 n3 −− n1 ∗ n2/n3). Jetzt können Sie 1/3 auf 5 oder 8 Stellen (ganz nach Wunsch) ausrechnen:

```
1␣100000␣3␣*/␣.⟨RET⟩ 33333  ok            (=100 000*1/3)
1␣100000000␣3␣*/␣.⟨RET⟩ 33333333 ok   (=10 000 000*1/3)
5␣100000000␣3␣*/␣.⟨RET⟩ 166666666 ok  (=10 000 000*5/3)
```

*/ ist nicht nur einfach * und / hintereinander ausgeführt, es hat ein doppelt genaues Zwischenergebnis. Damit können Sie Berechnungen ausführen, die den Wertebereich einer 32–Bit–Zahl sprengen würden, z. B. (1/3) * (4/7) auf 8 Stellen genau:

```
1 100000000 3 */ 4 100000000 7 */ 100000000 */ . ⟨RET⟩ 19047618  ok
```

Das Zwischenergebnis dieser Rechnung ist eine 64–Bit–Zahl, rechnen Sie sie mal aus:

```
1 100000000 3 */ 4 100000000 7 */ m* d. ⟨RET⟩ 1904761880952381  ok
```

Die hinteren 8 Stellen sind aber nicht brauchbar (eigentlich kommt bei der Rechnung 0, 190 476 190 476 190 476. . . heraus). Also werden sie weggeschnitten. Da immer abgerundet wird, ist auch die letzte Stelle nur bedingt brauchbar. Nun noch ein Tip, wie man statt dem Abrunden ein einfaches Runden (ab 0,5 wird aufgerundet) implementieren kann: Man teilt nur durch die Hälfte der Skalierung, addiert 1 dazu und teilt durch 2:

```
1 100000000 3 */ 4 100000000 7 */ 50000000 */ 1+ 2/ . ⟨RET⟩ 19047619  ok
```

Es zwingt Sie übrigens niemand, Zehnerpotenzen für die Skalierung zu verwenden, Sie können beliebige Werte benutzen. Stellen Sie sich z. B. vor, Sie wollen in Inches und Fuß rechnen (und als Ergebnis würde „Quadratfuß“ herauskommen), dann wäre als Skalar 12 sicher brauchbarer. Hier kann man sogar durch eine richtige Wahl des Skalars Rundungsfehler verweiden, die bei Fließkommazahlen häufig sind, da dort nur Brüche von Zweierpotenzen korrekt darstellbar sind, 0,1 oder 1/3 aber nicht.

Der Vorteil liegt hier in der Berechnungsgeschwindigkeit. Fließkommazahlen sind eine aufwendige Sache, eigentlich sind sie fast nur mit einem Coprozessor sinnvoll (ein paar wenige Anwendungen ausgenommen, bei denen die Ausführungszeit eine untergeordnete Rolle spielt). Ist bei */ das Skalar kleiner als 2^{16}, kann sogar der eingebaute Divisionsbefehl des 68000 verwendet werden, das geht dann insgesamt doppelt so schnell.

Als Übung können Sie ja die Parabelfunktion von Ganzzahlen auf Skalare umschreiben, wobei in einer Variable SCALING die Skalierung eingestellt werden kann (abhängig vom Ausgabegerät, z. B.). Hier sehen Sie auch gleich eine interessante Anwendung von globalen Variablen, denn das Wissen über das Ausgabegerät ist nicht Sache eines Programms, das nur Parabeln zeichnen kann. Dieser Parameter ist also nicht als Übergabeparameter gedacht. Die Nullstellen sollen weiterhin ganzzahlig angegeben werden.

Lösung:

```
Variable scaling  &10 scaling !
: parabel ( x a b -- y ) >r over >r scaling @ * -
  r> r> scaling @ * - scaling @ */ ;
```

Es gibt noch eine Reihe Varianten von /, /MOD, * und */, die wie M* mit doppelt genauen Zahlen zusammenarbeiten, oder mit vorzeichenlosen Zahlen. Sehen Sie dazu in den Referenzen im Kapitel „Integer–Arithmetik“ nach.

8. Formatierte Zahlenausgabe

Nun ist noch ein weiterer Aspekt wichtig: Wie bringt man solche skalierten Zahlen adäquat auf den Bildschirm? Es ginge nicht an, wenn man sich das Komma dazudenken muß. Gibt es in FORTH nicht so etwas wie PRINT USING? Natürlich gibt es das, es sieht sogar ganz ähnlich aus. Schreiben wir also eine Ausgaberoutine, die, sagen wir, 2 Nachkommastellen ausgibt (für Mark und Pfennig):

```
: .00 ( n -- ) <RET>  compiled
  extend under dabs <# # # ascii , hold #s rot sign #> type ; <RET>  ok

1234 .00 <RET> 12,34 ok
-19998 .00 <RET> -199,98 ok
23 .00 <RET> 0,23 ok
```

Nun im Einzelnen: Die Sequenz EXTEND UNDER DABS sorgt dafür, daß der Betrag der Zahl als doppelt genaue Zahl auf dem Stack liegt und darunter eine negative Zahl (-1), wenn die auszugebende Zahl negativ ist. Die formatierte Zahlenausgabe verlangt grundsätzlich doppelt genaue Zahlen, aber mit EXTEND ist das leicht zu bewerkstelligen. (EXTEND ist übrigens als `: EXTEND DUP 0< ;` definiert.) Vorzeichenlose Zahlen kann man mit 0 erweitern:

```
-1 0 d. <RET> 4294967295  ok
```

Der TOS ist bei doppelt genauen Zahlen der höherwertige Teil, der NOS (Next of Stack, liegt unter dem TOS) der niederwertige.

<# leitet die Zahlenausgabe ein. Es initialisiert den Puffer. Er wird von hinten her aufgebaut, deshalb werden die letzten Stellen zuerst verarbeitet. # wandelt die letzte Stelle der Zahl um, die Zahl ist dann durch die Basis dividiert, der Pufferzeiger wird um eins vorgesetzt. HOLD setzt ein beliebiges Zeichen in den String, ASCII <Zeichen> compiliert ein Zeichen als Literal, es steht dann im Code wie eine Zahl.

#S wandelt den Rest der Zahl, schreibt aber auf alle Fälle eine 0. Es hinterläßt eine doppelt genaue 0 auf dem Stack. #> (d -- addr count) schließt die Ausgabe ab, nimmt die doppelt genaue Zahl vom Stack und legt den Pufferanfang und dessen Länge auf den Stack, was von TYPE (addr count --) ausgegeben wird.

SIGN (n --) hängt ein „−“ an den String, wenn ihm eine negative Zahl übergeben wird. Das Vorzeichen kann dabei an jeder beliebigen Stelle im Zahlenstring stehen, ob vorne oder hinten, das können Sie bestimmen.

Der ganze Zahlenstring darf nicht länger sein, als die längste binär darstellbare Zahl im System, also in einem 32–Bit–System 64 Zeichen. Größer ist der Puffer nunmal nicht. Schaden wird das kaum etwas, da man längere Ausgaben, sofern man sie benötigt, ja aufteilen kann.

Die Zahlenbasis steht in der Variable BASE. Man kann sie also auch verändern. Das System ist normalerweise im Dezimalmodus. Mit HEX kann man es auf Hexadezimal umschalten, mit DECIMAL zurück. Auch beliebige andere Zahlenbasen kann man anwählen. Probieren Sie es einmal aus:

```
1234 hex . <RET> 4D2  ok
AAA 2 base ! . <RET> 101010101010  ok
111000 decimal . <RET> 56  ok
```

Man kann die Basis auch mitten in der Zahlenausgabe umschalten, man erhält dann eine Ausgabe in mehreren Zahlenbasen. Wozu das gut sein soll, sei jedem selbst überlassen, Basic bietet diese Möglichkeit mit PRINT USING jedenfalls nicht. Außerdem kann man beliebige Programme einbauen und damit allerhand anfangen, der Datumsstring im Editor (z. B. 28nov89) wird mit der Zahlenumwandlung erzeugt, der Monat wird einfach dazwischengehängt (siehe >DATE in FILEINT.SCR).

9. Codeaufbau

Vielleicht interessiert Sie das noch gar nicht, was der Compiler so compiliert: Hauptsache, er funktioniert. Doch die prinzipielle Funktion des Compilers ist in FORTH wichtig. Leider gibt es bei bigFORTH einige Schwierigkeiten mit der Erklärung, da es in diesem Punkt ganz gehörig vom Standard abweicht. Deshalb hier zuerst mal eine Erklärung, wie es z. B. in volksFORTH aussieht, und was man da machen kann, um die Geschwindigkeit nochmal enorm zu steigern.

Ein Wort besteht aus zwei Teilen, Header und Body, was man am besten mit Kopf und Rumpf übersetzt.

Im Header wird zuerst ein Verweis auf die Quelldatei und den Screen compiliert, aus dem das Wort stammt, im Direktmodus ist das eine 0. Dieser Verweis wird von dem Wort VIEW benutzt, um den Editor an der Stelle aufzurufen, an der das Wort steht. Das ist das View-Field, eine besondere Eigenschaft des Perry/Laxen-Modells, das auch hier übernommen wurde.

Anschließend steht der Zeiger auf die verkettete Wörterliste, den kennen wir schon. Hinter diesem Zeiger steht das Wort selbst als counted String. Diese Adresse wird NFA (Name Field Address) genannt. Das erste Byte ist hier die Längenangabe, der Text folgt danach. Die obersten drei Bits des Countbytes sind für andere Zwecke reserviert, dazu später. Die Länge des Feldes muß gerade sein, deshalb wird bei ungerader Länge noch ein Leerzeichen angehängt.

Dieses war der Kopf, anschließend folgt der Rumpf. Damit FORTH überhaupt weiß, was es tun soll, steht hier (wohlgemerkt, im volksFORTH!) ein Zeiger auf eine Assemblerroutine, die vom inneren Interpreter angesprungen wird. Diese Routine sorgt dafür, daß der Rest des FORTH-Wortes ausgeführt wird, oder bei einer Variablen deren Adresse auf den Stack gelegt wird. Es gibt noch einige andere Möglichkeiten, man kann sogar selbst Wörterklassen definieren, die ein einheitliches Verhalten haben.

Dieser Teil wird von : selbst erzeugt, danach wird mit] vom Interpreter auf den Compiler umgeschaltet. Der Compiler führt einzelne Wörter nicht aus, sondern schreibt ihre Adressen ins Wörterbuch. Dieser Teil des Wortes heißt Parameter Field, die Anfangsadresse dann PFA. ; compiliert noch UNNEST und schaltet den Compiler mit [aus. Der innere Interpreter liest diese Adressen und springt die Routine an, auf die dort im Code Field verwiesen wird.

Die letzte Adresse in einem Wort zeigt auf das Wort UNNEST, das den Rücksprung ausführt. Zahlen (Literals) werden hinter den Befehl LIT geschrieben, der die Zahl auf den Stack legt und die Rücksprungadresse erhöht, ebenso Strings hinter den Befehl "LIT. Der Teil des FORTH-Systems, in dem die Wörter stehen, wird Wörterbuch (Dictionary) genannt. Sein Ende, gleichzeitig der Ort, an dem neue Wörter compiliert werden, wird HERE genannt. Diese Adresse legt das gleichnamige Wort auf den Stack.

Der innere Interpreter besteht aus zwei Teilen, zum ersten aus dem Makro NEXT, mit dem jedes Assemblerwort (Primitive) abgeschlossen ist, und dem Programm DOCOL, auf das die CFA eines jeden Colon-Wort (Colon=":") weist. NEXT liest die nächste Adresse ein und springt an die CFA. DOCOL legt den aktuellen IP (Instruction Pointer) auf den Returnstack und setzt die Adresse, von der es aufgerufen wird (erhöht um eine Zeigerlänge) als neuen IP.

Kurz gesagt: Das was FORTH erzeugt, kann man mit Recht als Maschinencode für einen (im ST nicht eingebauten) Prozessor bezeichnen. Damit unser 68000 im volksFORTH weiß, was er tun soll, gibt es den inneren Interpreter und die Primitives, die in Assembler geschrieben sind und deren CFA direkt hinter sich deutet.

Nun, das führt zwar zu kompaktem und auch recht schnellem Code, aber man fragt sich (mit Recht!), ob sich ein Prozessor nicht besser nutzen läßt, der doch viel mehr kann als nur Zeiger lesen und springen. Viel mehr tut er ja in einem klassischen FORTH nicht.

Springen kann der Prozessor selbst. Auch die Verwaltung eines Returnstacks beherrscht er natürlich, da man Subroutinen auch von Assembler aus aufrufen können muß. Was liegt es da näher, den inneren Interpreter wegzurationalisieren, indem man das dem Prozessor überläßt? Dazu muß man lediglich 32–Bit–Adressen verwenden und mit jsr (Jump to SubRoutine) anspringen. Gleichzeitig stößt man die lästige Begrenzung von 64 KByte Adressraum um, also noch ein Vorteil!

In einem solchen FORTH benötigt man die CFA nicht mehr. Colon–Wörter und Assemblerwörter sind gleich aufgebaut, da der Prozessor beide direkt versteht. Die Adresse für die verkettete Liste ist 32 Bit groß (statt vorher 16 Bit), das View–Field kann so bleiben wie es ist. Andere Wörter, z. B. Variablen bekommen statt der CFA einen jsr–Befehl auf die entsprechende Routine. Die braucht nur die Adresse vom Returnstack zu lesen und weiß, woher sie aufgerufen wurde.

Nun fällt auch auf, daß viele einfache Befehle in FORTH auch auf dem 68000 durch einen oder ein paar wenige Befehle ausgedrückt werden können, was liegt näher, als bei diesen Befehlen auf den mühsamen Weg über jsr Adresse zu verzichten und sie als Makros direkt in den Code zu schreiben?

Gesagt, getan. Nur braucht man dann noch eine Angabe, wie lang denn so ein Makro ist. Hierzu fügen wir ein 16-Bit-Feld (länger wird wohl ein Wort kaum werden) zwischen Kopf und Rumpf ein. Da Befehlssequenzen auf dem 68000 immer eine gerade Länge haben, benutzen wir das unterste Bit gleich, um Makros zu markieren — ein ungerades Längenfeld bedeutet also für den Compiler, daß er kein jsr adresse compilieren, sondern das Wort (abzüglich 2 Bytes für das rts am Ende) als Makro kopieren soll.

Nun ist das Produkt sicher schon ganz schön schnell, nur fallen beim Disassemblieren dieses Codes ein paar „grauenhafte“ Stellen auf. Da steht doch direkt untereinander `move.l D0,-(A6)` und `move.l (A6)+,D0`, was bedeutet: Das Datenregister D0 wird erst auf den Stack (A6) geschoben und anschließend wieder zurückgeladen. Diese zwei Befehle kann man einfach weglassen.

Es gibt noch ein paar andere Kombinationen, sogar einige, bei denen mehr gespart wird; die meisten, bei denen es sinnvoll ist, werden vom Compiler auch abgekürzt. Hierzu muß er natürlich noch wissen, wie das Makro aufgebaut ist, zumindest am Anfang und am Ende, und wie er es dann verkürzen kann. Dazu brauchen wir ein weiteres Wortfeld (genauer: 2 Bytes, eines für den Wortanfang, das andere für das Ende). Dieses hängen wir der Einfachheit nur bei Makros hinten an, sind beide Bytes 0, so wird ohne Verkürzungen compiliert. Diese Form der Optimierung nennt man “Peephole–Optimierung” („Guckloch-Optimierung“), weil der Optimierer nur einen kurzen Code–Ausschnitt betrachtet und hier seine Optimierungen ansetzt.

Nochmal zur Übersicht:
Felder (darunter Länge in Bytes):

View	Link	Count	Name	(Blank)	Length	Code,rts	(Take	Push)
2	4	1	Count	0/1	2	Length	1	1

Die zusätzlichen Bits im Count–Byte und das Bit 0 im Length–Byte wurden hier zur besseren Übersicht weggelassen. Der Makro–Deskriptor ist in ein Take- und ein Push-Byte aufgeteilt, die Anfang und Ende eines Makros beschreiben. Genaueres dazu finden Sie im Kapitel 4.19.

Zu den Bits in Count (darunter Länge in Bits):

restrict	immediate	indirect	count
1	1	1	5

Da count nur ein 5–Bit–Feld ist, dürfen Wörter in FORTH grundsätzlich nicht länger als 31 Buchstaben sein, eine Einschränkung, mit der man sicher leben kann, schließlich sind in bigFORTH auch alle Buchstaben signifikant.

Das indirect–Bit zeigt an, daß statt des Length–Fields ein Zeiger auf den eigentlichen Wortrumpf steht, mit Hilfe dieses Bits kann man bereits definierte Wörter umbenennen oder den Wortkopf statt ins Directory auf den Heap schreiben, wo er später gelöscht werden kann.

immediate und restrict sind zwei Eigenschaften, die sich auf das Compilationsverhalten beziehen. Funktionell ist das immediate–Bit wichtiger. Das restrict–Bit dient nur dazu, Fehler abzufangen. Ein Wort, bei dem es gesetzt ist, darf vom Interpreter nicht ausgeführt werden, es führt dann zur Meldung „compile only“. Solche Wörter kennen wir schon: Es sind R@, R> und >R, auch die Strukturwörter IF, ELSE, THEN, BEGIN, WHILE, REPEAT und UNTIL gehören dazu.

Allerdings findet man im compilierten Code kein jsr auf diese Wörter (auch wenn der Decompiler SEE dies verschweigt), sondern nur zwei Makros, BRANCH und ?BRANCH und dahinter ein Wort, das die Sprungweite angibt (negativ heißt nach vorne). BRANCH springt immer, ?BRANCH (flag −−) nur, wenn es eine 0 vorfindet. Damit lassen sich alle bisher bekannten FORTH-Strukturen compilieren.

Wie? IF compiliert einen ?BRANCH, der entweder hinter das ELSE oder wenn nicht vorhanden hinter das THEN springt, ELSE compiliert einen BRANCH, der hinter das THEN springt. THEN trägt nur die Sprungweite nach.

BEGIN legt die Anfangsadresse der Schleife auf den Stack. WHILE compiliert ein ?BRANCH, das hinter REPEAT oder UNTIL springt. REPEAT compiliert einen BRANCH zum BEGIN und trägt in alle WHILEs die Sprungweiten nach, genauso UNTIL, das aber einen ?BRANCH compiliert.

Diese Wörter müssen während der Compilezeit aufgerufen werden. Dazu ist bei ihnen das immediate–Bit gesetzt. Dieses Bit zwingt den Compiler, ein Wort zu interpretieren, statt zu compilieren. Sehen wir uns mal z. B. IF an.

```
: IF   compile ?branch >mark 1 ; immediate restrict
```

Der Stackeffekt wurde hier absichtlich weggelassen, da er etwas irreführend wäre. Was macht also IF? Es compiliert offensichtlich ?BRANCH, legt mit >MARK eine Marke an und deren Adresse auf den Stack, sowie eine 1, die als Kennung dient. Eigentlich wäre der Stackeffekt von IF dann (−− marke 1), zumindest der während des Compilierens.

Im Programm aber ist der Stackeffekt von ?BRANCH (flag −−) wichtig, ein IF im Programm wirkt sich so aus. Man muß bei immediate–Wörtern also streng zwischen ihrem compiletime–Verhalten und ihrem runtime–Verhalten (also während des Programmablaufs) unterscheiden. Diese zwei Verhaltensweisen sind grundverschieden.

COMPILE ist übrigens auch ein immediate-Wort:

```
: compile  compile (compile ' A, ; immediate restrict
```

Wundern Sie sich nicht, wie COMPILE es anstellt, (COMPILE mit sich selbst zu compilieren, es tut das ja nicht im FORTH–System, sondern im Targetcompiler, und dort ist COMPILE schon vordefiniert, compiliert also ohne das Zutun des gerade definierten Wortes (COMPILE und die CFA des folgenden Wortes. ' liefert übrigens diese Adresse. A, compiliert eine Adresse ins Wörterbuch.

Eigentlich müßte es „,“ heißen, denn , compiliert eine Zahl, und eine Adresse ist ja auch fast nur eine Zahl, aber leider nur fast. A, setzt nämlich noch in einem Bitstring eine Marke, daß hier eine Adresse steht. Dieser Bitstring hat nur beim Laden oder Sichern des Systems eine Funktion, genaueres erfahren Sie im Kapitel 4.24.

Nun, Sie sehen, Sie können in FORTH neue Programmstrukturen einbauen, Programme schreiben, die in andere Programme etwas hineincompilieren, oder sogar (ganz dreist) andere Programme (Wörter) erzeugen. Das sind mächtige Möglichkeiten, die Sie nur in wenigen anderen Sprachen finden.

Aber solche Wörter wie VARIABLE? Die kein Programm, sondern einen Datentyp erzeugen? Natürlich, auch die kann der Benutzer selbst definieren, VARIABLE ist ja auch nur ein Programm. Das beste Beispiel ist es nicht, nehmen wir lieber das Wort CONSTANT. Es soll andere Wörter erzeugen, die einen Wert auf den Stack legen. CONSTANT soll diesen Wert vom Stack nehmen. Das erzeugte Wort soll ihn wieder zurücklegen. Wir müssen also einen erzeugenden Teil (engl. CREATE) und einen beschreibenden Teil definieren, in dem steht, was das Wort tun soll (engl. DOES). So sieht das dann aus:

```
: constant ( n -- )
  Create ,
  DOES> ( -- n ) @ ;
```

CREATE erzeugt also einen Wortkopf mit CFA. Diese CFA zeigt zuerst auf ein leeres DOES> in Create. DOES> setzt sie neu auf die Stelle hinter sich, an die es zur Compilezeit schon ein R> compiliert hat (DOES> ist ein immediate word, das Wort, das ausgeführt wird, heißt (;CODE, ist aber nicht sichtbar). Dieses R> legt die PFA des Wortes auf den Stack, von dem der Teil hinter DOES> aufgerufen wurde. Das ist also der Runtime–Stackeffect von DOES>. Das @ holt von dieser Stelle einen Wert, dorthin hat vorher , den Wert der Konstante compiliert.

Natürlich sind das Wort CONSTANT und noch eine Reihe weiterer solcher Wörter bereits im System definiert. Diese Methode, Wörter mit gleichem oder zumindest sehr ähnlichem Verhalten zu erzeugen, ist so mächtig, daß z. B. auch die ganze AES–Library von einem definierenden Wort namens AES' erzeugt wird.

Diese Eigenschaft gibt FORTH den Touch einer objektorientierten Sprache, allerdings gibt es nur eine mögliche Nachricht: Das Wort wird aufgerufen. Von einer richtigen „Vererbung“ (“Inheritance”) im Sinne von SMALLTALK kann auch nicht die Rede sein. Es gibt nämlich nur eine Methode, die ein Wort einer Klasse ausführen kann, das Erben von Eigenschaften einer Überklasse ist auch nur sehr eingeschränkt möglich.

Trotzdem: Der Komplex CREATE .. DOES> ist ein mächtiges Werkzeug von FORTH, das gekonnt eingesetzt viel bringt.

Sie haben gesehen, der FORTH–Compiler selbst ist sehr einfach aufgebaut. Wie Strukturen aussehen, wie man Schleifen baut, wie Datentypen geformt sind, all das weiß er nicht selbst. Selbst das erste, was ein Compiler überhaupt konnte, Ausdrucksverwertung, ist ihm völlig fremd. Trotzdem ist er nicht hilflos, denn er kann etwas, was andere Compiler nicht können: Wörter ausführen. Hier kann man ihn erweitern. Praktisch grenzenlos. Der Compiler kann ohne das FORTH–System nicht existieren. Er benutzt es ständig. Interpreter, Compiler und ausführende Einheit (allgemein „Innerer Interpreter“ genannt) sind eng miteinander verzahnt.

Ein Wort, das Parabelwerte berechnet, braucht kein INPUT, um mit dem Benutzer zu kommunizieren. Es bekommt seine Werte vom FORTH–System. Natürlich kann es auch mit dem Benutzer wie in Basic in Verbindung treten, FORTH stellt dafür Worte zur Verfügung. Oder als GEM–Programm, das ist sicher eine bessere Anwendung. Ein solches

Programm braucht keinerlei Anhaltspunkte für seine FORTH–Herkunft zu besitzen, ein Funktionsplotter kann auch Ausdrucksverwertung enthalten, auch wenn es in UPN viel leichter wäre.

10. Massenspeicher

Ein Massenspeicher ist das kleine Floppylaufwerk im (oder am) ST, das kaum dazu überredet werden kann, den Inhalt des ST–RAMs auf einmal zu schlucken. Da aber auch die Festplatte, die das TOS 1.0 oft zum Absturz und den Benutzer zur Verzweiflung bringt, zu den Massenspeichern gehört, haben diese Scheiben vielleicht doch den richtigen Namen, schließlich gab es früher auch noch nicht soviel Hauptspeicher.

Massenspeicher sind langsam. Und sie geben viele Daten auf einmal her, mindestens einen Sektor. Das sind auf einer ST-Disketten und den meisten Festplatten schon 512 Bytes, ein halbes KByte. Durch diese Menge (im Vergleich zu den 16 Bit des Hauptspeichers) machen sie einen Teil ihrer Lahmheit wieder wett — zumindest Festplatten, deren maximaler Durchsatz in die Nähe des Hauptspeichers kommt.

Hauptspeicher dagegen sind schnell und geben wenig auf einmal her, der ST hat einen 16–Bit–Speicher, also kommen gerade 2 Bytes auf einmal zum Prozessor. Wie man in FORTH auf den Hauptspeicher zugreift, wissen wir ja schon, aber wie greift man auf den Massenspeicher zu?

Auf alle Fälle muß mindestens ein Sektor in einen Bereich des Hauptspeichers geladen werden. Hier kann man sich dann die benötigten Daten heraussuchen. Der Ort, an dem das geschieht, heißt Sektorpuffer.

Damit haben wir schon fast das Konzept, das FORTH bietet. Anstelle von Sektoren, die von Gerät zu Gerät verschieden sind, lädt man gleich einen Block, das sind in FORTH immer 1024 Bytes (ein KByte). Die Adresse des Blockpuffers wird zurückgegeben. Anfordern kann man so einen Block mit dem Wort BLOCK (n −− addr). Die Adresse kann sich ändern. Sobald auf die Blockpuffer zugegriffen wird (es gibt mehrere), kann sich da schon etwas ändern, und das kann man in bigFORTH nicht genau vorhersagen, schließlich ist Multitasking möglich. Die Adresse soll also sofort benutzt werden.

Dieser Blockpuffer ist Hauptspeicher. Man kann auf die Daten so zugreifen wie auf Hauptspeicher. Veränderungen werden nicht automatisch zurückgeschrieben. Jeder Puffer verfügt über Verwaltungsinformationen, eine davon ist die Update–Flag, sie gibt an, ob der Puffer zurückgeschrieben werden muß, ehe er freigegeben wird. Setzen kann man diese Flag mit UPDATE, allerdings muß man den betreffenden Block direkt davor angefordert haben, seine Adresse muß noch sicher sein. UPDATE markiert den zuletzt gelesenen Puffer.

Wieviele Puffer es tatsächlich gibt, darf für den Programmierer nicht von Belang sein. Mindestens muß es zwei geben. Tatsächlich aber können bei bigFORTH sehr viele Puffer im Speicher gehalten werden, Limit ist hier nur der verfügbare Speicherplatz („Cache–RAM“). Damit braucht man nicht ständig vom Massenspeicher nachladen und hat trotzdem Zugriff auf dessen Daten. Die Puffer werden in bigFORTH als LRU–Puffer (Last Recently Used–Puffer) verwaltet, wie das in den meisten FORTH–Systemen üblich ist. Bei Speichermangel wird also der am längsten nicht benutzte Puffer verdrängt. Wie das alles genau funktioniert, erfahren Sie im Kapitel 7.1.

Natürlich kann man auch FORTH-Quelltexte auf Diskette speichern. Diese Programmblöcke nennt man Screens, auch wenn sie beim ST flächenmäßig gerade eine halbe Bildschirmseite belegen.

Diese Screens werden vom Interpreter wie Kommandozeilen bearbeitet. Man muß sie dazu mit LOAD (n −−) laden. Es gibt noch ein paar andere Befehle, die alle auf LOAD zurückgreifen: THRU (start range −−) das von start an range Screens lädt und ihre Brüder +LOAD und +THRU, die relativ arbeiten. Sie sind sinnvoll, wenn man in einem Screen von anderen lädt, da man dann nur die relative Position angeben muß, nicht aber die absolute. Und `-->` lädt von einem Screen aus direkt den nächsten, ohne wieder zum Screen zurückzukommen. Da es ein immediate Word ist, kann man es auch innerhalb einer Programmdefinition anwenden.

In bigFORTH kann nicht nur direkt auf die Massenspeicher zugreifen, sondern auch auf TOS–Dateien. Das ist sogar die bevorzugte Zugriffsweise, da man dann nicht mit dem Betriebssystem in Konflikt kommt. Woher sollte es wissen, daß eine Diskette von FORTH aus benutzt wurde, wenn nur ein paar Blöcke geändert, aber keine Verwaltungsinformation hinterlassen wurde?

Einen Screen einer beliebigen Datei kann man mit LOADFROM <Datei> (n −−) laden, INCLUDE <Datei> lädt den ersten Screen, der der Loadscreen sein sollte. LOAD, THRU, +LOAD und +THRU arbeiten mit der aktuellen Datei zusammen. Mit LIST (n −−) lassen Sie einen solchen Screen ausgeben.

Der Aufwand für diese Massenspeicherverwaltung ist ziemlich gering. Berücksichtigen Sie, daß FORTH standardmäßig keinerlei Dateihandling besitzt. BLOCK besteht also aus nicht mehr als einer Pufferverwaltung und der eigentlichen Leseroutine, die den Controller überhaupt zur Arbeit bringt. In bigFORTH greift man zwar hauptsächlich auf TOS–Dateien zu, hier wird also ein bereits vorhandenes Betriebssystem benutzt, aber beim Zugriff auf Blöcke auf eine effektive Art und Weise, denn das TOS kann dann den Schreib/Lese–Befehl direkt an den Treiber weiterleiten — es wird nicht sehr gebremst.

Ein sehr gutes Beispiel für Massenspeicherzugriffe finden Sie in `UNSINN.SCR`, in dem die Zeilen der Screens einer Datei als Grammatiksprache interpretiert werden. Da dieses Programm aber weiter reicht, als unsere bisherigen Erkenntnisse, ist es im Kapitel 11.1 gesondert erklärt.

Wollen wir uns deshalb nur ein einfaches Beispiel ansehen, das Wort LIST, das einen Block als Screen ausgibt. Es dokumentiert den Gebrauch ganz gut:

```
: list ( n -- )
  scr !  3 spaces  file?  ." Scr " scr @ dup u.  ." Dr "  drv?  .
  l/s 0 DO
    cr I 2 .r space scr @ block I c/l * + c/l -trailing type
  LOOP cr ;
```

Zuerst wird der zu listende Screen in SCR gespeichert. Diese Variable enthält den Screen, den der Benutzer gerade editiert, aus historischen Gründen muß LIST diese Variable beeinflussen (es gehört eigentlich zum Editor). Der Rest der ersten Zeile dient dazu, die benutzte Datei, den zu listenden Screen und das Laufwerk auszugeben (wenn es ein Direktzugriff ist, sonst heißt es immer „Dr 0“). ." ¡String¿" gibt einen (als Stringliteral gespeicherten) Text aus. Das erste Leerzeichen wird zur Abgrenzung zwischen Befehl und String benutzt.

L/S heißt Lines per Screen und ist eine Konstante: Es gibt 16 Zeilen pro Screen und 64 Zeichen pro Zeile (C/L). In der zweiten Zeile wird also eine Schleife von 0 bis 16 (ausschließlich) gestartet.

Nun wird eine Zeile ausgegeben. CR (Carriage Return, Wagenrücklauf) sorgt dafür, daß dies am Anfang der nächsten Bildschirmzeile geschieht. Dann wird mit .R (n chars −−) rechtsbündig in einem zwei Zeichen breiten Feld die Zeilennummer ausgegeben, gefolgt

von einem Leerzeichen (SPACE). Nun wird mit SCR @ BLOCK die Pufferadresse des Screens geholt, der Screen wird dann bei Bedarf geladen.

Zu diesem Zeiger wird die Länge einer Zeile mal die auszugebende Zeile addiert, da ein Zeichen ein Byte belegt, werden die bereits ausgegebenen Zeilen dadurch übersprungen. C/L gibt die Länge an, es soll eine Zeile ausgegeben werden. Eigentlich könnte das TYPE gleich erledigen, –TRAILING unterdrückt nur die Ausgabe der abschließenden Leerzeichen. Da sie sowieso unsichtbar wären, wird die Ausgabe schneller. Damit ist die Schleife schon beendet.

TYPE kennen Sie schon, es wurde schon mal benutzt, um einen Zahlenstring auszugeben. Damals lag der String an einer praktisch festen Adresse, nun geben wir mit demselben Wort einen Text vom Massenspeicher aus — er liegt schließlich im Hauptspeicher. Vielleicht haben Sie schon erkannt, daß TYPE (addr count −−) eigentlich nur einen ASCII–Dump liefert — aber es wird doch ganz korrekt eingesetzt, oder hatten Sie den gegenteiligen Eindruck?

Damit ist die Schleife von LIST auch schon zuende, am Schluß wird nochmals mit CR umgebrochen, damit der Interpreter sein `" ok"` in die nächste Zeile schreibt und es nicht so aussieht, als gehöre es zum Screen.

11. Das Terminal

Die Kommunikation mit dem Benutzer wird bei kaum einer Sprache völlig dem Betriebssystem überlassen — alle Sprachen bieten einen Standard, der zumindest ausreicht, Meldungen und Ergebnisse auf den Bildschirm zu schreiben oder an den Drucker zu senden. Bildschirme als Terminals gibt es noch nicht so lange. Die Befehle von FORTH waren deshalb ursprünglich für einen Drucker gedacht, was ja auch bei anderen Sprachen zu beobachten ist.

Ein paar wenige Befehle kennen wir schon: TYPE, EMIT und CR. Ihre Druckerherkunft können sie nicht leugnen. EMIT (char −−) gibt ein Zeichen aus, TYPE (addr count −−) mehrere auf einmal, und CR beginnt eine neue Zeile. Was brauchen wir noch? DEL löscht das letzte Zeichen und rückt den Cursor (oder den Druckkopf) dabei ein Zeichen zurück. Ein neues Blatt kann mit PAGE eingezogen werden, der Bildschirm wird mit diesem Befehl gelöscht.

Für Tabellen mag auch interessant sein, wo der Cursor/Druckkopf gerade ist, AT? (−− row col) liefert die Zeile und Spalte. Mit AT (row col −−) kann man den Cursor dorthin setzen, wo man es wünscht, viele Drucker werden da aber nur mitmachen, wenn die neue Position unterhalb der alten ist (außer bei Verwendung von Formulartraktoren).

Diese Wörter sind Standard. In bigFORTH gibt es noch ein paar mehr, die auch ganz brauchbar sind: FORM (−− rows cols) gibt das Format des Papiers oder Bildschirms aus, man kann damit die Ausgabe dem Format anpassen, z. B. zentriert ausgeben oder Tabellen, die trotzdem den ganzen Platz benutzen, mit Überschriften oder Seitennummern versehen.

Für den Bildschirm sind die Worte CURON und CUROFF recht nützlich, die den Cursor ein- und ausschalten. CURLEFT und CURRITE dienen der vereinfachten Cursorsteuerung.

Zuletzt kann man mit CLRLINE noch die Zeile löschen, auf der der Cursor steht. Dieses Wort ist ganz sinnvoll, wenn nur eine Zeile neugeschrieben werden muß, wie das nach einem UNDO im Kommandoeditor erforderlich ist.

Zum Ausgabegerät des Computers gehört auch ein Eingabegerät für den Benutzer. Das ist normalerweise die Tastatur und die wird von FORTH auch unterstützt. Allerdings kann

man auch andere zeichenorientierte Geräte (Lochkartenleser, Modems o. ä.) so benutzen. KEY (−− key) wartet auf einen Tastendruck und gibt den ASCII–Code der Taste zurück. Wurde schon vorher eine Taste gedrückt, so wird das erste Zeichen im Tastaturpuffer zurückgegeben.

In bigFORTH wird auch noch der Scancode geliefert, da etliche Atari–Tasten nur durch ihn identifiziert werden können (die Funktions–, Cursor–, Help– und Undo–Tasten). Der Scancode und der ASCII–code bilden zusammen ein 16–Bit–Wort, wobei der Scancode im höherwertigen Byte liegt. Für Abfragen, die nur den ASCII–code benötigen, muß man also KEY $FF AND schreiben.

Ideal ist das leider nicht, da die Scancodes von der Tastenposition abhängen, die ASCII–werte aber vom Treiber — so ist der Scancode von Z in Deutschland $15, in Amerika aber $2C, da dort Y und Z vertauscht sind. Und die </>–Taste mit dem Scancode $60 gibt es dort gar nicht.

Ob überhaupt eine Taste gedrückt wurde, kann man mit KEY? (−− flag) erfragen. Der Tastaturpuffer bleibt davon unberührt, es gibt auch keine Wartezeiten. So kann man einen Abbruch bei Tastendruck realisieren, STOP? (−− flag) z. B. macht davon Gebrauch.

Einen ganze Zeile einlesen kann man mit EXPECT (addr count −−), das entweder am Pufferende oder bei einem Druck auf ⟨RET⟩ endet. In der Variablen SPAN stehen dann die eingelesenen Zeichen. EXPECT bietet eine Menge Editiermöglichkeiten, verarbeitet werden sie von DECODE (addr pos1 key −− addr pos2).

Diese Befehle sind keine direkt definierten Wörter. Sie rufen alle nur gerätespezifische Wörter auf. Die Adressen dieser Wörter stehen in zwei Arrays, die man mit INPUT: ⟨***Wort***⟩ {<Gerätesp Wort>} [oder OUTPUT: (wird genauso angewendet) definieren kann. Die so definierten Wörter dienen zum Umlenken der Ein- und Ausgaben. Sie schreiben die Adresse des Arrays in die User–Variablen INPUT und OUTPUT.

User–Variablen sind Variablen, die nicht für das ganze System, sondern nur für einen einzigen Task global sind. In anderen Tasks können sie einen anderen Inhalt besitzen. Auch BASE ist eine solche Variable. Definiert werden sie mit USER ⟨***Name***⟩.

Die Ausgabe läßt sich in bigFORTH auf den Drucker umleiten (mit >PRINTER im Vokabular PRINTER), oder auf Dateien (dazu muß man `FILEIO.SCR` dazuladen), hier kann man zwischen Screenfiles (im FORTH–Format) und ASCII–Files wählen.

Man könnte problemlos auch eine Umleitung über die serielle Schnittstelle schreiben, den Interpreter in einem zweiten Task mit dieser Ein/Ausgabe starten und über Modem einen zweiten Benutzer mitspielen lassen. FORTH ist von Haus aus multiuserfähig, allerdings müßte man den weiteren Benutzern die Möglichkeit nehmen, Wörter zu definieren, da es dabei zu Konflikten kommen könnte.

12. Interpreterinterne Befehle

Grob kennen wir es schon: Der Interpreter liest eine Zeile ein, pickt sich Wort für Wort heraus und interpretiert es oder wandelt es in eine Zahl. Das Einlesen geschieht im Wort QUERY.

```
: query ( -- )
  tib &80 expect  span @ #tib !  >in off blk off ;
```

TIB, 80 Zeichen lang, ist also der Puffer für die Eingaben. Die Länge der Zeile wird in #TIB gespeichert. >IN gib an, wieviele Zeichen bereits interpretiert wurden, es wird mit OFF auf 0 gesetzt, ebenso wie BLK, in dem steht, von welchem Block geladen wird.

Block 0 heißt dabei, daß vom TIB interpretiert wird, deshalb kann man Block 0 einer Datei auch nicht laden.

Der Interpreter pickt sich nun mit NAME (−− addr) die Wörter aus dem Text, sucht mit FIND (string −− cfa t / string f), ob sie vorhanden sind, wenn ja, werden sie ausgeführt, andernfalls versucht er mit NUMBER? (string −− string false / n true / d 0>), sie in Zahlen umzuwandeln. Schlägt alles fehl, bricht er mit ABORT" Hä?" ab. Wörter, die restrict sind, werden auch nicht ausgeführt. Der Compiler geht prinzipiell genauso vor, compiliert aber alle Wörter, die nicht immediate sind, ebenso wie die Zahlen (letztere mit LITERAL (n −−) (immediate)).

NAME besteht aus BL WORD CAPITALIZE. WORD (char −− addr) überspringt im Puffer zuerst alle Leerzeichen. Dann sucht es nach dem Zeichen char. Alle Zeichen, die es vorher findet, kopiert es hinter das Ende des Wörterbuchs. Die Zahl der Zeichen trägt es direkt hinter dem Wörterbuch ein, dort liegt dann ein "counted string". Zum Schluß legt es das Wörterbuchende (HERE) auf den Stack. In bigFORTH hängt es hinter den String noch ein Leerzeichen, damit FIND etwas schneller arbeiten kann.

CAPITALIZE (addr −− addr) wandelt einen solchen counted string in Großbuchstaben um. Es verändert die Stringadresse nicht, arbeitet also „destruktiv".

FIND schließlich durchsucht die Vokabulare in der Reihenfolge, in der sie auf dem Vocabulary Stack (VS) liegen. ORDER gibt den VS aus:

```
ORDER⟨RET⟩ FORTH FORTH ROOT   FORTH  ok
```

Das letzte Wort ist das Current Vokabulary, in dem definiert wird. Es wird in der Variable CURRENT gespeichert. Vokabulare kann man mit VOCABULARY ⟨***Name***⟩ definieren. ⟨***Name***⟩ selbst bewirkt dann, daß das Vokabular an oberster Stelle im VS eingetragen wird, das vorherige Vokabular wird dabei gelöscht. Mit ALSO kann man das oberste Vokabular (Context Vocabulary) verdoppeln, damit ihm solches nicht wiederfährt.

Während ALSO wie DUP wirkt, kann man mit TOSS das oberste Vokabular vom Stack nehmen, der Platz wird dann freigegeben. DEFINITIONS schließlich macht das oberste Vokabular zum Current Vocabulary.

Diese Vokabulare dienen zur Wahrung der Übersicht. Libraries wie GEM, GEMDOS oder die Floating–Point–Arithmetik haben ihre eigenen Vokabulare, Tools wie der Assembler, der Druckertreiber oder der Tracer ebenfalls. Es empfielt sich, auch Anwendungen in einem eigenen Vokabular zu definieren.

Warum? Man kann Vokabulare ausblenden, die Suche geht dann schneller. Man kann unterschiedliche Prioritäten setzen, welches Vokabular zuerst durchsucht wird, auch das steigert die Geschwindigkeit. Und man kann in verschiedenen Vokabularen Wörter mit gleichen Namen definieren und trotzdem auf alle zugreifen, normalerweise überdeckt das neuere Wort das ältere (es wird beim Compilieren die Warnung „⟨***Wort***⟩ exists" ausgegeben).

13. Strings

Wie speichert FORTH Strings? Strings sind Zeichenketten mit unterschiedlicher Länge. Grundsätzlich gibt es zwei Formate: Entweder kennzeichnet man das Stringende durch ein besonderes Byte, meistens ist das das 0–Byte („0–terminiert"). Oder man schreibt seine Länge an den Anfang, das ist dann ein „counted string". In FORTH wird von der zweiten Methode Gebrauch gemacht, die Länge steht im Countbyte. Strings zur freien Verwendung werden mit " ⟨***String***⟩" compiliert. Zur Laufzeit wird die Adresse des Strings auf den Stack gelegt.

COUNT (addr −− addr count) bringt es in eine z. B. für TYPE brauchbare Form. COUNT könnte man so definieren:

```
: count ( addr -- addr count ) dup c@ >r 1+ r> ;
```

Das Wort .“ ⟨***String***⟩” ersetzt die Sequenz “ ⟨***String***⟩” COUNT TYPE. Es gibt noch ein paar andere Wörter, die Strings brauchen. ABORT“ ⟨***String***⟩” (flag −−) gibt eine Fehlermeldung aus, wenn flag wahr ist, löscht den Stack und startet mit QUIT den Interpreter neu. ERROR“ ⟨***String***⟩” (flag −−) tut dasselbe, löscht aber den Stack nicht.

Andere Wörter sind für die GEM–Library wichtig. Mit 0“ ⟨***String***⟩” kann man übrigens auch 0–terminierte Strings definieren, wie sie von GEM oft gebraucht werden, das ist aber kein Standard–Befehl.

Strings kann man im Speicher auch verschieben. Das ist mit drei Wörtern möglich, die sich durch ihre Eigenschaften unterscheiden: CMOVE (sourceaddr destaddr count −−), CMOVE> (saddr daddr count −−) und MOVE (saddr daddr count −−). Für nicht überlappende Speicherbereiche funktionieren alle drei gleich. CMOVE bewegt den Speicherinhalt Byte für Byte vom ersten Byte an. CMOVE> fängt beim letzten Byte an. MOVE entscheidet sich für die Methode, die den String ganz läßt. In bigFORTH ist MOVE bei längeren Speicherbereichen auch wesentlich schneller als CMOVE und CMOVE>, da es anders definiert ist.

Das Problem von CMOVE ist leicht erklärt. Geben Sie ein:

```
" Dies ist ein Text" count 2dup over 4+ swap cmove⟨RET⟩  ok
swap 4+ swap type⟨RET⟩ DiesDiesDiesDiesD ok
```

Dieses Verhalten ist so definiert, man kann CMOVE also dazu benutzen, Speicherbereiche mit beliebigen Zeichenketten zu füllen. Da CMOVE> genau andersrum vorgeht, könnte man es auch benutzen.

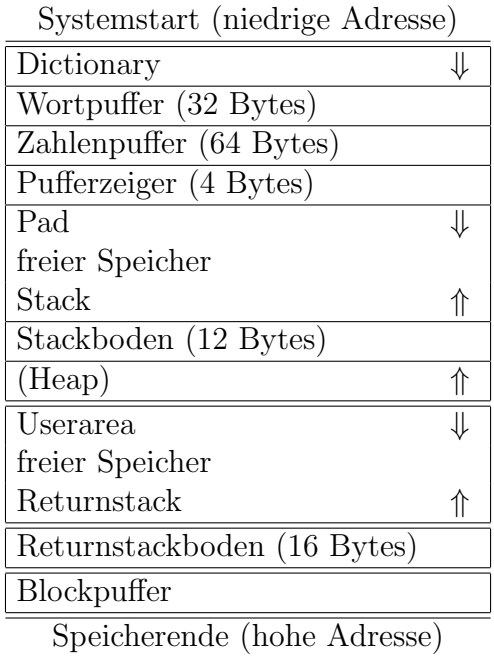
Zum Füllen mit einzelnen Zeichen kann man FILL (addr count char −−) benutzen, zum Löschen ERASE (addr count −−), das als `: ERASE 0 FILL ;` definiert ist.

14. Speicheraufbau und Multitasking

Was noch zum Verständnis von FORTH fehlt, ist der Aufbau des Systems im Speicher. Auch dieser ist standardisiert. Das mag ungewöhnlich erscheinen, kaum eine Sprache legt einen zwingenden Aufbau fest. Auch dies ist eine Betriebssystemeigenschaft von FORTH.

Stacks wachsen auf Prozessoren wie dem 68000 in Richtung niedriger Adressen. Programme dagegen werden in Richtung hörerer Adressen abgearbeitet. So ist das auch in FORTH. Damit ein gemeinsamer Speicher benutzt werden kann, wachsen Stack und Dictionary entgegen. Hinter dem Dictionary ist Platz für die Wörter, die von WORD dort abgelegt werden. Dann folgt der Puffer für Zahlenausgaben, hinter dem der Zeiger steht, der auf den Pufferanfang zeigt. Hinter diesem Puffer ist der Pad, ein Textpuffer (Seine Adresse liefert das Wort PAD).

Auf der anderen Seite liegt der Stack. Hinter dem Stack kommt die Userarea, die noch erklärt werden muß. Zwischen Stack und Userarea kann in bigFORTH ein Heap (Haufen) aufgebaut werden, auf dem unsichtbare Systemnamen abgelegt werden. Die Userarea kann erweitert werden und wächst in Richtung höhrere Adressen. Noch weiter „oben“ im Speicher ist der Returnstackboden. Dahinter sind die Blockpuffer. Nochmal grafisch zur Veranschaulichung („Feste“ Adressen, die sich nach dem Systemstart nicht mehr ändern, sind durch Doppelstriche gekennzeichnet, die Pfeile zeigen die Wachstumsrichtung):



Die Uservariablen sind im Gegensatz zu normalen Variablen nur innerhalb eines Tasks global. Hier werden also taskspezifische Werte gespeichert. In S0 und R0 z. B. sind Stackboden und Returnstackboden gespeichert, BASE enthält die Zahlenbasis.

Für jeden Task werden eigener Stack, Returnstack und Userarea angelegt. Auch das Ende des Dictionaries (HERE), das in DP gespeichert ist, hat bei jedem Task eine eigene Adresse. Damit sind der Textpuffer PAD und der Zahlenausgabepuffer der Tasks getrennt. Somit kommt es hier zu keinen Konflikten.

Der 68000 kann nur eine Task auf einmal bearbeiten. Um Multitasking zu bewirken, muß also umgeschaltet werden, es muß Stack, Returnstack und Userarea neu gesetzt werden. Dies wird durch das Wort PAUSE bewirkt. In alle längeren Berechnungen sollte man also PAUSE einfügen, damit die anderen Tasks nicht zu lange aufgehalten werden. Während man auf Benutzereingabe wartet, sollte man solange PAUSE aufrufen, bis eine Eingabe eingetroffen ist.

Mehr zur Verwaltung von Tasks finden Sie im Kapitel 7.6.

15. FORTH als Betriebssystem

Eine Reihe Eigenschaften von FORTH haben wir schon kennengelernt, die eigentlich nicht Sache einer Sprachdefinition, sondern eines Betriebssystems sind. Die Konzepte sind sehr einfach, aber wirkungsvoll gehalten. Dies hat zwei Gründe: Zum einen fällt die Implementierung leicht. Zum anderen sind gerade so einfache Konzepte die Grundvoraussetzung für Realtimefähigkeit (=Echtzeitfähigkeit).

Damit unterscheidet sich FORTH stark von den damals (als FORTH entstand) üblichen Betriebssystemen. Hier war die gängige Methode der Batch–Betrieb („Stapel–Verarbeitung“). Der Anwender gibt den Computer einen Auftrag, den dieser bei Gelegenheit ausführt. In den Anfangsjahren schlief man eine Nacht zwischen Auftrag und Ergebnis, später kochte man sich einen Kaffee, bis vor kurzem rauchte man eine Zigarette und inzwischen reicht die Zeit nur noch für ein nervöses Trommeln mit den Fingern (nervös vor allem, weil man sich das Rauchen abgewöhnen mußte).

Dennoch blieb die Strategie des Betriebssystems dieselbe: Es wird eine komplexe und umfangreiche Umgebung bereitgestellt, der Programmierer braucht sich nicht um die Einzelheiten und Interna kümmern. Die Antwortzeiten brauchen nicht festgelegt werden,

primäres Interesse gilt dem Gesamtdurchsatz. Hauptsache, die Aufgabe wird überhaupt erledigt.

Kommt es dagegen auf Antwortzeiten an, muß das System also auf externe Einflüsse in einer bestimmten Zeit reagieren, reicht diese Strategie nicht aus. Eine Robotersteuerung darf sich keine Zeit lassen, sonst ist das Werkstück auf dem Fließband schon vorbeigerauscht. Alle Aktionen, die das Programm durchführen muß, müssen in einer definierten Zeit abgeschlossen sein.

So ist das Massenspeicherkonzept von FORTH nicht nur eine sehr einfache Lösung, sondern erfüllt genau diese Bedingung: Lädt man einen neuen Block, so wird im ungünstigsten Fall ein anderer zurückgeschrieben und dann erneut auf den Massenspeicher zugegriffen — rechnet man die maximale Zugriffszeit plus die Übertragungszeit für 1 KByte zweimal, erhält man die oberste Grenze des Zeitfaktors für einen Zugriff. In einem System mit hierarchischer Dateiverwaltung kann nicht ausgeschlossen werden, daß auf eine Reihe von Sektoren zugegriffen werden muß, die zudem noch über die ganze Oberfläche des Mediums verstreut sind: Auf die FAT, eventuell mehrere Directories und auf die Datei selbst.

In FORTH hat der Programmierer die Kontrolle über sämtliche Resourcen. Er weiß auch, wie das Betriebssystem (also FORTH) auf eine Anforderung reagiert — auch zeitlich. Trotzdem ist FORTH nicht so hardwaregebunden, daß eine Portabilität nur sehr erschwert würde. Übrigens: Realtimefähigkeit ist nicht nur eine Sache für Hardwarebastler — ein interaktives System muß auf Eingaben des Benutzers genauso in einer gewissen Zeit reagieren.

16. Schlußbemerkung

Wir sind nun am Ende dieses bewußt knapp gehaltenen Kurses angekommen. Ein eigens für diesen Zweck geschriebenes FORTH–Buch kann sicher die Anforderungen besser erfüllen. Sollte der Kurs ausgereicht haben, Ihnen das Programmieren in FORTH beizubringen, lassen Sie sich eine Warnung geben: bigFORTH ist ein sehr komplexes System. Mit dem nur hier erlangten Wissen können Sie allenfalls an der Oberfläche kratzen. Allerdings können Sie bei der Arbeit mit bigFORTH Ihre Erkenntnisse enorm vertiefen.

Dieser Teil wurde geschrieben, um Einsteigern fürs Erste eine provisorische Unterstützung zu geben und um bereits Gelerntes aufzufrischen. Es wurde versucht, mit möglichst wenig Detailwissen den Systemcharakter von FORTH verständlich zu machen. Auf dieser Basis kann aufgebaut werden.

4 Ein Web-Server in Forth

1. Einleitung

Nachdem ich in den vergangenen Jahren immer Vorträge zu bigFORTH/MINOS gehalten habe, soll dieses Mal wieder Gforth drankommen. Auch mit Gforth kann man tolle Sachen machen und im Gegensatz zu manchen Unkenrufern geht eigentlich alles in Forth ziemlich einfach. Auch ein Web-Server.

Das Internet ist in Zeiten der "New Economy" wichtig. Jeder ist ,,drin". Außer Forth, das versteckt sich in der Embedded-Control-Nische. Ernsthaften Grund gibt's aber keinen. Der folgende Code ist in einigen Stunden Arbeit entstanden und verarbeitet hauptsächlich Strings. Das alte Vorurteil, Forth könne zwar gut Bits beißen, mache bei Strings aber Probleme, stimmt also nicht.

1.1. Motivation

Wozu braucht man einen Web-Server in Forth? Der eine hat Forth auf dem Meeresgrund oder im Krater eines Vulkans, und macht dort seine Messungen. Der andere hat Forth im Kühlschrank, und wenn der ausfällt, gibt's eine riesige Sauerei. Also hat man irgendeine Kommunikation eingebaut.

Ideal wäre es jetzt natürlich, wenn man nicht irgendeine Kommunikation eingebaut hat, sondern ein echtes Standard-Protokoll. HTTP kann auch der Browser im Web-Café auf Mallorca, oder mobiles Yuppie-Spielzeug wie PDAs und Handies. Vielleicht sollte man sowas in alle Herde und Badewannenmischbatterien einbauen, dann können die Leute im Urlaub über Handy jederzeit (alle drei Minuten) nachgucken, ob sie jetzt *wirklich* den Herd abgeschaltet haben.

Jedenfalls, der Kunde, Chef, oder wer auch immer das Produkt abnehmen soll, will sowieso, daß so ein Internet-Dingsbums da drin ist, wenn man schon nicht *e*-Bussiness macht. Und kosten darf das alles auch nichts.

Aber nun erst mal langsam, Schritt für Schritt.

2. Ein Web-Server, Schritt für Schritt

Eigentlich müßte man erstmal RFC[1]-Dokumente wälzen. Die in Frage kommenten RFCs sind RFC 1945 (HTTP/1.0) und RFC 2068 (HTTP/1.1), die verweisen natürlich ihrerseits auf andere RFCs. Da die Dokumente allein schon viel länger sind als der unten präsentierte Source-Code (und das Lesen würde länger dauern als das Schreiben dieser Sourcen), ersparen wir uns die für heute. Der Web-Server wird also nicht 100% RFC-konform sein (spricht: alle Features implementieren), sondern nur soweit, um mit typischen Clients wie Netscape zusammenzuarbeiten. Ergänzungen können leicht durchgeführt werden.

Ein typischer HTTP-Request sieht etwa so aus:

```
GET /index.html HTTP/1.1
Host: www.paysan.nom
Connection: close
```

[1] Request For Comments — die Dokumente von Internet-Standards heißen so.

(Man beachte die Leerzeile am Schluß). Und die Antwort

```
HTTP/1.1 200 OK
Date: Tue, 11 Apr 2000 22:27:42 GMT
Server: Apache/1.3.12 (Unix) (SuSE/Linux)
Connection: close
Content-Type: text/html

<HTML>
...
```

Das sieht irgendwie recht trivial aus. Also, frisch ans Werk. Der Web-Server soll unter Unix/Linux laufen. Damit haben wir ein Problem schon mal vom Hals: Das Problem, wie wir an unseren Socket kommen. Das macht inetd, der Internet Deamon. Dem brauchen wir nur zu sagen, auf welchem Port unser Web-Server Daten erwartet, etwa wie folgt in der Datei `/etc/inetd.conf`:

```
# Gforth web server

gforth stream tcp nowait.10000   wwwrun   /usr/users/bernd/bin/httpd
```

Da wir erstmal nicht den vorhandenen Web-Server ersetzen wollen (es könnte ja noch etwas nicht funktionieren), brauchen wir einen eigenen Port, der kommt in die Datei `/etc/services`:

```
gforth  4444/tcp # Gforth web server
```

Beim nächsten Neustart (oder einem `killall -HUP inetd`) kriegt der Inetd das mit und startet dann für alle Requests auf Port 4444 unseren eigenen Web-Server. Aber nun brauchen wir erst mal ein ausführbares Programm. Gforth unterstützt Scripting mit `#!`, wie unter Unix üblich. Man muß nur auf das Leerzeichen achten:

```
#! /usr/local/bin/gforth

warnings off
```

Die Warnungen schalten wir lieber aus. Nun noch eine kleine String-Bibliothek laden (siehe Anhang).

```
include string.fs
```

Wir brauchen einige Variablen, für die URL, die der Server so verlangt, für Argumente, gepostete Argumente, das Protokoll und für Zustände.

```
Variable url       \ speichert die URL (string)
Variable posted    \ speichert Argumente von POST (string)
Variable url-args  \ speichert Argumente in der URL (string)
Variable protocol  \ speichert das Protokoll (string)
Variable data      \ true, wenn Daten zurückgegeben werden
Variable active    \ true für POST
Variable command?  \ true in der Request-Zeile
```

Ein Request besteht aus zwei Teilen, der Request-Line und dem Header. Trenner sind dabei Spaces. Das erste Wort einer Zeile ist dabei jeweils ein "Token" des Protokolls, der Rest der Zeile, oder ein/zwei Wörter sind die Parameter.

Da wir einen Request erst abarbeiten können, wenn der ganze Header durchgelaufen ist, speichern wir alle Informationen zwischen. Dazu definieren wir uns zwei kleine Wörter, die ein Wort bzw. den Rest der Zeile in eine String-Variable speichern:

```
: get ( addr -- )  name rot $! ;
: get-rest ( addr -- )
  source >in @ /string dup >in +! rot $! ;
```

Wie gesagt, gibt's Header-Werte und Request-Kommandos. Damit wir die einfach interpretieren können, definieren wir zwei Wordlists:

```
wordlist constant values

wordlist constant commands
```

Doch bevor's richtig losgehen kann: Die URL könnte ja auch Spaces und andere Sonderzeichen enthalten, was macht man damit? HTTP schreibt vor, solche Sonderzeichen in der Form %xx zu übertragen, wobei xx für zwei Hex-Ziffern steht. Wir müssen also diese Zeichen in der fertigen URL ersetzen:

```
\ HTTP URL rework

: rework-% ( add -- ) { url }  base @ >r hex
    0 url $@len 0 ?DO
        url $@ drop I + c@ dup '% = IF
            drop 0. url $@ I 1+ /string
            2 min dup >r >number r> swap - >r 2drop
        ELSE  0 >r  THEN  over url $@ drop + c!  1+
    r> 1+ +LOOP  url $!len
    r> base ! ;
```

Uff, das ist geschafft. Doch halt! URLs bestehen aus zwei Teilen: Dem Pfad und optionalen Argumenten. Trenner ist das '?'. Also erst mal den String in zwei Teile zerlegen:

```
: rework-? ( addr -- )
    dup >r $@ '? $split url-args $! nip r> $!len ;
```

Na also, jetzt können wir ans Werk gehen. Requests besorgen sich also eine URL und ein Protokoll, zerlegen die URL in Pfad und Argumente und ersetzen die Sonderzeichen im Pfad durch darstellbare Zeichen (bei denen in den Argumenten warten wir vorerst noch mal ab, was damit überhaupt geschehen soll). Abschließend müssen wir auf ein anderes Vokabular umschalten, weil hinter dem Request ja der Header kommt.

```
: >values  values 1 set-order command? off ;
: get-url ( -- ) url get protocol get-rest
    url rework-? url rework-% >values ;
```

So, jetzt können wir unsere Kommandos definieren. Nach RFC brauchen wir nur GET und HEAD, POST gibt's also als Dreingabe

```
commands set-current

: GET   get-url data on  active off ;
: POST  get-url data on  active on  ;
: HEAD  get-url data off active off ;
```

Und nun zu den Header-Werten. Da wir für jeden Wert eine String-Variable brauchen und uns ansonsten einfach nur den String merken wollen, bauen wird das mit Create-DOES¿. Also nochmal: Wir brauchen eine Variable *und* ein Wort, das den Rest der Zeile dort abspeichert. In zwei verschiedenen Vokabularen. Das letztere mit einem Doppelpunkt hintendran.

Immerhin liefert uns Gforth mit `nextname` dafür ein passendes Werkzeug: Wir basteln uns einfach genau den String zusammen, den wir brauchen und rufen `Variable` und `Create` danach auf.

```
: value:  ( -- )  name
  definitions 2dup 1- nextname Variable
  values set-current nextname here cell - Create ,
  definitions  DOES> @ get-rest ;
```

Und jetzt frisch ans Werk und alle nötigen Variablen definiert:

```
value: User-Agent:
value: Pragma:
value: Host:
value: Accept:
value: Accept-Encoding:
value: Accept-Language:
value: Accept-Charset:
value: Via:
value: X-Forwarded-For:
value: Cache-Control:
value: Connection:
value: Referer:
value: Content-Type:
value: Content-Length:
```

Es gibt zwar noch ein paar mehr (siehe RFC), das sind aber die, die wir im Moment brauchen.

3. Parsen eines Requests

Also, nun müssen wir nur noch den Request parsen. Eigentlich wäre das total trivial, wir lassen einfach den Forth-Interpreter werkeln. Hat aber zwei kleine Haken:

1. Endet jede Zeile mit CR LF, während Gforth unter Unix davon ausgeht, daß Zeilen nur mit einem LF enden. Wir müßen also das CR wegnehmen. Und

2. endet der Header mit einer Leerzeile und nicht irgendeinem ausführbaren Forth-Wort. Wir müssen also Zeile für Zeile mit `refill` einlesen, CRs am Zeilenende wegnehmen,und dann noch gucken, ob die Zeile leer wahr.

```
Variable maxnum

: ?cr ( -- )
  #tib @ 1 >= IF  source 1- + c@ #cr = #tib +!  THEN ;
: refill-loop ( -- flag )
  BEGIN  refill ?cr  WHILE  interpret  >in @ 0=  UNTIL
  true  ELSE  maxnum off false  THEN ;
```

So, das wichtigste ist schon geschafft. Da wir den Forth-Interpreter nicht einfach auf stdin loslassen können, müssen wir das selber machen. Wir initialisieren also ein paar Variablen, die wir auf alle Fälle auswerten und klauen Code aus "included":

```
: get-input ( -- flag ior )
  s" /nosuchfile" url $!  s" HTTP/1.0" protocol $!
  s" close" connection $!
  infile-id push-file loadfile !  loadline off  blk off
  commands 1 set-order  command? on  ['] refill-loop catch
```

Mooooment! Der Request ist noch nicht ganz fertig. Die Methode POST, die's als Dreingabe gibt, erwartet jetzt noch ihre Daten. Die Länge ist freundlicherweise als Base-10-Zahl im Feld "`Content-Length:`" angegeben.

```
  active @ IF  s" " posted $! Content-Length $@ snumber? drop
      posted $!len  posted $@ infile-id read-file throw drop
  THEN  only forth also  pop-file ;
```

4. Auf einen Request antworten

Ok, der Request wäre damit abgehandelt, wir müssen also die Antwort geben. Der Pfad der URL ist leider nicht gerade so, wie wir ihn uns wünschen: Wir wollen irgendwie Apache-kompatibel sein, d.h. wir haben ein "global document root" und ein Verzeichnis im Home-Directory eines jeden Users, in dem der seine persönliche Homepage unterbringt. Also bleibt uns nichts anderes, als den Pfad erneut unter die Mangel zu nehmen und abschließend zu überprüfen, ob's die verlangte Datei überhaupt gibt:

```
Variable htmldir

: rework-htmldir ( addr u -- addr' u' / ior )
  htmldir $!
  htmldir $@ 1 min s" \textasciitilde{}" compare 0=
  IF    s" /.html-data" htmldir dup $@ 2dup '/ scan
        nip - nip $ins
  ELSE  s" /usr/local/httpd/htdocs/" htmldir 0 $ins  THEN
  htmldir $@ 1- 0 max + c@ '/ = htmldir $@len 0= or
  IF  s" index.html" htmldir dup $@len $ins  THEN
  htmldir $@ file-status nip ?dup ?EXIT
  htmldir $@ ;
```

Als nächstes müssen wir uns entscheiden, wie der Client die Datei darstellen soll, spricht, was für einen MIME-Typ sie hat. Entscheidungskriterium ist die Dateiendung. Die muß extrahiert werden.

```
: >mime ( addr u -- mime u' )  2dup tuck over + 1- ?DO
  I c@ '. = ?LEAVE  1-  -1 +LOOP  /string ;
```

Normalerweise werden wir die Datei so, wie sie ist, an den Client schicken (transparent). Dazu sagt man ihm am besten, wie lang die Datei ist (sonst müßte man die Connection für jeden Request schließen). Wir wandeln also einen Dateinamen in Größe und Filedeskriptor um und senden das dann an den Client.

```
: >file ( addr u -- size fd )
  r/o bin open-file throw >r
  r@ file-size throw drop
  ." Accept-Ranges: bytes" cr
  ." Content-Length: " dup 0 .r cr r> ;
: transparent ( size fd -- ) { fd }
  $4000 allocate throw swap dup 0 ?DO
      2dup over swap $4000 min fd read-file throw type
      $4000 - $4000 +LOOP  drop
  free fd close-file throw throw ;
```

Die Arbeit mit der Dateigröße machen wir uns für "Keep-Alive"-Connections, wie sie moderne Web-Browser gerne machen. Der Aufbau einer neuen Verbindung ist um einiges ,,teurer", als einfach mit der bestehenden weiterzumachen. Auch auf unserer Seite kommt uns das zugute, denn Gforth nochmal starten ist auch nicht umsonst. Wenn die Connection Keep-Alive ist, geben wir das also zurück, erniedrigen `maxnum` um eins und teilen dem Empfänger mit, wie oft er noch darf. Wenn's der letzte Request ist, oder gar kein weiterer verlangt wurde, senden wir das auch zurück.

```
: .connection ( -- )
  ." Connection: "
  connection $@ s" Keep-Alive" compare 0= maxnum @ 0> and
  IF  connection $@ type cr
      ." Keep-Alive: timeout=15, max=" maxnum @ 0 .r cr
      -1 maxnum +!  ELSE  ." close" cr maxnum off  THEN ;
```

Jetzt brauchen wir nur noch eine Möglichkeit, MIME-Dateienden in entsprechende Transmissions zu übersetzen. Auch bei der Rückantwort gilt: Zunächst einmal muß ein Header gesendet werden. Den bauen wir hier "von hinten her" auf, weil die oberen Definitionen ihren Senf weiter vorne dazugeben. Damit wir die Zuordnung Dateiende–MIME-Typ leicht haben, definieren wir für jede Datei einfach ein Wort. Das kriegt den MIME-Typ als String übergeben. `transparent:` macht uns das für all die Dateitypen, die transparent durchgereicht werden:

```
: transparent: ( addr u -- ) Create  here over 1+ allot place
  DOES>  >r  >file
  .connection
  ." Content-Type: "  r> count type cr cr
  data @ IF  transparent  ELSE  nip close-file throw  THEN ;
```

So, nun gibt's ja hunderte MIME-Typen, wer will die alle eintippen? Nichts leichter als das, klauen wir einfach die MIME-Typen, die dem System schon bekannt sind, etwa aus `/etc/mime.types`. Die Datei listet links jeweils einen MIME-Typ, rechts einige Dateiendungen (auch mal keine).

```
: mime-read ( addr u -- )  r/o open-file throw
    push-file loadfile !  0 loadline ! blk off
    BEGIN  refill  WHILE  name
        BEGIN  >in @ >r name nip  WHILE
            r> >in ! 2dup transparent:  REPEAT
        2drop rdrop
    REPEAT  loadfile @ close-file pop-file throw ;
```

Einen brauchen wir noch: Für aktive Inhalte wollen wir Serverseitiges Skripting (natürlich in Forth) verwenden. Bei solchen Inhalten kennen wir die Länge nicht im Vorraus, also geben wir sie einfach nicht an, sonder machen die Verbindung zu. Damit entledigen wir uns auch des Problems, den Müll, den der User mit seinen aktiven Inhalten anrichtet (das ist schließlich Forth-Code!) wieder aufräumen zu müssen.

```
: lastrequest
  ." Connection: close" cr maxnum off
  ." Content-Type: text/html" cr cr ;
```

Also, los geht's mit der Definition der MIME-Typen. Her mit einer neuen Wordlist. Aktive Inhalte enden mit `shtml` und werden per `included` geladen. Ansonsten ein paar Beispieldefinitionen und den Rest kriegen wir aus der oben genannten Systemdatei. Für unbekannte Dateitypen brauchen wir noch einen Standardtyp, `text/plain`.

```
wordlist constant mime
mime set-current

: shtml ( addr u -- )  lastrequest
    data @ IF  included  ELSE  2drop  THEN ;

s" application/pgp-signature" transparent: sig
s" application/x-bzip2" transparent: bz2
s" application/x-gzip" transparent: gz
s" /etc/mime.types" mime-read

definitions

s" text/plain" transparent: txt
```

5. Fehlermeldungen

Manchmal geht bei so einem Request auch was schief. Jedenfalls müssen wir darauf gefaßt sein und dem Client eine entsprechende Meldung machen. Er will wissen, welches Protokoll wir sprechen, was passiert ist (oder ob alles OK ist), wer wir sind und im Fehlerfall ist eine Klartextmeldung (in HTML codiert) ganz nett:

```
: .server ( -- )  ." Server: Gforth httpd/0.1 ("
    s" os-class" environment? IF  type  THEN  ." )" cr ;
: .ok  ( -- ) ." HTTP/1.1 200 OK" cr .server ;
: html-error ( n addr u -- )
    ." HTTP/1.1 " 2 pick . 2dup type cr .server
    2 pick &405 = IF ." Allow: GET, HEAD, POST" cr  THEN
    lastrequest
    ." <HTML><HEAD><TITLE>" 2 pick . 2dup type
    ." </TITLE></HEAD>" cr
    ." <BODY><H1>" type drop ." </H1>" cr ;
: .trailer ( -- )
    ." <HR><ADDRESS>Gforth httpd 0.1</ADDRESS>" cr
    ." </BODY></HTML>" cr ;
: .nok ( -- ) command? @ IF  &405 s" Method Not Allowed"
    ELSE  &400 s" Bad Request"  THEN  html-error
    ." <P>Your browser sent a request that this server "
    ." could not understand.</P>" cr
    ." <P>Invalid request in: <CODE>"
    error-stack cell+ 2@ swap type
    ." </CODE></P>" cr .trailer ;
: .nofile ( -- ) &404 s" Not Found" html-error
    ." <P>The requested URL <CODE>" url $@ type
    ." </CODE> was not found on this server</P>" cr .trailer ;
```

6. Top-Level-Definitionen

So, jetzt sind wir eigentlich fertig. Wir müssen die ganzen Stückchen noch zusammenkleben, damit ein Request richtig abgearbeitet wird: Den Input holen, die URL umformen, den MIME-Typ erkennen und das dann abarbeiten, jeweils mit Fehlerausgängen oder Default-Pfaden. Die Ausgabe müssen wir flushen, damit der nächste Request nicht hängenbleibt. Und das ganze evtl. n mal abarbeiten, bis wir den letzten Request erreicht haben.

```
: http ( -- )  get-input  IF  .nok  ELSE
    IF  url $@ 1 /string rework-htmldir
        dup 0< IF  drop .nofile
        ELSE  .ok  2dup >mime mime search-wordlist
            0= IF  ['] txt  THEN  catch IF  maxnum off THEN
        THEN  THEN  THEN  outfile-id flush-file throw ;

: httpd  ( n -- )  maxnum !
  BEGIN  ['] http catch  maxnum @ 0= or  UNTIL ;
```

Damit das dann beim Start von Gforth auch ausgeführt wird, patchen wir die Bootmessage damit. So können wir das auch als neues Systemimage abspeichern.

```
script? [IF]  :noname &100 httpd bye ; is bootmessage  [THEN]
```

7. Scripting

Als besonderes Bombon gibt's nun noch die aktiven Inhalte. Das ist wirklich ganz einfach: Wir schreiben unsere HTML-Datei so wie gewohnt, nur den Forth-Code in "¡$" und "$¿ " eingerahmt (das Leerzeichen für die schließende Klammer ist natürlich Absicht!). Definieren müssen wir nur zwei Wörter, `$>` und damit das Ganze überhaupt losgeht, `<HTML>`:

```
: $> ( -- )
    BEGIN  source >in @ /string s" <$" search  0= WHILE
        type cr refill  0= UNTIL  EXIT  THEN
    nip source >in @ /string rot - dup 2 + >in +! type ;
: <HTML> ( -- )  ." <HTML>" $> ;
```

Das reicht in der Tat, mehr brauchen wir nicht, den Rest erledigt Forth, wie in folgendem Beispiel:

```
<HTML>
<HEAD>
<TITLE>GForth <$ version-string type $>  presents</TITLE>
</HEAD>
<BODY>
<H1>Computing Primes</H1><$ 25 Constant #prim $>
<P>The first <$ #prim . $> primes are: <$
: prim?  0 over 2 max 2 ?DO  over I mod 0= or  LOOP  nip 0= ;
: prims ( n -- )  0 swap  2
  swap 0 DO  dup prim? IF  swap  IF  ." , "  THEN  true swap
  dup 0 .r 1+  1  ELSE  1+  0  THEN
  +LOOP  drop ;
#prim prims $> .</P>
</BODY>
</HTML>
```

8. Ausblick

Das waren jetzt ein paarhundert Zeilen Code. Eigentlich viel zu viel. Ich habe auch einen fast vollwertigen Apache-Klon geliefert. Das wird man auf dem Meeresgrund oder im Kühlschrank nicht brauchen. Fehlerbehandlung ist auch nur Ballast. Und wenn man sich auf Einmalverbindungen beschränkt (Performance ist ja nicht das Ziel), kann man auch die ganzen Protokoll-Variablen einfach ignorieren. Ein MIME-Typ (text/html) reicht aus — die Bilder speichert man auf einem anderen Server. Es besteht jedenfalls die Hoffnung, ein einigermaßen ordentliches HTTP-Protokoll mit server-side Scripting in einem Screen unterzubringen.

9. Anhang: String-Funktionen

Natürlich braucht man irgendwelche String-Funktionen, sonst geht's nicht. Die folgenden String-Bibliothek speichert Strings in normalen Variablen, die dann einen Pointer auf einen counted String enthalten. Statt einem Count-Byte gibt's gleich eine ganze Zelle, das sollte ausreichen. Die String-Bibliothek stammt ursprünglich aus bigFORTH, ich habe sie für Gforth (ANS Forth) angepaßt. Doch nun die Funktionen im Detail. Zunächst brauchen wir einmal zwei Wörter, die bigFORTH schon so mitbringt:

```
: delete   ( addr u n -- )
  over min >r  r@ - ( left over )  dup 0>
  IF  2dup swap dup  r@ +  -rot swap move  THEN  + r> bl fill ;
```

`delete` löscht die ersten n bytes aus einen Puffer und füllt den Rest hinten mit Blanks auf.

```
: insert   ( string length buffer size -- )
  rot over min >r  r@ - ( left over )
  over dup r@ +  rot move   r> move  ;
```

`insert` fügt einen String vorne in einen Buffer ein. Die restlichen Bytes werden nach hinten geschoben.

So, nun geht's richtig zur Sache:

```
: $padding ( n -- n' )
  [ 6 cells ] Literal + [ -4 cells ] Literal and ;
```

Damit wir die Speicherverwaltung nicht überfordern, gibt's nur bestimmte String-Größen; `$padding` sorgt für Vielfache von vier Zellen.

```
: $! ( addr1 u addr2 -- )
  dup @ IF  dup @ free throw  THEN
  over $padding allocate throw over ! @
  over >r  rot over cell+  r> move 2dup ! + cell+ bl swap c! ;
```

`$!` speichert einen String an einer Adresse ab. Falls da vorher schon ein String drin war, wird der freigegeben.

```
: $@ ( addr1 -- addr2 u )  @ dup cell+ swap @ ;
```

`$@` gibt den gespeicherten String zurück.

```
: $@len ( addr -- u )  @ @ ;
```

`$@len` gibt die Länge eines Strings zurück

```
: $!len ( u addr -- )
  over $padding over @ swap resize throw over ! @ ! ;
```

`$!len` ändert die Länge eines Strings. Dazu muß sowohl der Speicherbereich vergrößert werden, als auch die Adresse und die Count-Cell geändert werden.

```
: $del ( addr off u -- )   >r >r dup $@ r> /string r@ delete
  dup $@len r> - swap $!len ;
```

`$del` löscht u bytes aus einem String mit Offset off.

```
: $ins ( addr1 u addr2 off -- ) >r

  2dup dup $@len rot + swap $!len  $@ 1+ r> /string insert ;
```

`$ins` fügt einen String am Offset off ein.

```
: $+! ( addr1 u addr2 -- ) dup $@len $ins ;
```

`$+!` hängt einen String an einen anderen an.

```
: $off ( addr -- )  dup @ free throw off ;
```

`$off` gibt einen String wieder frei.
Als Bonus gibt's noch Funktionen, um Strings zu zerlegen.

```
: $split ( addr u char -- addr1 u1 addr2 u2 )
  >r 2dup r> scan dup >r dup IF  1 /string  THEN
  2swap r> - 2swap ;
```

`$split` teilt einen String in zwei, als Trenner dient ein Zeichen (z.B. '? für Argumente)

```
: $iter ( .. $addr char xt -- .. ) { char xt }
  $@ BEGIN  dup  WHILE  char $split >r >r xt execute r> r>
     REPEAT  2drop ;
```

`$iter` zerlegt einen String Stück für Stück in seine Bestandteile, ebenfalls mit einem Zeichen als Trenner. Für jedes Teilstück wird ein übergebens Token aufgerufen. Damit kann man Argumente, die mit '& separiert sind, bequem trennen.

5 Referenzen - Kernel

In diesem und den nächsten Kapiteln werden alle Wörter erklärt, die in BIGFORTH.PRG definiert sind, sowie die Libraries, die noch dazugeladen werden können. Um die Übersicht zu wahren, sind die Wörter thematisch geordnet.

Leider muß doch immer wieder auf Systeminterna eingegangen werden. Das Verständnis dieser Informationen wird für den Einsteiger erst im Laufe der Zeit notwendig und dann durch die Erfahrung sehr erleichtert. bigFORTH ist eben ein sehr komplexes System, das nicht von den Einzelheiten her allein begriffen werden kann.

Eine alte Erfahrung sagt zudem, daß die beste Dokumentation der Sourcecode ist. Auch wenn er logisch auf einem noch niedrigeren Level liegt: Hier ist exakt beschrieben, was das Programm tut. Deshalb empfielt es sich, auch den Sourcecode zu studieren. Den Kernel-Source findet man in der Datei FORTH.SCR.

1. Der Kernel

FORTH ist der klassische Fall eines Self-Bootstraps: Es ist zum größten Teil in sich selbst definiert. Auch der Assembler ist ein FORTH-Programm. Nun bringt es natürlich ein reales FORTH nicht fertig, sich wie Münchhausen am eigenen Schopf aus dem Sumpf herauszuziehen („to lift oneself on his own bootstraps“ heißt „sich an den eigenen Schnürsenkeln herausziehen“). Eine gewisse „kritische Masse“ muß das System schon haben, so müssen Compiler und Interpreter laufen, der Massenspeicherzugriff funktionieren und genügend Wörter vorhanden sein, um alle weiteren zu definieren.

Diesen „Kern“ nennt man Kernel. Er wird vom Target-Compiler erzeugt. Dieser Targetcompiler ist natürlich auch ein FORTH-Programm und läuft, bis ein lauffähiges Kernel existiert, auf einem anderen FORTH-System und möglicherweise auf einem anderen Computertyp ab. Der Target-Compiler für bigFORTH ist in bigFORTH selbst lauffähig, er ist in der Datei TARGET.SCR definiert. Nach Änderungen im Kernel braucht man nur die Kerneldatei FORTH.SCR mit INCLUDE FORTH.SCR zu laden und das Ergebnis mit SAVE-TARGET FORTHKER.PRG sichern. So ist auch FORTHKER.PRG auf der grauen Diskette entstanden.

Soweit nicht anders angegeben, sind die Kernelwörter im Vokabular FORTH. Die Schlüsselwörter selbst sind fett gedruckt. Zu jedem Wort werden der Stackeffekt und die Compilerflags (immediate und restrict), wenn vorhanden, sowie eine knappe Beschreibung der Funktion angegeben. Symbolisch ausgedrückt:

Befehl::= ⟨*Name*⟩ (⟨*In*⟩ −− ⟨*Out*⟩) (⟨*Stackname*⟩ ⟨*In*⟩ −− ⟨*Out*⟩) ⟨*Inputstring*⟩[⟨*Begrenzer*⟩][immediate][restrict][:⟨*Befehl*⟩]

Stackname::=RS|VS|FS|$S

In::= ⟨*Parameter*⟩ / ⟨*Parameter*⟩

Out::= ⟨*Parameter*⟩ / ⟨*Parameter*⟩

NOOP (−−): No Operation. Dieses Wort tut nichts. Es verhindert aber eine Optimierung zwischen dem Macro vor und nach NOOP.

2. Stackbefehle

Der Stack dient der Parameterübergabe. Auf dem Returnstack liegen die Rücksprungadressen, hier können innerhalb eines Wortes eingeschränkt Werte abgelegt werden, die alle wieder heruntergenommen werden müssen, ehe das Wort verlassen wird. Mit Returnstackmanipulationen ist es außerdem möglich, den Programmablauf zu verändern, z. B. eine Ebene zu überspringen.

SP@ (−− addr): Legt den Stackpointer auf den Stack.
SP! (addr −−): Setzt **addr** als neuen Stackpointer.
RP@ (−− addr): Gibt den Returnstackpointer zurück.
RP! (addr −−): Setzt **addr** als neuen Returnstackpointer.
>R (n −−) (RS −− n) restrict: Schiebt den Top of Stack (TOS) auf den Returnstack.
R@ (−− n) (RS n −− n) restrict: Kopiert den obersten Wert des Returnstacks auf den Stack.
R> (−− n) (RS n −−) restrict: Schiebt den obersten Wert des Returnstacks zurück auf den Stack.
DUP (n −− n n): Verdoppelt den TOS.
?DUP (n / 0 −− n n / 0): Verdoppelt den TOS, wenn er nicht Null ist. Eine Null wird nicht verdoppelt.
DROP (n −−): Nimmt den TOS vom Stack.
NIP (n1 n2 −− n2): Nimmt den Wert unter dem TOS (Next of Stack, NOS) weg.
RDROP (−−) (RS n −−) restrict: Nimmt den obersten Wert des Returnstacks weg.
SWAP (n1 n2 −− n2 n1): Vertauscht TOS und NOS.
OVER (n1 n2 −− n1 n2 n1): Kopiert den NOS über den TOS.
UNDER (n1 n2 −− n2 n1 n2): Kopiert den TOS unter den NOS.
ROT (n1 n2 n3 −− n2 n3 n1): Rotiert den drittobersten Wert des Stacks nach oben.
−ROT (n1 n2 n3 −− n3 n1 n2): Rotiert den TOS an die drittoberste Stelle im Stack nach unten. ROT und -ROT sind Gegenspieler; jeweils zwei ROT ersetzen ein -ROT und umgekehrt.
PICK (n0 .. nx x −− n0 .. nx n0): Kopiert den x-ten Wert des Stacks nach oben. Die Zählung beginnt beim TOS, der die Nummer 0 hat.
ROLL (n0 n1 .. nx x −− n1 .. nx n0): Rollt den x-ten Wert des Stacks nach oben. Zählung wie bei PICK.
−ROLL (n0 .. nx-1 nx x −− nx n0 .. nx-1): Rollt den TOS an die x-te Position im Stack, Zählung wie bei PICK.

Ein Wertepaar kann in FORTH auch als doppelt genaue Zahl interpretiert werden. Der höherwertige Teil liegt dabei weiter oben auf dem Stack oder an der niedrigeren Speicheradresse. FORTH hat damit dasselbe Speichermodell wie der 68000. Für Wertepaare gibt es einige besondere Stackbefehle:

2SWAP (d1 d2 −− d2 d1): Vertauscht die obersten beiden Werte**paare** auf dem Stack.
2DUP (d −− d d): Verdoppelt das oberste Werte**paar** auf dem Stack, wirkt wie OVER OVER.
2OVER (d1 d2 −− d1 d2 d1): Kopiert das zweitoberste Werte**paar** über das oberste, wirkt wie OVER, aber auf Wertepaare.
2DROP (d −−): Löscht das oberste Wertepaar, wirkt wie DROP DROP.

DEPTH (−− depth): Berechnet die Stacktiefe. Es lagen *depth* Werte auf dem Stack (jetzt liegt mit *depth* einer mehr auf dem Stack).

RDEPTH (−− rdepth): Berechnet die Returnstacktiefe. Es liegen insgesamt *rdepth* Werte auf dem Returnstack.

3. Integer-Arithmetik

Standardmäßig verarbeitet FORTH Integerzahlen. Der Wertebereich richtet sich nach der Größe der Stackelemente. In bigFORTH umfaßt ein Stackelement 32 Bit, es gibt also Zahlen von -2^{31} bis 2^{31} (Bereich: $[-2\,147\,483\,648; 2\,147\,483\,647]$) bzw. 0 bis 2^{32} (Bereich: $[0; 4\,294\,967\,295]$) in vorzeichenloser Darstellung.

Ein Zahlenpaar kann auch als doppelt genaue Zahl (64-Bit-Zahl) aufgefaßt werden. Dabei ist der höherwertige Teil der weiter oben auf dem Stack liegende bzw. an der niedrigeren Adresse gespeicherte (Bereich ca. $[-9.2E18; 9.2E18]$ bzw. $[0; 18.4E18]$).

Die Arithmetik-Wörter nehmen ihre Eingangswerte vom Stack und legen das (die) Ergebnis(se) zurück. Die daraus resultierende Notation heißt Postfix- oder Zeitfolgennotation bzw. Umgekehrt Polnische Notation (UPN) (s. Kapitel 3).

+ (n1 n2 −− n1+n2): Addiert den TOS zum NOS.

− (n1 n2 −− n1-n2): Subtrahiert den TOS vom NOS.

OR (n1 n2 −− n): Führt eine binäre Oder-Verknüpfung von n_1 und n_2 durch. Dabei gilt folgende Wahrheitstabelle:

	0	1
0	0	1
1	1	1

(In der Tabelle stehen oben und links die Eingangsbits, in den Zellen in der Mitte die Ergebnisse. Diese Bitverknüpfung wird für jedes der 32 Bits durchgeführt.)

AND (n1 n2 −− n): Führt eine binäre Und-Verknüpfung von n_1 und n_2 durch:

	0	1
0	0	0
1	0	1

XOR (n1 n2 −− n): Führt eine binäre Exklusiv-Oder-Verknüpfung von n_1 und n_2 durch:

	0	1
0	0	1
1	1	0

NOT (n1 −− n2): Führt ein binäres Not (Einerkomplement) durch. Ein 1-Bit wird ein 0-Bit und umgekehrt.

NEGATE (n −− -n): Gibt das Zweierkomplement von n zurück. Das Zweierkomplement einer Zahl ist das um eins erhöhte Einerkomplement der Zahl. Addiert man das Zweierkomplement einer Zahl zur ursprünglichen Zahl, so kommt 0 heraus. Deshalb wird der Befehl NEGATE genannt.

DNEGATE (d −− -d): Führt das Zweierkomplement für eine doppelt genaue Zahl durch.

D+ (d1 d2 −− d1+d2): Addiert zwei doppelt genaue Zahlen.

D− (d1 d2 −− d1-d2): Subtrahiert zwei doppelt genaue Zahlen.

ABS (n −− u): Bildet den absoluten Betrag von n.

DABS (d −− ud): Bildet den absoluten Betrag von d.

EXTEND (n −− d): Erweitert n vorzeichenbehaftet zu der doppelt genauen Zahl d. Beim Erweitern werden die zusätzlichen höherwertigen Bits mit dem höchsten Bit von n aufgefüllt, also mit 1, wenn n negativ ist, sonst mit 0.

WEXTEND (16b −− n): Erweitert die 16-Bit-Zahl $16b$ vorzeichenrichtig auf eine 32-Bit-Zahl.

UM∗ (u1 u2 −− ud): Multipliziert ohne Berücksichtigung des Vorzeichens u_1 und u_2. Das Ergebnis ist doppelt genau (64 Bit). Dieses Wort ist das Basiswort für die Multiplikation, alle anderen bauen darauf auf.

M∗ (n1 n2 −− d): Multipliziert mit Berücksichtigung des Vorzeichens n_1 und n_2. Das Ergebnis ist auch hier doppelt genau.

∗ (n1 n2 −− n): Multipliziert unter Berücksichtigung des Vorzeichens, löscht aber den höherwertigen Teil, damit das Ergebnis auch eine einfach genaue Zahl ist. Ein Überlauf wird nicht abgefangen.

D∗ (d1 d2 −− d): Multipliziert zwei doppelt genaue Zahlen. Das Ergebnis ist auch doppelt genau, ein Überlauf wird nicht abgefangen.

Q∗ (16b1 16b2 −− 32b): Multipliziert mit dem eingebauten Multiplikationsbefehl des 68000 (muls Dn,Dn) zwei 16-Bit-Zahlen. Das Ergebnis ist 32 Bit lang. Q∗ wird als Makro compiliert.

UM/MOD (ud u −− urem uquot): Teilt die doppelt genaue vorzeichenlose Zahl ud ohne Berücksichtigung des Vorzeichens durch u. Dabei werden Modulowert ($urem$) und Quotient in dieser Reihenfolge zurückgegeben. Für Quotient und Modulowert gelten folgende Beziehungen: $ud = uquot * u + urem$ und $urem < u$. Der Quotient ist also ganzzahlig abgerundet. UM/MOD ist das Basiswort für Divisionen.

Mögliche Fehlermeldungen:

Division by Zero ! Eine Division durch Null kann nicht ausgeführt werden.

Division Overflow! Der Quotient wäre größer als $2^{32}-1$ und hat daher in einer 32-Bit-Zahl keinen Platz. In diesem Fall kann man eventuell auf UD/MOD ausweichen.

M/MOD (d n −− rem quot): Teilt die doppelt genaue Zahl d durch n und legt Modulowert und den Quotient in dieser Reihenfolge auf den Stack. Mögliche Fehlermeldungen wie bei UM/MOD.

/MOD (n1 n2 −− rem quot): Teilt n_1 durch n_2 und gibt Modulowert und Quotient zurück. Fehlermeldungen wie UM/MOD.

/ (n1 n2 −− quot): Wie /MOD NIP, gibt also nur den Quotient von n_1/n_2 zurück.

MOD (n1 n2 −− rem): Wie /MOD DROP, gibt nur den Modulowert zurück.

U/MOD (u1 u2 −− urem uquot): Dividiert die vorzeichenlosen Zahlen u_1 und u_2 und legt Modulowert und Quotient (ebenfalls vorzeichenlos) auf den Stack.

UD/MOD (ud u −− urem udquot): Dividiert die vorzeichenlose, doppelt genaue Zahl ud durch u, gibt Modulowert einfach genau und Quotient als doppelt genaue, vorzeichenlose Zahl zurück.

Q/MOD (32b 16b −− 16brem 16bquot): Schnelle Variante von /MOD, die den Divisionsbefehl (divs Dn,Dn) des 68000 benutzt. Wird als Makro compiliert. Der Divisor darf dabei nur 16 signifikante Bit haben, weitere werden nicht berücksichtigt. Es gibt auch einen Überlauf, wenn der Quotient nicht mit 16 Bit inclusive Vorzeichen dargestellt werden kann. Division durch 0 gibt die Fehlermeldung „Division by Zero !“.

Abweichendes Verhalten von /MOD: Bei einem negativen Quotient ist auch der Rest negativ. Da die Beziehung $Dividend = Divisor * Quotient + Rest$ weiterhin gilt, ist ein negativer Quotient betragsmäßig um eins kleiner als bei /MOD.

Q/ (32b 16b −− 16bquot): Wie Q/MOD NIP. Bezüglich des Quotienten gilt dasselbe wie oben, Q/ ist ebenfalls ein Makro.

QMOD (32b 16b −− 16brem): Wie Q/MOD DROP.

QUD/MOD (ud 16b −− udquot 16brem): Teilt die vorzeichenlose doppelt genaue Zahl *ud* mit dem eingebauten Divisionsbefehl des 68000 (divu Dn,Dn) durch einen 16-Bit-Quotient.

Achtung! Der Quotient wird hier im Gegensatz zu UD/MOD zuerst zurückgegeben, der Rest liegt oben auf dem Stack!

∗/MOD (n1 n2 n3 −− rem quot): Multipliziert $n1$ und n_2 mit M∗ und teilt das Ergebnis mit M/MOD durch n_3. Dieses Wort wird als Basis für das Rechnen mit skalierten Zahlen benutzt.

∗/ (n1 n2 n3 −− quot): Wie ∗/MOD NIP, löscht den bei skalierten Zahlen selten benötigten Rest und gibt nur $n_1 * n_2/n_3$ zurück.

Q∗/ (16b1 16b2 16b3 −− 16b): Wie ∗/, da aber muls Dn,Dn und divs Dn,Dn verwendet werden, werden nur die unteren 16 Bit der Zahlen berücksichtigt. Bei negativem Ergebnis ist genauso wie bei Q/ zu beachten, daß es betragsmäßig um eins kleiner ist als das Ergebnis von ∗/. Man sollte es daher besser nur für positive Zahlen verwenden. Q∗/ wird als Makro compiliert.

Einige Zahlen sind als Makros vordefiniert. Sie ergeben keinen besseren Code als andere vergleichbare Zahlen. Da aber diese Konstanten eine CFA besitzen, kann man sie wie normale Wörter behandeln. Außerden werden die Symbole TRUE und FALSE durch solche Makros vordefiniert. Ein Kommentar erschien überflüssig, da der Stackeffekt die Wirkung genau ausdrückt.

0 (−− 0):
1 (−− 1):
2 (−− 2):
3 (−− 3):
4 (−− 4):
−1 (−− -1):
TRUE (−− -1):
FALSE (−− 0):

Kommen wir nun zu einigen Abkürzungen, die schneller ausgeführt werden und einen kompakteren Code haben als ihre ausgeschriebenen Varianten:

1+ (n −− n+1): Wie 1 +
2+ (n −− n+2): Wie 2 +
3+ (n −− n+3): Wie 3 +
4+ (n −− n+4): Wie 4 +
6+ (n −− n+6): Wie 6 +
8+ (n −− n+8): Wie 8 +
1− (n −− n-1): Wie 1 -
2− (n −− n-2): Wie 2 -
4− (n −− n-4): Wie 4 -
2∗ (n −− n∗2): Wie 2 ∗ (Bitshift, also viel schneller)
2/ (n −− n/2): Wie 2 /, ebenfalls ein Bitshift
4∗ (n −− n∗4): Wie 4 ∗, Bitshift links um zwei Bitstellen
4/ (n −− n/4): Wie 4 /, Bitshift rechts um zwei Bitstellen

Ein FORTH-System hat ein einheitliches Speichermodell. Eine Speicherzelle („cell“) hat eine einheitliche Größe, in einem 16-Bit-FORTH sind es 16 Bit, oder 2 Bytes, in einem 32-Bit-FORTH entsprechend 32 Bit, also 4 Bytes. Stackelemente haben diese Größe, auch mit , compilierte Zahlen; @ und ! holen und speichern ebenfalls immer eine ganze Zelle ab.

Da der Speicher aber mit Byte-Adressen angesprochen wird, muß man die Größe einer Zelle kennen, um Zeigerberechnungen durchzuführen. Setzt man die Zellengröße (2 oder 4 Bytes) direkt als Zahl ein, so erhält man unterschiedlichen Sourcecode in 16- und 32-Bit-Systemen. Solche Unterschiede sind der Kompatibilität kaum dienlich, deshalb hat man hier eine Abhilfe gefunden: Die Länge einer Zelle steht in der Konstante CELL. Des weiteren können oft benötigte Kürzel zur Zeigerberechnung wie CELL+, CELL-, CELL∗ und CELL/ zur Verfügung gestellt werden. Auch -CELL, also die negierte Zellengröße, kann hin und wieder benötigt werden.

Doch leider, wie bei so vielen guten Ideen, kam sie viel zu spät. Diese Wörter (oder ein Teil davon) sollen erst in die ANSI-Norm übernommen werden. Die aber ist noch nicht fertig. So werden Sie bei existierenden 16-Bit-Sources doch die Zeigerberechnungen anpassen müssen. Verwenden Sie dann (und in eigenen Programmen) aber die folgenden Befehle, um eine Übertragung zumindest zu erleichtern, schließlich müssen dann nur noch diese sechs Wörter neu definiert werden, damit alle Adreßberechnungen stimmen.

CELL (−− 4): Gibt die Länge einer Speicherzelle in Bytes zurück.
−CELL (−− -4): Gibt die negierte Länge einer Speicherzelle zurück.
CELL+ (n −− n+4): Addiert zu n die Länge einer Speicherzelle. n als Adresse zeigt dann auf die folgende Zelle.
CELL− (n −− n-4): Subtrahiert von n die Länge einer Zelle. n als Adresse zeigt dann auf die vorhergehende Zelle.
CELLS (n −− n∗4): Multipliziert n mit der Länge einer Zelle. Damit kann man aus einem Index n den Adreßoffset in einem Array berechnen.
CELL/ (n −− n/4): Dividiert n durch die Länge einer Zelle. Damit kann man aus dem Adreßoffset n einen Index (das n-te Speicherelement) berechnen.

4. Zahlenvergleiche

Wie die Arithmetik werden Vergleiche in UPN notiert. Bei Vergleichen wird jedoch eine Flag „berechnet“. Sie liegt als Ergebnis auf dem Stack. 0 bedeutet dabei „false“, also „falsch“, −1 bedeutet „true“, d. h. „wahr“.

> (n1 n2 −− n1>n2): Gibt true zurück, wenn n_1 größer als n_2 ist.
< (n1 n2 −− n1<n2): Gibt true zurück, wenn n_1 kleiner als n_2 ist.
U> (u1 u2 −− u1>u2): Gibt true zurück, wenn u_1 größer als u_2 ist. Dabei wird das Vorzeichen nicht berücksichtigt, „negative“ Zahlen sind also größer als alle positiven Zahlen. Bei gleichem Vorzeichen gibt es dieselben Ergebnisse wie bei >.
U< (u1 u2 −− u1¡u2): Gibt true zurück, wenn u_1 vorzeichenlos kleiner als u_2 ist.
= (n1 n2 −− n1=n2): Gibt true zurück, wenn n_1 und n_2 gleich sind.
CASE? (n1 n2 −− t / n1 f): Gibt $t$rue zurück, wenn n_1 und n_2 gleich sind, andernfalls n_1 und $f$alse. CASE? wird für Mehrfachverzweigungen vergleichbar mit „Case ⟨Variable⟩ of“ in Pascal bzw. „switch(⟨Variable⟩)“ in C eingesetzt. Beispiel:

```
: TEST ( n $--$ )
  123 case? IF ." one-two-three" exit THEN
```

```
  0 case? IF ." Niete"         exit THEN
 16 case? IF ." 16=4*4"        exit THEN
. ." war ein Fehlschlag" ;
```

Typischer Einsatz als Syntaxdiagramm:
: ⟨*Name*⟩({⟨*Input*⟩}*n* − −{⟨*Output*⟩})
{⟨*Number*⟩ case? IF {⟨*word*⟩} exit THEN }
{⟨*word*⟩}({⟨*Input*⟩}*n* − −{⟨*Output*⟩}) ;

UWITHIN (u1 u2 u3 −− u2≤u1<u3): Gibt true zurück, wenn u_1 zwischen u_2 und u_3 liegt, wobei u_2 den Beginn des Bereichs angibt (true bei $u_1 = u_2$) und u_3 hinter dem Ende des Bereichs liegt (false bei $u_1 = u_3$). Das Vorzeichen wird dabei außer Acht gelassen.

Für Vergleiche mit 0 gibt es natürlich Abkürzungen:

0<> (n −− flag): Gibt true zurück, wenn n ungleich 0.
0= (n −− flag): Gibt true zurück, wenn n gleich 0.
0< (n −− flag): Gibt true zurück, wenn n kleiner als 0 ist.
0> (n −− flag): Gibt true zurück, wenn n größer als 0 ist.

Auch für Vergleiche von doppelt genauen Zahlen gibt es einige Befehle:

D= (d1 d2 −− d1=d2): Gibt true zurück, wenn d_1 gleich d_2 ist.
D< (d1 d2 −− d1¡d2): Gibt true zurück, wenn d_1 kleiner d_2 ist. D> kann man durch 2SWAP D¡ ersetzen.
D0= (d −− flag): Gibt true zurück, wenn d eine doppelt genaue 0 ist (zweimal 0 übereinander).

5. Limitierung

MIN (n1 n2 −− n1 / n2): Gibt die kleinere der beiden Zahlen zurück.
MAX (n1 n2 −− n1 / n2): Gibt die größere der beiden Zahlen zurück.
UMAX (u1 u2 −− u1 / u2): Wie MIN, das Vorzeichen wird nicht berücksichtigt.
UMIN (u1 u2 −− u1 / u2): Wie MAX, ohne Berücksichtigung des Vorzeichens.

Beispiel: Benötigt man einen Wert, der innerhalb eines gewissen Bereichs liegt, kann sich aber nicht sicher sein, daß ein solcher Wert übergeben wird, so kann man ihn mit der Sequenz
⟨*Untergrenze*⟩ `MAX` ⟨*Obergrenze*⟩ `MIN`
trimmen. „Obergrenze“ ist dabei der letzte Wert, der innerhalb des Bereichs liegt.

6. Programmablaufänderung

Normalerweise wird ein Programm sequentiell ausgeführt, Wort für Wort. Gäbe es nur diese Möglichkeit, so wäre man gezwungen, einen endlos langen Bandwurm zu schreiben. Wörter müssen beendet werden, Schleifen und bedingte Anweisungen müssen möglich sein. Auch muß man andere Wörter aufrufen können, aus den Interpretereigenschaften FORTHs kann man schließen, daß dies nicht nur statisch mit compilierten Wörtern möglich ist.

Bedingte Anweisungen benötigen eine Flag, wie sie bei Vergleichen auf den Stack gelegt wird. Natürlich kann die Flag auch der Wert einer Variable sein oder die Rückgabe eines anderen Wortes.

EXIT (−−): Beendet die Ausführung eines Wortes. Der Teil nach EXIT wird nicht mehr ausgeführt. EXIT steht innerhalb von bedingten Anweisungen, denn ansonsten wird ein Wort bis zum eigentlichen Ende ausgeführt, es wäre sinnlos, „toten“ Code zu definieren, der nie ausgeführt wird.

UNNEST (−−): Unterscheidet sich (außer im Namen) nicht von EXIT. Der eigentliche Unterschied ist die Verwendung: UNNEST wird von ; compiliert, in FORTH-Systemen, die einen einfachen Adreß-Compiler besitzen, kann man die beiden Wörter im Code unterscheiden und dann genau feststellen, wann das Wort tatsächlich zu Ende ist.

?EXIT (flag −−): Bedingter Ausstieg. Das Wort wird nur beendet, wenn *flag* true ist. ?EXIT hat dieselbe Wirkung wie IF EXIT THEN.

EXECUTE (cfa −−): Ruft das durch *cfa* gekennzeichnete Wort auf und kehrt nach der Ausführung zum Aufrufer zurück. Die CFA in FORTH ist ein Funktionszeiger.

PERFORM (addr −−): Ruft das Wort auf, dessen CFA an *addr* gespeichert ist. Entspricht @ EXECUTE.

>MARK (−− addr): Legt eine Marke für einen Vorwärtssprung an. Da die Distanz in bigFORTH ein 16-Bit-Wert ist, wird ein leeres 16-Bit-Feld compiliert und dessen Adresse auf den Stack gelegt. Dieses Feld muß von >RESOLVE gesetzt werden.

>RESOLVE (addr −−): Löst einen Vorwärtssprung auf. Der Sprung führt zu HERE, das Distanzfeld liegt an *addr*.

<MARK (−− addr): Legt eine Marke für einen Rückwärtssprung an. Der Sprung wird später compiliert.

<RESOLVE (addr −−): Löst einen Rückwärtssprung auf. Der Sprung führt an die Adresse *addr*, die Distanz wird am aktuellen Ende des Dictionaries compiliert.

BRANCH (−−): Springt unbedingt um die Distanz, die mit >MARK oder ¡RESOLVE hinter BRANCH compiliert wurde.

?BRANCH (flag −−): Springt bedingt um die Distanz, die mit >MARK oder ¡RESOLVE dahinter compiliert wurde. Gesprungen wird, wenn *flag* 0 (false) ist, ansonsten wird direkt hinter dem Distanzfeld weitergemacht.

?PAIRS (n1 n2 −−): Bricht mit dem Fehler „**unstructed**“ ab, wenn n_1 und n_2 nicht gleich sind. Dieses Wort wird von den Strukturwörtern benutzt, um Verletzungen der Struktur aufzuspüren. Da jede Struktur ihre eigene Nummer hat, können tatsächlich nur wohlgeformte Wörter definiert werden.

(DO (end start −−): Wird von DO compiliert. Es legt das alten Index- und Endregister auf den Returnstack und schreibt *start* in das Index- und *end* in das Endregister.

(?DO (end start −−): Wird von ?DO compiliert. Wie (DO, springt aber an das Ende der Schleife, wenn *start* und *end* gleich sind. Dazu ist hinter (?DO ein Branch hinter die Schleife compiliert (4 Bytes), der andernfalls übersprungen wird.

(LOOP (−−): Wird von LOOP compiliert und addiert 1 zum Indexregister. Wenn Index- und Endregister gleich sind, wird die Schleife beendet.

(+LOOP (n −−): Wird von +LOOP compiliert und addiert n zum Indexregister. Ansonsten wie (LOOP.

ENDLOOP (−−) restrict: Restauriert den alten Index- und Endregister. Wird von LOOP und +LOOP hinter (LOOP bzw. (+LOOP compiliert. Will man in einer Zählschleife mit EXIT aus einem Programm aussteigen, muß man zuvor ebenfalls mit ENDLOOP die alten Werte restaurieren.

Da die Strukturwörter nicht allein existieren können, werden sie im Zusammenhang als Syntaxdiagramm beschrieben. Ein Stackeffekt gilt nur für das direkt davorstehende Wort. Mit | abgetrennte Alternativen gelten nur für eine Zeile.

IF (flag −−)
{⟨*wort*⟩ }
[**ELSE** {⟨*wort*⟩ }]
THEN

BEGIN
{⟨*wort*⟩ }
{ **WHILE** (flag −−) {⟨*wort*⟩ } }(*n*)
REPEAT {**THEN** }(*n* − 1) | **UNTIL** {**THEN** }(*n*) (flag −−) | **AGAIN** {**THEN** }(*n*)

DO (end start −−) | **?DO** (end start −−) {⟨*wort*⟩ }
{ **LEAVE** {⟨*wort*⟩ } | **?LEAVE** (flag −−) {⟨*wort*⟩ } }
LOOP | **+LOOP** (n −−)

Nun im Einzelnen:

IF (flag −−) immediate restrict: Compiliert einen ?BRANCH hinter das dazugehörige ELSE bzw. THEN, wenn es kein ELSE gibt. Wenn *flag* 0 ist, wird dann der Teil hinter IF nicht ausgeführt, andernfalls der hinter ELSE.

ELSE (−−) immediate restrict: Compiliert einen BRANCH hinter das nächste THEN und löst den Branch-Offset von IF auf.

THEN (−−) immediate restrict: Löst den Branch-Offset vom letzten ELSE bzw. THEN auf. Der Programmteil hinter THEN wird auf alle Fälle ausgeführt.

Beispiele:

`: .flag ( flag $--$ ) IF ." Wahr" ELSE ." Falsch" THEN ;` ⟨RET⟩ `ok` Gibt den Wert einer Flag aus („Wahr“, wenn true, „Falsch“, wenn false)

`true .flag` ⟨RET⟩ `Wahr ok`

`false .flag` ⟨RET⟩ `Falsch ok`

`: .0? ( n $--$ ) dup 0<> IF ." Keine" THEN ." Null" ;` ⟨RET⟩ `ok` Gibt aus, ob *n* eine Null ist, oder keine.

`0 .0?` ⟨RET⟩ `Null ok`

`4711 .0?` ⟨RET⟩ `Keine Null ok`

BEGIN (−−) immediate restrict: Legt eine Marke an.

WHILE (flag −−) immediate restrict: Compiliert einen ?BRANCH hinter das dazugehörige REPEAT bzw. UNTIL. Solange *flag* nicht 0 ist, wird der Teil hinter WHILE ausgeführt, WHILE setzt die Schleife fort, wenn ihm „TRUE“ übergeben wird. WHILE kann beliebig oft in einer Schleife zwischen BEGIN und REPEAT bzw. UNTIL stehen.

REPEAT (−−) immediate restrict: Löst alle ?BRANCHes der WHILEs auf und compiliert einen BRANCH zum dazugehörigen BEGIN.

UNTIL (flag −−) immediate restrict: Löst ebenso wie REPEAT alle ?BRANCHes der WHILEs auf, compiliert aber einen ?BRANCH zum zugehörigen BEGIN, es wird also nur nach vorne gesprungen, wenn *flag* 0 ist. UNTIL bricht die Schleife ab, wenn ihm „TRUE“ übergeben wird.

Beispiele:

`: waitkey ( $--$ )` ⟨RET⟩ `compiling`

`  BEGIN  key? not  WHILE  ." Keine Taste gedrückt" cr  REPEAT` ⟨RET⟩ `compiling`

␣␣*."␣Tastencode:"␣key␣.␣;*⟨RET⟩ ok Gibt solange „Keine Taste gedrückt“ aus, solange keine Taste gedrück wurde.
Abweisende Schleife:
:␣waitkey2␣(␣$--$␣)⟨RET⟩ compiling
␣␣*BEGIN␣␣."␣Keine␣Taste␣gedrückt"␣cr␣␣key?␣UNTIL*⟨RET⟩ compiling
␣␣*."␣Tastencode:"␣key␣.␣;*⟨RET⟩ ok Wie WAITKEY, nur wird die Bedingung erst am Schleifendende ausgewertet, der Text wird also auf alle Fälle einmal ausgegeben.

DO (end start −−) immediate restrict: Compiliert (DO. Startet damit eine Schleife von *start* bis *end*.

?DO (end start −−) immediate restrict: Compiliert (?DO und LEAVE. (?DO überspringt den Code von LEAVE (einen Branch), wenn *start* und *end* nicht gleich sind. Andernfalls wird die Schleife verlassen, ehe sie beginnt.

LOOP (−−) immediate restrict: Compiliert (LOOP und ENDLOOP, löst alle LEAVEs und ?LEAVEs (mit ENDLOOPS) auf. Es steht am Ende einer Schleife mit der Schrittweite 1.

+LOOP (n −−) immediate restrict: Compiliert (+LOOP, ansonsten wie LOOP. Es steht am Ende einer Schleife mit wählbarer Sprungweite.

LEAVE (−−) immediate restrict: Compiliert einen BRANCH hinter das Schleifenende. Mit LEAVE wird die Schleife vorzeitig verlassen. LEAVE wird daher nur in bedingten Anweisungen eingesetzt, sonst würde die Schleife ja immer beim ersten Durchgang abgebrochen werden.

?LEAVE (flag −−) immediate restrict: Compiliert einen ?BRANCH hinter das Schleifenende. ?LEAVE verläßt die Schleife nur, wenn *flag* true ist. Ein IF LEAVE THEN kann durch ?LEAVE ersetzt werden.

ENDLOOPS (−−): Löst alle von LEAVE und ?LEAVE angelegten Branches hinter die Schleife auf.

BOUNDS (start len −− end start): Formt eine Start/Längenangabe in ihre Grenzen („bounds“), wobei das Ende nicht mehr zum Bereich gehört. Man setzt es ein, um Angaben im Format *start len* für DO bzw. ?DO aufzubereiten.

I (−− index) restrict: Liest das Indexregister aus und legt den Wert auf den Stack.

J (−− j-index) restrict: Liest das alte (von (DO bzw. (?DO auf den Returnstack gesicherte) Indexregister aus und legt es auf den Stack.

I' (−− end) restrict: Liest das Endregister aus und legt den Wert auf den Stack.

Beispiele:

:␣.index␣(␣end␣start␣$--$␣)␣?DO␣␣i␣.␣␣LOOP␣;⟨RET⟩ ok Gibt alle Zahlen von start bis end aus.
5␣0␣.index⟨RET⟩ 0 1 2 3 4 ok
10␣5␣.index⟨RET⟩ 5 6 7 8 9 ok
0␣0␣.index⟨RET⟩ ok
:␣.index2␣(␣start␣number␣$--$␣)␣bounds␣DO␣␣i␣.␣␣stop?␣?LEAVE␣␣LOOP␣;⟨RET⟩ ok
Gibt number Zahlen von start an aus. Kann mit ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ abgebrochen werden.
2␣7␣.index2⟨RET⟩ 2 3 4 5 6 7 8 ok
0␣0␣.index2⟨RET⟩ 0 1 2 3 ... Wird erst mit mit einem Druck auf ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ beendet.

7. Hauptspeicherzugriffe

Der Hauptspeicher ist fest in das Konzept von FORTH eingebunden. Jede Zahl kann auch als Adresse verstanden werden. Der Wert, der an dieser Adresse steht, wird mit @ ("fetch") geholt oder ein Wert wird mit ! ("store") in der Zelle dieser Adresse gespeichert.

Standard-FORTH hat einen linearen (nicht segmentierten) 16-Bit-Adreßraum, ein 32-Bit-System natürlich einen 32-Bit-Adreßraum. Um Realtime-Eigenschaften zu verwirklichen, ist der Adreßraum real, die Zugriffszeit ist also im Gegensatz zum virtuellen Speicher genau definiert. Der Adreßraum muß (gerade in einem 32-Bit-System) nicht vollständig sein. Auch können Teile als ROM (Read Only Memory) verwirklicht sein, die sich dann nicht ändern lassen.

Um auf unterschiedlichen Prozessoren eine schnelle Ablaufgeschwindigkeit zu erreichen, wird ein Zugriff auf Misalignement soweit wie möglich verhindert. Misalignements sind Adressen, von denen nicht mit einer minimalen Anzahl an Buszyklen gelesen oder geschrieben werden kann. Konkret: Beim 68000 z. B. kann ein 16-Bit-Wort nur von einer geraden Adresse gelesen werden. Bytezugriffe werden ausgeführt, indem eine Bushälfte ausgeblendet wird. Es wird dann bei geraden Adressen nur das höherwertige Byte am Bus benutzt, bei ungeraden Adressen das niederwertige Byte.

Misalignements führen beim 68000 zu einem "Address Error". Aufwendige 32-Bit-Prozessoren haben oft eine Schaltung, um solche Zugriffe durchzuführen, aber diese Schaltung bremst ziemlich: Beim Intel 80486 z. B. braucht ein Zugriff auf eine durch 4 teilbare Adresse einen Taktzyklus, ein Zugriff auf ein Misalignement dagegen 4 Taktzyklen!

Normalerweise können solche Probleme gar nicht auftreten, da Adressen nur symbolisch gehandhabt werden. Man greift in FORTH nicht auf eine bestimmte Speicherstelle zu. Die Adresse z. B. einer Variable wird vom Compiler vergeben. Zudem kann sie sich ändern, wenn man das System mit SAVESYSTEM sichert und in einer anderen Speicherkonfiguration wieder startet. Die Zahlen, die als Adressen interpretiert werden, sind also ihrerseits nur Ausdrucksmittel für symbolische Objekte, wie Variablen, CFAs etc.

Mögliche Fehlermeldungen:

Address Error Bei @ oder ! wurde auf eine ungerade Adresse zugegriffen. Solche Misalignements sollte man vermeiden oder eventuell mit ODD@ bzw. ODD! darauf zugreifen.

Bus Error Auf eine Adresse kann nicht zugegriffen werden. Dieses Signal wird von einem Peripheriebaustein an den Prozessor geleitet. Entweder wurde versucht, auf eine Adresse im ROM zu schreiben oder auf eine nicht belegte Adresse zugegriffen. Dieser Fehler deutet darauf hin, daß etwas im Aufbau des Wortes nicht stimmt, denn der Compiler erzeugt keine Adressen, auf die nicht zugegriffen werden kann.

@ (addr −− n): Liest den 32-Bit-Wert, der an der Adresse *addr* gespeichert ist.
! (n addr −−): Speichert den 32-Bit-Wert *n* an der Adresse *addr*.
C@ (addr −− char): Liest an *addr* ein Byte aus und legt es auf den Stack.
C! (char addr −−): Schreibt das Byte *char* an die Adresse *addr*.
W@ (addr −− 16b): Liest einen 16-Bit-Wert an der Adresse *addr*. Hier kann auch auf Misalignements zugegriffen werden.
W! (16b addr −−): Speichert den 16-Bit-Wert 16b bei *addr*. Auch hier kann auf Misalignements zugegriffen werden.
ODD@ (addr −− n): Wie @, erlaubt aber auch Zugriffe auf Misalignements (ungerade Adressen).

ODD! (n addr −−): Wie !, erlaubt aber Zugriffe auf Misalignements.

CTOGGLE (char addr −−): Verknüpft *char* und das Byte an *addr* mittels xoder und speichert das Ergebnis an *addr*. Dient dazu, Bitflags zu ändern.

+! (n addr −−): Addiert *n* zu dem 32-Bit-Wert an addr und speichert das Ergebnis in *addr* ab.

ODD+! (n addr −−): Wie +!, kann aber auch auf ungerade Adressen zugreifen.

ON (addr −−): Speichert das Symbol TRUE (-1) an *addr*. Mit ON werden Schalter angeschaltet.

OFF (addr −−): Speichert 0 an *addr*. Schalter werden ausgeschaltet.

PUSH (addr −−) restrict: Rettet den Wert an der Stelle *addr*, um ihn nach Ende des Wortes, aus dem PUSH aufgerufen wurde, zu rekonstruieren. Beispiel:

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

.HEX gibt Zahlen hexadezimal aus, ohne die Zahlenbasis dauerhaft zu verändern. Die Änderung von BASE (auf 16) nach dem Aufruf von PUSH gilt also nur innerhalb von .HEX.

8. Veränderungen im Speicher

Nicht nur zwischen Stack und Speicher kann kommuniziert werden, es ist auch möglich, einen Teil des Speichers in einen anderen zu kopieren, mit einem Zeichen zu füllen oder zu löschen.

CMOVE (addr1 addr2 n −−): Kopiert *n* Zeichen von *addr*1 nach *addr*2 und dahinter. Es wird zeichenweise kopiert, dabei wird bei *addr*1 angefangen und in Richtung höherer Adressen weitergemacht. CMOVE arbeitet füllend, wenn *addr*2 im Bereich zwischen *addr*1 und *addr*1+*n* liegt, da die Kopie schon bei *addr*2 liegt, wenn das Kopierprogramm diese Adresse erreicht — CMOVE kann dann als Füllroutine für N Bytes eingesetzt werden. Beispiel:

```
"‘ Dies ist ein Text"’ count 2dup over 4+ swap 4- cmove⟨RET⟩ ok
type⟨RET⟩ DiesDiesDiesDiesD ok
```

CMOVE> (addr1 addr2 n −−): Wie CMOVE, nur wird „rückwärts“ kopiert. Es wird also bei $addr1 + n - 1$ angefangen und in Richtung niedriger Adressen weitergemacht. CMOVE> wird benutzt, wenn CMOVE aufgrund der sequenziellen Kopie unbrauchbar ist. Natürlich gibt es auch Situationen, in denen CMOVE> nicht wie gewünscht arbeitet, dann muß CMOVE benutzt werden. Beispiel:

```
"‘ Dies ist ein Text"’ count 2dup over 4+ -rot 4- cmove>⟨RET⟩ ok
type⟨RET⟩ tTextTextTextText ok
```

MOVE (addr1 addr2 n −−): Kopiert *n* Zeichen ab *addr*1 nach *addr*2 und allerdings wird dabei die erforderliche Kopierrichtung automatisch gewählt. MOVE ist in bigFORTH für große Datenmengen optimiert und kopiert diese wesentlich schneller als CMOVE.

PLACE (addr1 n addr2 −−): Speichert *addr*1*n* als counted String nach *addr*2. Dabei wird *n* als erstes Byte nach *addr*2 geschrieben, der Speicherbereich wird dahinterkopiert.

FILL (addr len char −−): Füllt den Bereich *addr len* mit dem Zeichen *char*.

ERASE (addr len −−): Löscht den Bereich *addr len* (füllt ihn mit 0-Bytes), entspricht 0 FILL.

9. Die Userarea

bigFORTH ist multitaskingfähig. Damit solche Eigenschaften möglich sind, braucht jeder Task einen Bereich, in dem taskspezifische Werte gespeichert werden. Dieser Bereich heißt in FORTH "User-Area". Diese Userareas sind miteinander verbunden, bei einem Taskwechsel wird von einer zur nächsten gewechselt. Dabei steht am Start entweder der Prozessoropcode "trap #3", um den Task zu wechseln, oder, sollte der Task inaktiv sein, ein "jmp", der dann an die darauf folgende Adresse der nächsten Userarea in der Kette springt.

Hinter dieser Link-Adresse wird beim Taskwechsel der Stackpointer gespeichert. Auf dem Stack liegen Instruction Pointer, Returnstack Pointer, Index- und Endregister, die auch für jeden Task spezifisch sein müssen. Alle anderen Register können nach einem Taskwechsel verändert sein.

ORIGIN (−− addr): Hier werden die Uservariablen beim Sichern des Systems gespeichert und von hier nach dem Start geholt. Die Uservariablen in der Userarea sind nur eine Kopie dieses Bereichs, Veränderungen können also rückgängig gemacht werden.

UP@ (−− addr): Legt die Adresse des Userpointers auf den Stack.

UP! (addr −−): Setzt den Userpointer neu. Dazu muß an dieser Adresse aber auch eine funktionsfähige Userarea sein!

S0 (−− useraddr): Hier wird der Stackboden gespeichert.

R0 (−− useraddr): Hier wird der Returnstackboden gespeichert.

DP (−− useraddr): Dictionary Pointer. Der DP zeigt auf HERE.

OFFSET (−− useraddr): Relikt aus der „Steinzeit". In dieser Variable wird beim Direktzugriff auf Massenspeicher ein Offset gespeichert, aus dem das Laufwerk berechnet wird ($40000 z. B. heißt Laufwerk B:).

BASE (−− useraddr): In BASE wird die aktuelle Zahlenbasis gespeichert.

OUTPUT (−− useraddr): Zeigt auf den Output-Block, in dem in einem Array die Adressen der gerätespezifischen Output-Wörter stehen.

INPUT (−− useraddr): Zeigt auf den Input-Block, in dem in einem Array die Adressen der gerätespezifischen Input-Wörter stehen.

ERRORHANDLER (−− useraddr): Zeigt auf eine Routine, die im Fehlerfall für die Ausgabe eines Strings und den Restart des Systems sorgt. Diese Routine hat den Stackeffekt (string −−). Sie wird von ABORT" und ERROR" aufgerufen und heißt im Kernel (ERROR, in BIGFORTH.PRG BOXHANDLER.

VOC–LINK (−− useraddr): Zeiger auf die verkettete Liste aller Vokabulare (siehe VOCABULARY).

UDP (−− useraddr): User Dictionary Pointer: Gibt an, wieviele Bytes in der Userarea schon belegt sind.

TSTART (−− useraddr): Tasks können beim Sichern nicht „eingefroren" werden. Deshalb steht in TSTART die Adresse einer Routine, die den Start eines Tasks übernimmt. Stackeffekt: (Taskaddr −−). S. AUTOSTART (Kapitel 7.6).

UALLOT (n −− oldudp): Erhöht den UDP um n und legt den alten UDP auf den Stack. Mit UALLOT reserviert man einen n Bytes großen Bereich, der ab *oldupd* (Offset zu UP) beginnt.

USER (−−) ⟨*Name*⟩:⟨*Name*⟩ (−− useraddr): Legt eine Uservariable an. ⟨***Name***⟩ selbst legt beim Aufruf die ihm zugeordnete Useradresse *useraddr* auf den Stack.

10. Compilerbefehle

HERE (−− addr): Ende des Dictionaries. Hier wird compiliert.

ALLOT (n −−): Erhöht den DP um n. Es werden damit n Bytes ab HERE reserviert. Der neue HERE befindet sich hinter dem Bereich.

PAD (−− addr): Textpuffer, in bigFORTH $64=&100 Bytes hinter HERE.

, (n −−): Compiliert die Zahl n in der Speicherzelle am Ende des Dictionaries. HERE ist dann 4 Bytes höher. Sollte HERE nicht gerade sein, erscheint eine „Address Error"-Meldung. Zur Disziplinierung der Programmierer gibt es kein ODD, , verwenden Sie im Zweifelsfall ALIGN vor , .

C, (8b −−): Compiliert ein Zeichen am HERE. Der DP wird um eins erhöht.

W, (16b −−): Compiliert ein 16-Bit-Wort am HERE. Misalignements sind erlaubt, da ja auch W@ auf ungerade Speicherstellen zugreifen kann.

ALIGN (−−): Compiliert ein Leerzeichen ($20), wenn HERE ungerade ist und erhöht damit den DP auf die nächste gerade Zahl. Verhindert Misalignements.

EVEN (n1 −− n2): Erhöht n_1 um eins, wenn es ungerade ist.

CFA! (cfa addr −−): Speichert die cfa an $addr$ als 68000-Befehl jsr adresse.

NOOP! (addr −−): Speichert an $addr$ drei NOPs ($4E71) hintereinander. Die Wirkung ist dieselbe wie ' NOOP $addr$ CFA!, der Code aber schneller.

(COMPILE (−−): Wird von COMPILE compiliert. Compiliert das Wort, dessen CFA hinter dem Aufruf von (COMPILE steht, am HERE.

COMPILE (−−) ⟨***Word***⟩ **immediate restrict:** Compiliert (COMPILE und die CFA des Wortes ⟨***Word***⟩. Bei der Ausführung wird dann das Wort ⟨***Word***⟩ compiliert.

LITERAL (n −−) immediate restrict: Compiliert n als Literal. Außerhalb des Compilers verwendet man LITERAL, um einmalige Berechnungen in den Interpreterteil zu schieben, ohne das Programm der Befehle zur Berechnung zu berauben und dadurch unlesbar zu machen.

Beispiel (als Zeile in einer Programmdefinition):

```
[ &365 &100 * &100 4 / + ( Tage im 20. Jahrhundert ) ] Literal
```

wirkt wie `&36525`

ASCII (−− 8b) ⟨***char***⟩ **immediate:** Liest ⟨***char***⟩ und wandelt es in seinen Ascii–Wert. Im Programm compiliert es diesen Wert als Literal.

11. Stringbefehle

Strings (Zeichenketten) werden in FORTH als "Counted Strings" abgelegt. Hier wird die Länge der Zeichenkette im ersten Byte angegeben. Es gibt auch noch andere Möglichkeiten, die Länge eines Strings anzugeben, beispielsweise mit einem Stringendezeichen (Beispiel: 0-terminated Strings mit 0-Byte am Ende).

Damit man die unterschiedlichen Stringformate leicht mit denselben Befehlen bearbeiten kann, wird ein String auf dem Stack als Bytefeld mit Adresse und Länge auf den Stack gelegt (addr count .. −− ..). Ein String wird also durch zwei Stackelemtente charakterisiert.

COUNT (addr0 −− addr len): Legt Anfangsadresse des eigentlichen Textes und Länge eines counted Strings (auf den $addr0$ zeigt) auf den Stack.

/STRING (addr count n −− addr+n count-n): Schneidet von einem String die ersten n Bytes ab. Beispiel:

```
"'␣Dies␣ist␣ein␣Text"'␣count␣5␣/string␣type⟨RET⟩ ist ein Text ok
```

n Bytes von hinten schneidet ein einfaches - ab.

SKIP (addr1 count1 char −− addr2 count2): Alle Zeichen *char* werden vorne am String abgeschnitten. Beispiel:

"␣....Text"␣count␣ascii␣.␣skip␣type⟨RET⟩ `Text ok`

SCAN (addr1 count1 char −− addr2 count2): Alle Zeichen bis zum ersten *char* werden abgeschnitten. Bespiel:

"␣Ein␣Text"␣count␣ascii␣T␣scan␣type⟨RET⟩ `Text ok`

−SKIP (addr1 count1 char −− addr2 count2): Wie SKIP, nur von hinten:

"␣Text...."␣count␣ascii␣.␣-skip␣type⟨RET⟩ `Text ok`

−SCAN (addr1 count1 char −− addr2 count2): Wie SCAN, nur von hinten:

"␣Ein␣Text"␣count␣ascii␣T␣-scan␣type⟨RET⟩ `Ein T ok`

CAPITAL (char −− CHAR): Kleinbuchstaben (a-z, ä, ö und ü) werden in Großbuchstaben (A-Z, Ä, Ö und Ü) gewandelt.

CAPITALIZE (string −− STRING): Alle Buchstaben des counted Strings *string* werden in Großbuchstaben gewandelt. Dies geschieht direkt im String, also bleiben die Adressen dieselben - CAPITALIZE arbeitet „destruktiv“.

,“ (−−) ⟨*String*⟩”: Compiliert die Zeichenkette ⟨***String***⟩, die durch Anführungszeichen begrenzt wird, als counted String. Vorsicht! Da kein ALIGN durchgeführt wird, kann HERE ungerade werden.

”LIT (−− addr) restrict: Holt einen als counted String (mit ALIGNement) hinter dem Aufruf des Programms, das “LIT benutzt, abgelegten Text. Weil’s so kompliziert ist, folgen die Erklärungen der nächsten vier Befehle als Beispiele.

“ (−− addr) ⟨*String*⟩” immediate: Compiliert (“ und ⟨***String***⟩ als counted String. Führt ein Alignement durch. Zur Laufzeit wird die Adresse des Strings auf den Stack gelegt und hinter den String gesprungen.

(“ (−− addr) restrict: Holt mit “LIT die Adresse des counted Strings, der hinter (“ compiliert wurde. “LIT verändert auch die Returnadresse, d. h. (“ kehrt hinter den String zurück.

.“ (−−) ⟨*String*⟩” immediate restrict: Compiliert (.“ und ⟨***String***⟩. Zur Laufzeit wird String auf dem Terminal ausgegeben.

(.“ (−−) restrict: Holt mit “LIT die Adresse des Strings und gibt ihn mit COUNT TYPE auf dem Terminal aus.

BL (−− $20): Konstante: Der Ascii-Wert des Leerzeichens (Blank).

−TRAILING (addr len1 −− addr len2): Löscht abschließende Leerzeichen, wirkt wie BL -SKIP.

SPACE (−−): Gibt ein Leerzeichen aus.

SPACES (n −−): Gibt n Leerzeichen aus.

12. Der TIB und Screen Interpretation

FORTH enthält bekanntlich einen Zeileninterpreter. Ebenso werden nachgeladene Screens interpretiert; der Compiler selbst ist ja auch nur ein FORTH-Wort, das zunächst ausgeführt werden muß. Der TIB oder der gerade geladene Screen wird als Eingabestrom (Inputstream) behandelt, es wird also sequenziell zugegriffen.

#TIB (−− useraddr): In #TIB wird die Anzahl eingegebener Zeichen gespeichert.

PUSH#TIB (−− useraddr): Bei einer Umleitung von TIB wird hier #TIB gesichert, allerdings nur, wenn PUSH#TIB vorher leer war. Bei einem Fehler wird hieraus die Länge des ursprünglichen TIBs geholt.

>TIB (−− useraddr): Zeiger auf den TIB.

>IN (−− useraddr): In >IN wird die Anzahl der bereits interpretierten Zeichen gespeichert. Ist >IN @ gleich oder größer als #TIB @, wird die Interpretation beendet.

BLK (−− useraddr): Ist der Inhalt von BLK nicht 0, so wird der Block geladen, dessen Nummer in BLK gespeichert ist. Andernfalls wird der TIB interpretiert.

TIB (−− addr): Die Eingaben vom Terminal landen hier und werden interpretiert.

SPAN (−− useraddr): Variable, die die Zahl der eingegebenen Zeichen enthält.

QUERY (−−): Liest eine Zeile vom Terminal in den TIB. Es werden 80 Zeichen eingelesen, auch wenn eine Zeile des Terminals möglicherweise eine andere Länge hat (z. B. in der niedrigen Auflösung).

LOADFILE (−− addr): Hier wird die Datei gespeichert, von der eingelesen wird. Ist der Inhalt 0, so wird direkt (physikalisch) von Diskette oder Platte gelesen.

SOURCE (−− addr len): Gibt die Adresse und Länge der zu interpretierenden Source (Inputstream, Screen oder TIB) zurück.

WORD (char −− addr): Liest, bis ein *char* im Inputstream ist. Führende Leerzeichen werden übersprungen. Es wird ein counted String zurückgegeben. Der Puffer (auf den auch *addr* zeigt) liegt direkt nach dem HERE. Der zurückgegebene String darf nicht länger als 32 Bytes (mit Countbyte) sein.

(WORD (char addr0 len −− addr): Wird von WORD benutzt. Hier wird noch angegeben, welcher Bereich (*addr*0 und *len*) durchsucht wird.

PARSE (char −− addr len): Sucht im Inputstream nach *char*. Alle Zeichen bis char sind in dem Bereich *addr len* gefunden worden. Dieser Bereich ist ein Bestandteil des Inputstreams!

NAME (−− addr): Wie BL WORD CAPITALIZE. Sucht eine von Leerzeichen begrenzte Zeichenkette im Inputstream und wandelt das gefundene Wort in Großbuchstaben.

(LOAD (blk offset −−): Lädt vom Screen *blk* ab dem Zeichen $offset$.

LOAD (blk −−): Lädt den Screen *blk*.

+LOAD (offset −−): Addiert zum aktuellen Screen (BLK @) $offset$ und lädt diesen Screen.

THRU (from to −−): Lädt die Screens von $from$ bis *to* einschließlich.

+THRU (from+ to+ −−): Lädt die nächsten Screens von $from+$ bis $to+$, diese Werte werden zum aktuellen Screen addiert.

−−> (−−) immediate: Beendet die Interpretation des aktuellen Screens und zwingt den Interpreter, gleich beim nächsten weiterzumachen. −−> kann auch während der Compilation eines Wortes, das über mehrere Screens geht, benutzt werden (unschön!).

LOADFROM (blk −−) ⟨*File*⟩: Lädt den Screen *blk* der Datei ⟨***File***⟩.

INCLUDE (−−) ⟨*File*⟩: Lädt den Screen 1 (Loadscreen) der Datei ⟨***File***⟩.

PROMPT (−−): Gibt den Prompt aus (" ok" im Interpreter-Modus, " compiling" im Compiler-Modus).

(QUIT (−−): Hauptschleife des FORTH-Interpreters. Gibt den Status aus (.STATUS), liest eine Zeile vom Terminal, interpretiert sie, gibt den Prompt aus, geht mit CR in die nächste Zeile und fängt von vorn an.

'QUIT (−−): Deferred Word, das normalerweise (QUIT enthält. Es kann auf eine andere Hauptschleife des FORTH-Systems umgesetzt werden, z. B. den Event-Dispatcher der GEM-Library.

QUIT (−−): Löscht den Returnstack und startet die Hauptschleife 'QUIT.

.STATUS (−−): Deferred Word: Gibt eine Statusmeldung aus. .STATUS wird später von .BLK besetzt und gibt die aktuelle Blocknummer aus, sowie bei Dateiwechsel die neue Datei, von der nun geladen wird.

13. Kommentare

Kommentare sollen Programme im Sourcecode dokumenieren. Diese Dokumentation soll das Programm wartbar machen. In FORTH gibt es einige Regeln zur Dokumentation, die unbedingt eingehalten werden sollen:

Der Stackeffekt eines jeden Wortes muß hinter dem Namen in einer Klammer mit Doppelstrich (.. −− ..) festgehalten werden. Diese Klammer kann weggelassen werden, wenn es keinen Stackeffekt gibt (−−).

Die erste Zeile eines Screens ist die Index-Zeile. Hier steht als Kapitelüberschrift ein zusammenfassender Kommentar zu allen Wörtern des Screens — ein einfaches Aufzählen der Wörter ist allenfalls in Libraries statthaft, aber auch hier ist es oft möglich, einen gemeinsamen Nenner zu finden.

((−−) ⟨*Kommentar*⟩) immediate: Überliest alle Zeichen bis zur nächsten). (klammert Kommentare aus, die nicht interpretiert werden.

.((−−) ⟨*String*⟩) immediate: Gibt alle Zeichen bis zum nächsten) sofort aus. Es dient dazu, während des Compilierens Meldungen auszugeben.

\ (−−) immediate: Kommentiert alles bis zum Ende der Zeile aus.

\\ (−−) immediate: Kommentiert alles bis zum Ende des Screens aus.

\NEEDS (−−) ⟨*Wort*⟩: Ist ⟨*Wort*⟩ vorhanden, wird der Rest der Zeile auskommentiert, ansonsten ausgeführt. Dient zum Nachladen oder -definieren dringend benötigter Wörter. Beispiel:

```
\needs floating  include FLOAT.SCR
```

Das Beispiel lädt die Datei FLOAT.SCR nach, wenn das Vokabular FLOATING nicht vorhanden ist — es kann dann ganz sicher auf die FP-Routinen zugegriffen werden.

14. Compiler-Variablen

LAST (−− addr): Enthält die NFA des zuletzt definierten Wortes.

LASTCFA (−− addr): Enthält die CFA des zuletzt definierten Wortes.

LASTOPT (−− addr): Enthält die Adresse des Optimizing-Wertes des zuletzt compilierten Makros. Dieser 16-Bit-Wert wird von MACRO direkt hinter dem Ende des eigentlichen Codes angelegt.

LASTDES (−− addr): In den 4 Bytes von LASTDES sind die letzten beiden Optimizing-Werte der letzten beiden Makros gespeichert. Sie stehen in der Reihenfolge ihres Eingangs, das ältere liegt also an der niedrigeren Adresse. Dadurch stoßen das Pushbyte des älteren und Take-Byte des jüngeren aufeinander, die beiden können mit LASTDES 1+ W@ geholt werden. LASTDES wird vom optimierenden Compiler benutzt.

STATE (−− useraddr): STATE enthält true, wenn der Compiler angeschaltet ist, sonst false.

15. Compiler-Optionen

HIDE (−−): Macht das letzte Wort „unsichtbar“, es wird aus der verketteten Liste der Wörter ausgehängt. Der Colon-Compiler ermöglicht so, daß man während der Definition eines Wortes dieses selbst nicht compilieren kann. Dafür kann man Worte nochmal definieren (die Warnung „exists!“ wird ausgegeben!) und bei dieser Definition auf das alte Exemplar mit demselben Namen zugreifen.

REVEAL (−−): Macht das letzte Wort wieder sichtbar, hängt es in die Kette ein. REVEAL ist nicht hundertprozentig sauber, das letzte Wort wird einfach als neues Ende der Kette gesetzt, sollten nachher Wörter definiert worden sein, ohne LAST zu ändern, so werden diese wieder ausgehängt.

RECURSIVE (−−) immediate: Wie REVEAL. Da RECURSIVE ein immediate-Word ist, wird es während der Definition eines Wortes eingesetzt, um einen Selbstaufruf (Rekursion) zu compilieren, die vom Compiler normalerweise ja verhindert wird.

IMMEDIATE (−−): Setzt das Immediate-Bit des letzten Wortes. Dieses Wort wird dann auch während der Compilation ausgeführt.

RESTRICT (−−): Setzt das Restrict-Bit des letzten Wortes. Es kann dann nur noch vom Compilier benutzt werden (ob compiliert oder interpretiert, entscheided das Immediate-Bit). Der Interpreter weist Restrict-Wörter mit der Meldung „compile only“ zurück.

MACRO (−−): Definiert das letzte Wort als Makro. Es wird dann nicht mehr ein jsr bzw. bsr zu diesem Wort compiliert, sondern der Code kopiert (außer den zwei Bytes für das RTS am Ende). Zudem wird noch ein leeres OptimizingWort angelegt (Inhalt: 0).

16. Der Heap

In bigFORTH gibt es wie in volksFORTH einen Wort-Heap. Hier werden Wortheader abgelegt, die später nicht mehr benötigt werden. Der Heap befindet sich zwischen Userarea und Stackboden. Auch Labels für den Assembler finden hier Platz.

HEAP (−− addr): Gibt die Anfangsadresse des Heaps zurück. Da der Heap in Richtung niedrigerer Adressen wächst, beginnt an dieser Adresse der „jüngste“ Teil des Heaps.

HALLOT (n −−): Vergrößert den Heap um n Bytes. Der Heap wächst zwischen Stack und Userarea, also muß der Inhalt des Stacks bei HALLOT verschoben werden. Im Gegensatz zu ALLOT ist ein n Bytes großer Bereich ab HEAP nach (!) diesem Aufruf belegt, bei ALLOT ist ein n Bytes großer Bereich ab HERE vor dem Aufruf von ALLOT belegt!

HEAP? (addr −− flag): Gibt true zurück, wenn *addr* im Heap liegt.

HMACRO (−−): Wie MACRO. Nur wird der Wortrumpf auf den Heap gelegt. Das Wort kann dann während der Compilation verwendet werden, verschwindet aber nach dem Sichern des Systems (oder einem SAVE bzw. CLEAR, das den Heap löscht). Im gesicherten System wird dann kein Platz für dieses Wort belegt. Kopiert wird aber nur, wenn auch schon der Wortkopf auf dem Heap liegt. Als HMACRO definierte Wörter dürfen nicht mit COMPILE weiterverwendet werden, ebenfalls darf ihre CFA nicht mit [’] im Code fixiert werden.

?HEAD (−− addr): Enthält eine Flag, ob der Wortkopf im Dictionary oder im Heap angelegt wird. Ist ?HEAD gelöscht, so wird im Dictionary angelegt, sonst im Heap und ?HEAD wird um eins erhöht.

— **(−−):** Setzt ?HEAD auf −1. Dadurch wird genau der nächste Wortkopf auf den Heap gelegt. Beispiel:

```
| :␣UNSICHTBAR␣."␣UNSICHTBAR␣verschwindet␣nach␣einem␣CLEAR"␣;⟨RET⟩ ok
:␣SICHTBAR␣UNSICHTBAR␣;⟨RET⟩ ok
words⟨RET⟩ SICHTBAR |UNSICHTBAR ⟨andere Wörter⟩
unsichtbar⟨RET⟩ UNSICHTBAR verschwindet nach einem CLEAR ok
clear␣words⟨RET⟩ SICHTBAR ⟨andere Wörter⟩
```

HALIGN (−−): Führt einen Align für den Heap durch. Der Heap beginnt dann an einer geraden Adresse.

WARNING (−− addr): Schalter. Steht in WARNING true, so wird die Meldung "exists" ausgegeben (Umgekehrt wie in volksFORTH, aber jetzt logisch!).

MAKEVIEW (−− %fffffffbbbbbbbbb): Gibt den 16-Bit-Wert zurück, der in das View-Field gehört. Die niederwertigen 9 Bits sind die aktuelle Blocknummer (es sind damit Nummern von 1-512 möglich), die oberen 7 Bits sind die Dateinummer (127 Dateien sind möglich). Die Datei 0 ist der direkte Zugriff, der Block 0 bedeutet vom TIB eingelesen ("Hand made").

17. Der Colon-Compiler

FORTH-Wörter werden mit dem Colon-Compiler compiliert. Colon bedeutet Doppelpunkt („:"). Das Wort : erzeugt nur den Wortheader und schaltet den eigentlichen Compiler mit] an. Compiler und Interpreter „picken" sich Wort für Wort aus dem Inputstream heraus, der Interpreter führt die gefundenen Worte mit EXECUTE aus (wenn sie nicht restrict sind), der Compilier compiliert mit CFA, ihre CFAs, immediate Words führt er aus. Damit sind Compilerstreuerungen und -erweiterungen möglich.

HEADER (−−) ⟨*Name*⟩:⟨*Name*⟩ (??): Erzeugt einen Wortheader und das Längenfeld. Da für das erzeugte Wort (noch) kein Code existiert, kann es noch nicht aufgerufen werden.

CREATE (−−) ⟨*Name*⟩:⟨*Name*⟩ (−− addr): Erzeugt einen Wortheader eine CFA. Das erzeugte Wort ist ausführbar und liefert (wie VARIABLE) die Adresse der PFA zurück. Nur muß man die PFA selbst anlegen.

DOES> (−− addr) immediate: Compiliert ;CODE und R>. Vor DOES> muß der definierende Teil eines Defining-Words stehen, hinter DOES> die Methode für diese Klasse User-defined-Words. CREATE und DOES> spielen eng zusammen. Syntax:
: ⟨***Defining Word***⟩ ({*input*} −−) \ ⟨***Name***⟩:⟨***Name***⟩ ({*input*} −− {*output*})
CREATE ⟨***PFA anlegen***⟩
DOES> ⟨***PFA auswerten, Funktion ausführen***⟩ ;

: (−− 0) (VS voc −− current) ⟨*Name*⟩:⟨*Name*⟩ ({*input*} −− {*output*}): Colon-Compiler. Erzeugt einen Wort-Header und schaltet den Compiler an. Syntax:
: ⟨***Name***⟩ {⟨***Wort***⟩ } ; {⟨***Option***⟩ }
Optionen sind Wörter wie IMMEDIATE, RESTRICT oder MACRO. Damit die wohlgeformte Struktur des Wortes überprüft werden kann, legt : eine 0 auf den Stack.

!LENGTH (−−): Speichert die Länge des letzten Wortes in dessem Length-Field. Es wird dabei angenommen, daß das Wort fertig compiliert ist. Steht im Length-Field bereits ein Wert ungleich 0, so wird der alte Wert belassen.

; (0 −−) immediate: Compiliert UNNEST und schaltet den Compiler aus. Es muß die 0 von : auf dem Stack liegen, nur dann ist das Wort wohlstrukturiert.

CONSTANT (N −−) ⟨*Name*⟩:⟨*Name*⟩ (−− N): Erzeugt eine Konstante. Jeder Aufruf der Konstante legt dabei den in die PFA compilierten Wert N auf den Stack. Es ist sichergestellt, daß der Wert tatsächlich aus der PFA geholt wird, er kann dort also im Nachhinein gepatcht werden.

VARIABLE (−−) ⟨*Name*⟩:⟨*Name*⟩ (−− addr): Erzeugt ein Wort und legt eine Zelle als Raum für eine globale Variable an. Das erzeugte Wort legt die Adresse der Zelle (also seine PFA) auf den Stack.

ALIAS (cfa −−) ⟨*Name*⟩:⟨*Name*⟩ (⟨*input*⟩ −− ⟨*output*⟩): Erzeugt einen neuen Namen für ein bereits existierendes Wort. Beide Wortköpfe haben dieselbe CFA, damit denselben Code.

DEFER (−−) ⟨*Name*⟩:⟨*Name*⟩ ({*input*} −− {*output*}): Legt eine Vordefinition an. Dieses Wort ist bereit, ein anderes in sich aufzunehmen, dieses wird dann ausgeführt. Das deferred Word dient generell einem bestimmten Zweck (Defer), was genau jetzt getan wird, bestimmt das Wort, auf das umgeleitet wird.

IS (cfa −−) ⟨*Deferred Word*⟩: Setzt ein deferred Word auf das Wort cfa. Diese CFA wird beim Aufruf des deferred Words aufgerufen, alle Wörter, die das deferred Word aufrufen, verhalten sich so, als sei cfa compiliert worden.

(FIND (string thread −− string false / nfa true): Sucht im Vocabular *thread* nach einem Wort, das denselben Namen hat wie *string*. Bei erfolgreicher Suche wird die nfa und *true* zurückgegeben, andernfalls die Stringadresse und $false$.

FIND (string −− string false / cfa n): Sucht nach dem Wort *string*. Dabei wird der VS von oben nach unten durchgegangen, es wird also zuerst das Context-Vocabulary durchsucht, zuletzt ROOT. Bei erfolgreicher Suche wird die CFA und ein Wert ungleich Null zurückgegeben, andernfalls die Stringadresse und $false$. n gibt an, ob das Wort immediate und/oder restrict ist:

−1: Weder noch.

−2: restrict.

1: immediate.

2: immediate restrict.

' (−− cfa) ⟨*Wort*⟩: Gibt die CFA des nächsten Wortes im Inputstream zurück. Wird das Wort nicht gefunden, bricht ' mit „Hä?“ ab.

['] (−− cfa) ⟨*Wort*⟩ immediate: Wie ', nur wird die CFA im Programm gleich als Literal gespeichert, während ' hier erst bei der Ausführung des Programms ausgeführt wird.

[COMPILE] (−−) ⟨*Wort*⟩: Compiliert ⟨*Wort*⟩ auf alle Fälle. [COMPILE] wird unbedingt benötigt, wenn ein immediate-Word compiliert werden soll.

NULLSTRING? (string −− string true / false): Gibt $false$ zurück, wenn der counted String *string* 0 Bytes lang ist (Countbyte=0). Andernfalls wird die Stringadresse und *true* zurückgegeben.

?STACK (−−): Überprüft den Stack. Bei einem Stackleerlauf wird mit „**Stack empty**“ abgebrochen, bei einem Stacküberlauf mit „**Stack full**“ Sollte das Dictionary so groß sein, daß es mit dem Stack direkt in Kollision kommt, wird „**Dictionary full**“ ausgegeben und das zuletzt definierte Wort wieder vergessen. Bei diesen Fehlern wird der Stack gelöscht. Ist alles ok, so wird er nicht verändert. ?STACK wird vom Interpreter/Compiler vor jedem Wort und am Ende der Zeile/des Screens aufgerufen, um Stackfehler frühzeitig abzufangen.

>INTERPRET (−−): Setzt die Ausführung des Interpreters/Compilers fort. >INTERPRET kehrt nicht in die aufrufende Ebene zurück, sondern zur Ebene darüber. Dadurch kann

der Interpreter/Compiler als Schleife ausgeführt werden, die zwar nicht rekursiv ist, aber trotzdem nicht wohlstrukturiert sein muß.

INTERPRET (−−): Ruft den Interpreter/Compiler auf, der den TIB oder den gerade geladenen Block interpretiert.

NOTFOUND (string −−): Kann *string* weder als Wort noch als Zahl verstanden werden, wird es dem deferred Word NOTFOUND übergeben. Hier kann man Erweiterungen einhängen.

NO.EXTENSIONS (string −−): Bricht mit der Fehlermeldung „Hä?“ ab. Es ist ursprünglich in NOTFOUND eingehängt und bedeutet, daß es keine Erweiterungen gibt.

18. Wortstruktur

Ein compiliertes Wort beginnt mit View Field und Link Field. Über letzteres sind alle Wörter in ihrem Vokabular als verkettete Liste zusammengefaßt. Dahinter steht das Name Field, das Length Field und das Code Field (Header), zuletzt das Parameter Field (Body).

Oft hat man nur eine Adresse (NFA, CFA oder PFA) und benötigt eine andere. Die Adresse des View Fields und des Link Fields kann man leicht aus der NFA berechnen, ebenso die NFA aus der Adresse des Link Fields. Name Field und Code Field haben unterschiedliche Längen, das Code Field in anderen 32-Bit-FORTH-Systemen ist ausschließlich 4 Bytes lang, in bigFORTH bei Kernelworten ebenfalls, bei anderen aber 6 Bytes.

(NAME> (nfa −− addr): Liefert das Ende der NFA (Name Field Address). Hier steht entweder ein Zeiger auf die CFA oder das Length-Field des Wortes.

NAME> (nfa −− cfa): Rechnet NFA in CFA um.

NFA? (thread cfa −− nfa / false): Sucht nach einem Wort im Vocabular *thread* mit der CFA cfa. Zurückgegeben wird entweder die NFA oder (bei Mißerfolg) false.

>NAME (cfa −− nfa / false): Sucht den Namen des Wortes mit der CFA cfa in allen Vokabularen (im Current-Vocabular zuerst) und gibt die NFA bzw. $false$ zurück. Gegenspieler zu NAME¿.

>BODY (cfa −− pfa): Rechnet die CFA in die PFA um. Da die CFA nicht als einfache Adresse gespeichert ist, sondern als jsr adresse (im Kernel bsr adresse), muß man mit >BODY umrechnen. >BODY kann auch das Offsetfeld von Uservariablen, die als Makro realisiert sind, und den Body von deferred Words berechen.

BODY> (pfa −− cfa): Gegenspieler von >BODY. BODY> fuktioniert nur, wenn vor der PFA ein bsr adresse oder ein jsr adresse steht.

CFA@ (cfa −− addr): Holt die Adresse aus dem Code-Field. Konkret wird die Adresse berechnet, an die das hier stehende jsr bzw. bsr springt. Steht kein solcher 68000-Opcode an der Stelle, wird die CFA wieder zurückgegeben (Verdacht auf Assemblerroutine).

.NAME (nfa −−): Gibt den Namen nfa aus. Ist die NFA 0, so wird “???” ausgegeben. Liegt sie im Heap, so wird vor das Wort “|” gesetzt. Die Ausgabe wird mit einem Leerzeichen abgeschlossen.

19. Der optimierende Compiler

bigFORTH besitzt einen optimierenden Compiler. Er versucht FORTH-Code als möglichst schnellen Maschinencode zu compilieren. Dabei wird vom Standard abgewichen, denn der Standard basiert auf dem sogenannten „threaded Code“. Diesen Ausdruck übersetzt man

etwa mit „gefädeltem Code“. Gemeint ist damit, daß der Compiler die CFAs der compilierten Wörter aneinander reiht.

Der innere Interpreter liest diese Adressen der Reihe nach, liest von dort die CFA aus und springt an diese Stelle. Bei einem FORTH-Wort ist dies der innere Interpreter, der nun weitermacht und eine Adresse nach der anderen liest. So springt ein FORTH-Interpreter hauptsächlich von einem Wort zum nächsten, bis er schließlich in den Primitives, den Maschinensprachewörtern anlangt. Erst dort kann er wirklich etwas tun.

Damit bigFORTH Maschinencode erzeugt, wendet es folgende Taktik an:

1. Ein FORTH-Wort besitzt kein Code Field. Gleich nach dem Header steht der compilierte Maschinencode. Bei mit CREATE definierten Wörtern steht hier ein „Jump to SubRoutine“ (jsr), im Kernel ein „Branch to SubRoutine“ (bsr).

2. FORTH-Wörter werden als jsr Adresse oder als bsr Adresse compiliert. Der innere Interpreter, der den Aufruf besorgt, ist im Prozessor-Opcode enthalten.

3. Kurze Primitives (auch ganz kurze FORTH-Wörter) werden als Makros compiliert. Makros sind Folgen von wenigen Assemblerbefehlen, die zusammen eine Funktion ausführen können. Da bigFORTH kein vollständiger Makroassembler ist, können an diese Makros bei der Codegenerierung keine Parameter übergeben werden. Makros sind also kurze Codestückchen, die ins Programm statt eines jsr eingesetzt werden.

4. Der 68000 ist ein registerorientierter Prozessor. Berechnungen finden also in Registern statt. Für FORTH-Code ist daher ein Verschieben von Werten vom Stack in Register und von Register auf den Stack unumgänglich. Damit hier zwischen zwei Makros keine unnötige Arbeit erledigt wird, optimiert der Compiler die Schnittstelle zwischen den Makros. So kann folgene Sequenz ganz weggelassen werden:

   ```
   move.l  D0,-(A6)        ;Datenregister D0 auf den Stack legen

   move.l  (A6)+,D0        ;TOS in D0 laden
   ```

   Andere Sequenzen können zumindest deutlich verkürzt werden. Da ein Hauptspeicherzugriff auf ein Langwort (32 Bit) auf dem ST mindestens 8 Taktzyklen benötigt, der 16-Bit-Opcode allein 4 Taktzyklen, wurden im vorherigen Beispiel 24 Taktzyklen (3μs) gespart. Um diese Optimierung zu ermöglichen, benötigt der Compilier Informationen, die im Optimizer-Wort jedes Makros stehen.

T&P (takemode pushmode −−): Setzt das Optimizing-Wort des zuletzt erzeugten Makros. *takemode* und *pushmode* sind Konstanten, die davon abhängen, mit welchem Opcode das Makro beginnt oder endet. Ist *takemode* bzw. *pushmode* 0, so ist vorne bzw. hinten keine Optimierung möglich.

	beginnt mit		**endet mit**	
:D0 (−− n):	`move.l`	`(A6)+,D0`	`move.l`	`D0,-(A6)`
:A0 (−− n):	`movea.l`	`(A6)+,A0`	`move.l`	`A0,-(A6)`
:>R (−− n):	`move.l`	`(A6)+,-(A7)`	−−	
:DUP (−− n):	−−		`move.l`	`(A6),-(A6)`
:OVER (−− n):	−−		`move.l`	*xx*`(A6),-(A6)`
:+LOOP (−− n):	`add.l`	`(A6)+,D5`	−−	
:COMP (−− n):	`cmpm.l`	`(A6)+,(A6)+`	−−	
:LIT (−− n):	−−		`move.l`	`#`*xx*`,-(A6)`
:FLAG (−− n):	`move.l`	`(A6)+,D0`	`s`*xx* `D0`	
			`ext.w`	`D0`
			`ext.l`	`D0`
			`move.l`	`D0,-(A6)`
:R> (−− n):	−−		`move.l`	`(A7)+,-(A6)`
:@ (−− n):	−−		`move.l`	`(A0),-(A6)`
:R@ (−− n):	−−		`move.l`	`(A7),-(A6)`
:+ (−− n):	`move.l`	`(A6)+,D0`	`add.l`	`D0,(A6)`
:− (−− n):	`move.l`	`(A6)+,D0`	`sub.l`	`D0,(A6)`
:OR (−− n):	`move.l`	`(A6)+,D0`	`or.l`	`D0,(A6)`
:AND (−− n):	`move.l`	`(A6)+,D0`	`and.l`	`D0,(A6)`
:XOR (−− n):	`move.l`	`(A6)+,D0`	`eor.l`	`D0,(A6)`
:D0\− (−− n):	`move.l`	`(A6)+,D0`	`move.l`	`D0,-(A6)`
			(N-Bit im CCR nicht richtig gesetzt!)	
:D0\F (−− n):	`move.l`	`(A6)+,D0`	`move.l`	`D0,-(A6)`
			(CCR nicht richtig!)	

OPTTAB (−− addr): Enthält die Tabelle aller möglichen Verkürzungen zwischen Makroende und Anfang des nächsten Makros, die mit diesem System möglich sind. Das Format:
|Pushbyte|Takebyte|Verkürzung|Zwischencodelänge|Zwischencode|.

#OPT (−− len): Konstante, gibt die Länge der OPTTAB an.

OPT? (−− addr): Schalter. Ist OPT? off, ist der Optimizer abgeschaltet, d. h. Makros werden in voller Länge kopiert und nicht verkürzt.

!LASTDES (−−): Initialisiert LASTDES für die Compilation des nächsten Wortes.

REL (−− addr): Schalter, ob ein Wort relokatibel compiliert werden soll (REL on) oder nicht (REL off). Hiermit ist nicht der Relocater gemeint, sondern eine sonst übliche Eigenschaft von FORTH-Wörtern: Ein FORTH-Wort ist frei verschiebbar, da innerhalb des Wortes nur relativ adressiert wird, nach „draußen“ aber ausschließlich absolut. In bigFORTH wird aus Optimierungsgründen auch nach „draußen“ relativ adressiert (wenn es geht). Setzt man REL on, wird diese Optimierung ausgeschaltet.

CFA, (cfa −−): Compiliert das Wort *cfa*. Sämtliche Optimierungsmöglichkeiten werden berücksichtigt. CFA, ersetzt das , eines F83-Systems für CFAs.

[(−−) immediate: Schaltet den Compiler aus.

] (−−): Schaltet den Compiler wieder an. Innerhalb der beiden eckigen Klammern können Ausdrücke interpretiert werden. Bricht der Compiler eine Programmdefinition ab, weil er ein Wort nicht findet, so kann mit] an der fehlerhaften Stelle wieder aufgesetzt und die Definition beendet werden.

T] (−−): Schaltet den Table-Compiler an, der wie ein F83-System nur die CFAs der einzelnen Wörter compiliert, ohne ausführbaren Maschinencode zu erzeugen.

TABLE: (−−) ⟨*Name*⟩ {⟨*Wort*⟩ } [:⟨*Name*⟩ (−− addr): Benutzt den Table-Compiler um eine Sprungtabelle anzulegen. Auf die kann dann mit ⟨*Index*⟩ CELL∗ ⟨*Name*⟩ + PERFORM zugegriffen werden.

20. Vokabulare

Vokabulare dienen zur Strukturierung des Dictionaries. Vokabulare können ausgeblendet werden, bei der Suche nach Wörtern muß also nicht das ganze Dictionary durchsucht werden. Außerdem können in mehreren Vokabularen Wörter mit gleichem Namen definiert werden, welches nun zur Ausführung kommt oder compiliert wird, entscheidet die Reihenfolge der Vokabulary im Vocabulary Stack. Zuerst wird das Context Vocabulary durchsucht.

VP (−− addr): Vocabulary Pointer: Enthält einen Offset, der vom Beginn des Vocabulary Stacks (VS) auf das Context Vocabulary zeigt. Der VS selbst beginnt direkt hinter diesem Offset, es ist hier Platz für 16 Vokabulare. FIND geht bei der Suche nach Wörtern vom Context Vocabulary aus durch den VS durch und sucht in jedem eingetragenen Vokabular nach dem Wort.

CONTEXT (−− addr) (VS Voc −− Voc): Bildet den Zeiger auf das aktuelle Vokabular (Context Vocabulary), das zuerst durchsucht wird. Um die Bedeutung des VPs zu demonstrieren, hier die Definition:

```
: CONTEXT  VP DUP @ + CELL+ ;
```

CURRENT (−− addr): Variable, die das Current Vocabulary festhält. Definitionen werden in dieses Vokabular compiliert.

ALSO (−−) (VS Voc −− Voc Voc): Verdoppelt das Context Vocabulary. Es ist dann nach Aufruf eines anderen Vokabulars noch in der Suchreihenfolge.

TOSS (−−) (VS Voc −−): Löscht das Context Vocabulary, das darunterliegende wird neues Context Vocabulary.

DEFINITIONS (−−) (VS voc −− voc): Legt das Context Vocabulary als Current Vocabulary fest.

VOCABULARY (−−) ⟨*Name*⟩:⟨*Name*⟩ (−−) (VS voc −− ⟨*Name*⟩): Erzeugt ein Vokabular. Mit ⟨*Name*⟩ wird das Vokabular als neues Context Vocabulary gesetzt (Es wird mit einem anderen Vokabular der Kontext gewechselt, gleiche Wörter können damit eine andere Bedeutung bekommen, andere Wörter können benutzt werden). Das Parameterfeld ist folgendermaßen aufgebaut:

|Thread|Coldthread|Voc-link|

Thread ist ein Zeiger auf die verkettete Wörterliste des Vokabulars. Coldthread ist der Inhalt dieses Zeigers nach dem Systemstart. Voc-link ist ein Zeiger, der alle Vokabulare als verkettete Liste verbindet. Der Start dieser Liste steht in der Uservariablen VOC-LINK.

FORTH (−−) (VS voc −− FORTH): Vokabular FORTH. Dieses Vokabular ist das Basisvokabular, in dem alle Standard-FORTH-Wörter definiert sind.

ROOT (−−) (VS voc −− ROOT): Vokabular ROOT. Das Wurzel-Vokabular, das auf alle Fälle im VS stehen muß, da sonst nicht mehr weitergearbeitet werden kann.

ONLY (−−) (VS vocs −− ROOT ROOT): Setzt den VS auf ROOT ROOT. Dies ist die Mindestbelegung, die noch erlaubt ist. ROOT ist hier auch Current Vocabulary.

ONLYFORTH (−−) (VS vocs −− ROOT FORTH FORTH): Setzt den VS auf FORTH FORTH ROOT (Ausgabereihenfolge von Order). Wie ONLY FORTH ALSO DEFINITIONS. FORTH ist dann auch das Current Vocabulary.

ORDER (−−) (VS −−): Vocabulary-Stackdump. Zuletzt (mit etwas mehr Abstand) wird das Current-Vocabulary ausgegeben.

WORDS (−−): Listet alle Wörter des Context-Vocabularies auf, dabei wird mit den zuletzt definierten begonnen. Der Schwall dieser Wörter kann mit ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ abgebrochen werden, mit einer anderen Taste unterbrochen und fortgesetzt.

Im Vokabular ROOT sind folgende Wörter definiert:

SEAL (−−): Löscht alle Wörter im Vokabular ROOT.

Des weiteren sind die Wörter **ONLY**, **FORTH**, **WORDS**, **ALSO**, und **DEFINITIONS** in ROOT als Alias definiert.

21. Eigene Fehlermeldungen

FORTH besitzt ein eigenes System für Fehlermeldungen. Auch die Error recovery wird vom System übernommen. Natürlich kann man dieses Fehlersystem in eigene Hände nehmen, um einer eigenen Applikation ein komfortables System der Fehlermeldung in die Hand zu geben.

END–TRACE (−−): Schaltet den Tracer aus. Das Tracebit im Status-Register des 68000 wird gelöscht.

CLEARSTACK (n0 .. ndepth −−): Löscht den Stack.

(ABORT (−−): Wird in das deferred Word 'ABORT eingesetzt. Setzt den TIB zurück und schaltet den Tracer aus.

'ABORT (−−): Deferred Word: Führt eine Teilreinitialisierung des Systems nach einem ABORT, ABORT“ oder ERROR“ durch. Damit später problemlos weitere Teilreinitialisierungen eingehängt werden können, muß das Wort, das in 'ABORT hängt, (ABORT heißen und im Vokabular FORTH definiert sein.

ABORT (−−): Löscht den Stack und führt eine Teilreinitialisierung des Systems (Warmstart) durch.

(ERROR (string −−): Gibt den Puffer von WORD (ab HERE) aus, also das letzte interpretierte/compilierte Wort, das einen Fehler erzeugt hat, und die Meldung *string*. Danach wird die Hauptschleife QUIT aufgerufen. (ERROR hängt im ERRORHANDLER.

LASTERR (−− addr): In dieser Variable steht die letzte Fehlernummer. Ist bisher alles reibungslos verlaufen, steht hier eine 0, sonst die TOS-Fehlernummer des letzten Fehlers. FORTH-interne Fehler werden unter der Nummer -1 („allgemeiner Fehler“) gespeichert.

(ABORT“ (flag −−) restrict: Wird von ABORT“ compiliert und gibt die hinter seinem Aufruf als counted String compilierte Meldung an die in ERRORHANDLER gespeicherte Routine weiter, wenn *flag* true ist. Ist *flag* false, passiert nichts.

ABORT“ (flag −−) ⟨*Meldung*⟩” immediate restrict: Bricht mit ⟨***Meldung***⟩ ab, wenn *flag* nicht 0 ist. Der Stack wird dabei gelöscht.

ERROR“ (flag −−) ⟨*Meldung*⟩” immediate restrict: Wie ABORT“, nur wird der Stack nicht gelöscht.

SCR (−− useraddr): Enthält den Screen, der vom Editor gerade bearbeitet wird. Nach einem Fehler beim Laden eines Screens steht dessen Nummer in SCR.

R# (−− useraddr): Enthält die Position des Cursors im vom Editor gerade bearbeiteten Screen. Nach einem Fehler beim Laden steht der Cursor hinter dem fehlerhaften (fehlerauslösenden) Wort.

22. Zahlenausgabe

FORTH besitzt einige Worte, um Zahlen in Ziffernstrings umzuwandeln. Die Zahlenbasis für die Wandlung steht in der Uservariablen BASE, sie ist somit frei wählbar. Die zur Zahlenwandlung gehörenden Befehle können nur im Zusammenhang angewendet werden, für sich allein sind sie sinnlos.

Typisches Beispiel für die Zahlenwandlung ist das Wort .00 aus dem Kapitel 3.8:

```
: .00 ( n -- )
extend under dabs <# # # ascii , hold #s rot sign #> type ;
```

Der Zahlenpuffer liegt in dem Speicherbereich vor dem Textpuffer PAD. Direkt vor dem Pad steht ein Zeiger auf den Pufferanfang, der Puffer selbst wird direkt vor diesem Zeiger nach vorne aufgebaut. Es haben maximal 64 Zeichen in ihm Platz (die längste doppelt genaue Zahl binär dargestellt), mehr führt zu Fehlern.

<# (d −− d): Startet die Zahlenumwandlung. Der Zahlenpuffer wird initialisiert. Da eine doppelt genaue Zahl umgewandelt wird, sollte sie hier schon auf dem Stack liegen, auch wenn sie erst später gebraucht wird.

#> (d −− addr count): Beendet die Zahlenumwandlung. Der Rest der Zahl (meist eine doppelt genaue 0) wird vom Stack genommen und der Zahlenstring als Adresse und Länge auf den Stack gelegt.

HOLD (char −−): Fügt das Zeichen *char* vorne an den Zahlenstring und setzt den Zeiger auf den Zahlenpuffer um eins nach vorne.

 # (d −− d/base): Wandelt die letzte Ziffer von d in ein Ascii-Zeichen und hängt dieses vorne an den Zahlenstring an. d wird dabei durch die Basis geteilt und damit liegt die nächste Ziffer als letzte Ziffer von d zur Umwandlung bereit auf dem Stack. Gewandelt wird also immer von der letzten Ziffer an.

#S (d −− 0.): Wandelt alle verbleibenden Ziffern, mindestens aber eine 0. Es liegt dann auf alle Fälle eine doppelt genaue 0 auf dem Stack.

SIGN (n −−): Fügt ein „-“ in den Zahlenstring, wenn n negativ war.

Für die standardisierte Ausgabe von Zahlen gibt es in bigFORTH eine Reihe von Befehlen, die die üblichen Bereiche abdecken. Terminologie: Ausgaben werden mit einem „.“ (Punkt) getätigt. Prefixe wie u und d (oder gemischt) kennzeichnen die auszugebende Zahl als vorzeichenlos (unsigned, u) oder doppelt genau (d), der Postfix r gibt an, daß die Zahl rechtsbündig in einem r Zeichen großen Feld ausgegeben wird. r wird dabei als TOS übergeben.

D.R (d r −−): Gibt die doppelt genaue Zahl d (mit Vorzeichen) rechtsbündig in einem r Zeichen großen Feld aus. Braucht d mehr Platz als im Feld vorhanden, so werden die überstehenden Ziffern rechts vom Feld ausgegeben.

UD.R (ud r −−): Gibt die doppelt genaue Zahl ud ohne Vorzeichen rechtsbündig in einem r Zeichen großen Feld aus.

.R (n r −−): Gibt n (mit Vorzeichen) rechtsbündig in einem r Zeichen großen Feld aus.

U.R (u r −−): Gibt u (ohne Vorzeichen) rechtsbündig in einem r Zeichen großen Feld aus.

D. (d −−): Gibt d aus und hängt ein Leerzeichen an.

UD. (ud −−): Gibt ud vorzeichenlos aus und hängt ein Leerzeichen an.

. (n −−): Gibt n mit Vorzeichen aus, hängt ein Leerzeichen an.

U. (u −−): Gibt u aus und hängt ein Leerzeichen an.

.S (−−): Gibt einen Stackdump aus. Jede Zahl des Stacks wird als vorzeichenbehaftete Zahl ausgegeben, gestartet wird beim TOS. Es werden höchstens 16 Zahlen ausgegeben.
HEX (−−): Setzt Base auf 16. Das System ist dann im Hexadezimalmodus.
DECIMAL (−−): Setzt Base auf 10. Das System ist im Dezimalmodus.

23. Zahleneingabe

Das Format der Zahleneingabe ist im 1. Kapitel beschrieben, hier zur Wiederholung nochmal das Format in BNF:
Zahl::= [-][%|&|$]⟨***Ziffer***⟩[,|.]{⟨***Ziffer***⟩[,|.]}

DIGIT? (char −− n true / false): Wenn *char* eine Ziffer in der aktuellen Zahlenbasis ist, wird ihr Wert n und *true* zurückgegeben, sonst $false$.
ACCUMULATE (d addr n −− d∗base+n addr): Multipliziert d mit der aktuellen Zahlenbasis und addiert n dazu. *addr* zeigt auf die nächste auszulesende Ziffer, wird aber nicht beeinflußt.
CONVERT (d1 addr1 −− d2 addr2): Konvertiert so lange, bis es hinter *addr*1 keine Ziffern mehr findet. Die Adresse, an der die erste Nicht-Ziffer steht, und die bisher gewandelte Zahl werden zurückgegeben. CONVERT ist als
`: CONVERT BEGIN count digit? WHILE accumulate REPEAT 1- ;`
definiert.
DPL (−− useraddr): In dieser Variable steht die Anzahl der Ziffern plus eins, die nach dem letzten Punkt bzw. Komma standen oder eine -1. DPL wird von NUMBER? benutzt.
NUMBER? (string −− string false / d 0> / n −1): Versucht den counted String *string* in eine Zahl umzuwandeln. Ist das nicht möglich, so wird die Stringadresse und $false$ zurückgegeben. Enthält die Zahl . oder , , so wird eine doppelt genaue Zahl zurückgegeben und die Anzahl der Ziffern hinter dem letzten Punkt oder Komma plus 1, andernfalls eine einfach genaue Zahl und -1.
NUMBER (string −− d): Wandelt *string* in die doppelt genaue Zahl d. Schlägt dies fehl, so wird mit der Meldung „?“ abgebrochen. Gewandelt wird mit NUMBER?, somit werden Zahlen, in denen kein . oder , steht, nur erweitert, zusätzliche Ziffern sind trotzdem nicht signifikant.

24. Der Relocater

bigFORTH ist relokatibel. Diese Eigenschaft ist für ein normales TOS-Programm unbedingt erforderlich. Es gibt in TOS keine feste Adresse, an denen Programme gestartet werden, wie die TPA in CP/M. Beim Programmstart wird einfach der größte zusammenhängende Bereich reserviert. Das Programm muß sich darauf einrichten, daß es hier auch ablaufen kann.

Obwohl der 68000 eine Reihe von Möglichkeiten bietet, frei verschiebbare Programme zu schreiben, sind diese doch eingeschränkt. So geht ein relativer Sprung über eine maximale Distanz von 32 KByte. Deshalb läßt man nach dem Laden des Programms zuerst einen „Relocater“ über das Programm laufen, der alle Adressen anpaßt. Dazu wird das Programm so gesichert, daß es eigentlich nur an der Adresse 0 laufen könnte — was allerdings in der Praxis nie passieren kann, da dort die Vektoren des Prozessors stehen.

Die Adressen selbst werden durch die Relocater-Information gekennzeichnet, denn der Relocater will natürlich nicht raten müssen, was eine Adresse ist und was ein Befehl.

TOS selbst besitzt einen Relocater, der hinter dem Programm eine Byte-Liste benutzt, in der die Abstände von Adresse zu Adresse gespeichert sind. Die Liste beginnt mit einem Langwort, in dem der erste Offset steht. Sie endet mit einem 0-Byte. Bei einem 1-Byte werden 254 Bytes übersprungen, ohne die nächste Adresse anzupassen. Damit können beliebige Abstände zwischen Adressen überbrückt werden.

Diese Methode hat einen entscheidenden Nachteil: Änderungen in der Liste sind nur mit großem Aufwand möglich. Also kann sie allenfalls im Nachhinein erzeugt werden. Dazu müssen alle Adressen aber schon markiert sein. Für das Markierungsproblem sind in bigFORTH zwei Lösungen implementiert, die beide ihre speziellen Vor- und Nachteile haben.

Die erste Idee ist, die Markierung (Relocaterinfo) in einem Bitstring zu speichern. Ein Bit bezieht sich dabei auf zwei Bytes, da Adressen ja nur an geraden Speicherstellen liegen können. Auf einen Bitstring läßt sich frei zugreifen, er läßt sich auch frei verändern. Jede zu relozierende Adresse wird durch ein gesetztes Bit markiert. Der Compiler muß also solche Adressen markieren. Dazu dienen Befehle wie A!, ALITERAL und A,. Leider muß auch der Programmierer immer wieder Adressen markieren, muß Adreßvariablen mit AVARIABLE statt VARIABLE anlegen und Adreßkonstanten mit ACONSTANT.

Diese unterschiedliche Behandlung paßt nicht mit dem F83-Standard zusammen, denn in FORTH werden Adressen wie Zahlen behandelt, ohne irgendwelche Unterschiede. Eine Abkehr von dem Prinzip verletzt das Prinzip der Typenlosigkeit in FORTH.

Zudem kommt noch die Fehlerträchtigkeit des Verfahrens. Solange das System an derselben Adresse bleibt, läuft alles, egal, ob man die Adressen markiert oder nicht. Ändern kann sich nur nach dem Sichern und Neustarten etwas, und das bei den meisten Benutzern auch nicht, da die Speicherkonfiguration meist gleich bleibt. Erst wenn man die Größe der RAM-Disk ändert oder ein anderes Accessory lädt, wird bigFORTH auch in einen anderen Speicherbereich geladen. Dann erst machen sich versehentlich nicht markierte Adressen bemerkbar (oder versehentlich als Adressen markierte Zahlen, alles kann passieren.)

Die zweite Lösung verzichtet auf die Verwaltungsinformation. Das System wird zweimal aufgerufen, es wird ihm eine Kommandozeile übergeben, die z. B. eine Datei nachlädt und damit eine eigene Applikation compiliert. Beide Systeme sind schließlich identisch, bis auf den entscheidenden Unterschied, daß sie an unterschiedlichen Adressen stehen (müssen sie auch, da sie beide gleichzeitig im Speicher gehalten werden). Durch Vergleich kann man so die zu relozierenden Adressen herausfinden und nachträglich markieren.

Diese Lösung ist unabhängig vom FORTH-System selbst, dieses muß nur eine Kommandozeile als FORTH-Befehlzeile interpretieren können und beim Verlassen ein wieder startbares System hinterlassen. Dieses Tool, mit dem die Lösung implementiert ist, heißt RELOCATE.PRG und ist auf der blauen Diskette zu finden. Die Bedienung wurde im Kapitel 2.21 erklärt.

Der Bitstring der Verwaltungsinformation hat denselben Aufbau wie eine Zeile des Monochrombildschirms. Bits mit höherer Position liegen an höherer Adresse, aber an niedrieger Wertigkeit im selben Byte.

B$ON (B$addr pos −−): Setzt das Bit *pos* im Bitstring *B$addr* auf eins.

B$OFF (B$addr pos −−): Setzt das Bit *pos* im Bitstring *B$addr* auf null.

B$X (B$addr pos −−): Invertiert das Bit *pos* im Bitstring *B$addr*. War es vorher eins, so wird es null und umgekehrt.

B$@ (B$addr pos −− flag): Fragt das Bit *pos* ab. Ist es null, so wird false zurückgegeben, bei eins wird true zurückgegeben.

B$MOVE (B$addr start ziel len −−): Schiebt einen *len* Bit langen Bereich im Bitstring von der Position *start* nach *ziel*. Überlappende Bereiche können wie bei CMOVE nur in Richtung niedriger Positionen geschoben werden, andernfalls gibt es eine Fehlfunktion.

B$ERASE (B$addr start len −−): Löscht einen *len* Bit langen Bereich ab *start* im Bitstring.

RELON (addr −−): Markiert *addr* im Relocater-Bitstring. Dazu wird die Differenz von *addr* und der Startadresse des FORTH-Systems (FORTHSTART) durch zwei geteilt und das Bit mit dieser Position auf eins gesetzt.

RELOFF (addr −−): Löscht die Markierung von *addr* im Relocater-Bitstring.

A! (addr1 addr2 −−): Wie !, markiert aber *addr*2 im Relocater-Bitstring. A! dient nur zur Initialisierung von Feldern. Es muß ja auch nur einmal angewandt werden, danach kann man ganz normal ! verwenden.

V! (n addr −−): Wie !, hebt aber eine Markierung von *addr* im Relocater-Bitstring auf.

A, (n −−): Wie , , markiert aber HERE im Relocater-Bitstring.

RELMOVE (addr1 addr2 len −−): Wie CMOVE, die Marken im Bitstring werden aber mitkopiert.

ALITERAL (n −−) immediate restrict: Wie LITERAL, markiert die als Literal compilierte Zahl als Adresse.

ACONSTANT (Addr −−) ⟨*Name*⟩:⟨*Name*⟩ (−− Addr): Wie CONSTANT, *Addr* wird mit A, compiliert.

AVARIABLE (−−) ⟨*Name*⟩:⟨*Name*⟩ (−− addr): Wie VARIABLE, die reservierte Speicherzelle wird als Adresse markiert.

AUSER (−−) ⟨*Name*⟩:⟨*Name*⟩ (−− useraddr): Wie USER, nur wird der reservierte Platz im Feld, auf das ORIGIN zeigt, als Adresse markiert. Da eine neue Uservariable auch nur den UDP des Main-Tasks beeinflußt, braucht auch in den Userareas der anderen Tasks nichts markiert werden.

25. Listing

C/L (−− $40): Konstante: Ein Screen hat $40=64 Zeichen pro Zeile (characters per line).

L/S (−− $10): Konstante: Es gibt $10=16 Zeilen pro Screen (lines per screen).

LIST (blk −−): Listet den Screen *blk* der aktuellen Datei aus. *blk* wird dabei in der Variablen SCR gespeichert. LIST gibt in der ersten Zeile die aktuelle Datei, den Screen und das Laufwerk aus (Dr 0, wenn nicht im Direktzugriff). In den nächsten 16 Zeilen werden die Zeilen des Screens mit vorangestellter Zeilennummer ausgegeben. Die Zeilennummer wird rechtsbündig in einem 2 Zeichen großen Feld ausgegeben, zwischen Nummer und Zeile ist noch ein Leerzeichen.

26. Tasker Primitives

PAUSE (−−): Regt einen Taskwechsel an. In bigFORTH wird ein Task nur auf expliziten Befehl gewechselt. Während auf Ein/Ausgabe gewartet wird, muß ein Task seine Kontrolle abgeben, d. h. solange PAUSE aufrufen, bis die Ein/Ausgabe beendet ist. Auch bei längeren Berechnungen muß immer wieder PAUSE aufgerufen werden.

LOCK (addr −−): Belegt ein „Semaphor“. Semaphore sind Schlösser, die der Zugriffsberechtigung dienen. Ein Semaphor ist frei, wenn sein Inhalt 0 ist, im belegten Zustand ist der UP (d. h. die Taskadresse) des besitzenden Tasks in dem Semaphor gespeichert. LOCK wartet nun solange, bis das Semaphor frei ist und belegt es dann für den gerade laufenden Task.

Semaphore benötigt man für Resourcen, die geteilt werden müssen, wie Laufwerkzugriffe o. ä. Sie sollen verhindern, daß zwei Tasks durch einen gleichzeitigen Zugriff z. B. auf denselben Drucker ein Chaos anrichten. Das Problem dieser Lösung: Das „Dead-Lock“: Es entsteht, wenn zwei Tasks sich gegenseitig aussperren, also beide gegenseitig auf die Aufgabe eines Locks warten und damit das alte Lock nicht aufgegeben werden kann. bigFORTH ignoriert dieses Problem, dies ist der einfachste bisher bekannte Algorithmus.

UNLOCK (addr −−): Gibt ein Semaphor wieder frei. Dazu muß es natürlich auch im Besitz des gerade laufenden Tasks sein.

27. Massenspeicherzugriffe

FORTH greift blockweise auf Massenspeicher zu. bigFORTH verfügt neben dem einfachen Direktzugriff auf das einzelne Laufwerk auch ein umfangreiches Fileinterface, das sämtliche Möglichkeiten des TOS ausnützt und zudem noch einen Environmentpfad bietet, in dem die Dateien gesucht werden.

Die eigentliche Blockverwaltung wird dem Memory Management überlassen. Soviel sei nur gesagt: Ein Block im Blockpuffer besteht aus einer Verwaltungsinformation und dem eigentlichen Datenbereich, der $400=1024 Bytes=1 KByte belegt. Die Verwaltungsinformation ist systemspezifisch und wird im Kapitel 7.1 genauer erklärt. Nur eines ist hier wichtig: Verändert man den Inhalt des Puffers, muß man die Update-Flag setzen, dann wird der Puffer irgendwann auch auf Diskette zurückgeschrieben.

Dieses Blockkonzept verwirklicht teilweise einen virtuellen Speicher. Auf Teile des Massenspeichers oder einer Datei kann (fast) wie auf den Hauptspeicher zugegriffen werden.

Der Puffer hat eine garantierte Mindestgröße von zwei Blöcken. Ansonsten kann jeder angeforderte Block bei einer weiteren Anforderung oder einem Taskwechsel wieder auf Diskette zurückgeschrieben werden. Er muß dann erneut angefordert werden. In bigFORTH ist es wichtig, zu wissen, daß auch ein Block, der noch nicht verdrängt wurde, an einer anderen Adresse wiedergefunden werden kann.

Deshalb muß nach einem Taskwechsel oder einer Anforderung eines weiteren Blocks der vorher benutzte auf alle Fälle nochmal angefordert werden.

ISFILE (−− useraddr): In dieser Uservariable wird die aktuelle Datei gespeichert.

ISFILE@ (−− file): Liefert die aktuelle Datei (Isfile).

FROMFILE (−− useraddr): In dieser Variable kann man eine zweite Datei speichern, auf die man neben der Isfile auch Zugriff hat. CONVEY und COPY lesen von der hier gespeicherten Datei und kopieren in die Isfile.

PREV (−− addr): Zeiger auf eine verkettete Liste der Verwaltungsinformationen der Blockpuffer. PREV zeigt auf die Verwaltungsinformation des zuletzt angeforderten Blockes.

MEMORY (−−) (VS voc −− MEMORY): Vokabular für die Worte des Memory Managements (s. Kapitel 7.1).

BLOCKR/W (file pos len addr r/w −−): Deferred Word. Von der Datei *file* werden ab der Position *pos len* Bytes in den Puffer ab *addr* geschrieben oder von

diesem Puffer in die Datei gespeichert. Gelesen wird, wenn r/w=0, bei r/w=1 wird geschrieben.

DISKERR (error# string −−): Deferred Word. Gibt die Meldung *string* und die TOS-Fehlernummer *error#* (eventuell in Klartext gewandelt) aus.

(DISKERR (error# string −−): Hängt in DISKERR. Im Kernel wird die Fehlernummer als Dezimalzahl ausgegeben.

BACKUP (addr −−): Sichert den Puffer mit der Verwaltungsinformation *addr* auf Diskette zurück, wenn dessen Update-Flag gesetzt ist und löscht diese anschließend.

EMPTYBUF (addr −−): Entfernt den Puffer mit der Verwaltungsinformation *addr* aus dem Pufferspeicher. Wurde er vorher verändert, werden die Veränderungen nicht gespeichert.

UPDATE (−−): Setzt die Update-Flag des zuletzt angeforderten Blocks.

CORE? (blk file −− dataaddr / false): Sucht den Block *blk* in der Datei *file* im Puffer. Ist er vorhanden, wird die Datenadresse (die Pufferadresse des Inhalts) zurückgegeben, sonst *false*.

BLK/DRV (−− n): Deferred Word. Gibt die Anzahl tatsächlich vorhandener Blöcke im aktuellen Laufwerk bzw. die Länge der aktuellen Datei in Blöcken zurück.

CAPACITY (−− n): Gibt die Länge der aktuellen Datei in Blöcken zurück.

(BUFFER (blk file −− addr): Sucht die Adresse des Blocks *blk* in der Datei *file* im Puffer. Wird sie nicht gefunden, legt (BUFFER die Verwaltungsinformation für diesen Block an und ordnet ihm einen Puffer mit undefiniertem Inhalt zu. (BUFFER wird benutzt, wenn ein Block völlig neu geschrieben wird und auf die Information auf Massenspeicher verzichtet werden kann.

BUFFER (blk −− addr): Wie ISFILE@ (BUFFER. Sucht den Block *blk* der aktuellen Datei. Wird er nicht gefunden, so wird ein leerer Puffer (mit zufälligem Inhalt) angelegt.

(BLOCK (blk file −− addr): Fordert den Block *blk* der Datei *file* an. Steht er nicht im Puffer, wird eine neue Verwaltungsinformation angelegt und und der Block vom Massenspeicher geladen.

BLOCK (blk −− addr): Fordert den Block *blk* der aktuellen Datei an. Wie ISFILE@ BLOCK.

SAVE–BUFFERS (−−): Sichert alle veränderten Blöcke des Puffers, d. h. alle Blöcke, deren Update-Flag gesetzt ist.

EMPTY–BUFFERS (−−): Leert den Blockpuffer. Veränderte Blöcke werden nicht gesichert.

FLUSH (−−): Zusammenfassung von SAVE-BUFFERS EMPTY-BUFFERS. Leert den Blockpuffer und sichert alle veränderten Blöcke. Zudem werden alle Dateien geschlossen und die Handles damit ans TOS zurückgegeben.

28. File-Interface

Standard-FORTH kann nur direkt auf Massenspeicher zugreifen. TOS aber organisiert Massenspeicher in Dateien. Hier hat man den Vorteil der Gliederung. Außerdem kann man Dateien problemlos verlängern und ist nicht starr an bereits belegte Blöcke gebunden wie im Direktzugriff.

Um den Dateizugriff transparent zu ermöglichen, gibt es eine neue Uservariable, ISFILE. Sie enthält den Zeiger auf den File Control Block (FCB) der aktuellen Datei. Zeigt ISFILE auf Nil, so wird der Direktzugriff benutzt.

In einem FCB müssen folgende Daten gespeichert sein: Name, Länge und Handle der Datei und wieoft die Datei mit OPEN geöffnet wurde. Die FCBs sind in einer verketteten Liste zusammengehängt, zudem hat jeder eine eigene Nummer, anhand der er identifiziert werden kann, wenn das View-Field eines Wortes ausgewertet wird. Nummer und Link-Field sind statische Informationen, Name, Länge, Handle und Anzahl der OPENs sind dynamisch und können verändert werden.

Um Speicher zu sparen, werden nur die statischen Informationen direkt compiliert, die anderen werden im Memory Heap dynamisch angelegt, also erst, wenn sie wirklich benötigt werden. Als Name wird vorerst der Wortname des FCBs eingesetzt - dazu muß der natürlich auch vorhanden sein.

Genaueres über das File-Interface steht im Kapitel 6.

FILE–LINK (−− useraddr): Zeigt auf die verkettete Liste aller File Control Blocks (FCBs), die im System angelegt wurden, also alle bisher benutzten Dateien.

DOS (−−) (VS voc −− DOS): Vokabular, das die Basisbefehle für das Fileinterface und die Aufrufe des GEMDOS enthält.

!FCB? (file −−): Prüft nach, ob der FCB der Datei *file* korrekt angelegt ist, wenn nicht, wird er neu angelegt. Dateilänge, Handle und der Zähler für die Zahl der OPENs auf diese Datei werden auf 0 gesetzt, der Dateiname wird auf den Wortnamen gesetzt, unter dem der FCB angelegt ist.

!FCB? muß angewendet werden, wenn man auf einen FCB zugreifen will, der möglicherweise nicht geöffnet wurde, und man auf OPEN verzichten muß.

OPEN (−−): Öffnet die aktuelle Datei. Ist sie schon offen, wird der Zähler der OPEN-Befehle um eins erhöht.

CLOSE (−−): Schließt die aktuelle Datei. Tatsächlich geschlossen wird nur, wenn soviele CLOSE-Befehle auf die Datei angewendet wurden, wie vorher OPEN-Befehle. Alle Blöcke der Datei werden gesichert und aus dem Puffer gelöscht.

CLOSE! (−−): Schließt die aktuelle Datei auf alle Fälle, egal wie oft sie vorher geöffnet wurde.

ASSIGN (−−) ⟨*Filename*⟩: Schließt die aktuelle Datei und öffnet in ihrem FCB die Datei ⟨***Filename***⟩.

"USE (addr count −−):[⟨*Filename*⟩ (−−):]: Wählt die Datei mit dem Namen aus, der in *addr count* steht. Wurde diese Datei bereits angewählt, so wird der alte FCB benutzt, andernfalls ein neuer erzeugt.

USE (−−) ⟨*Filename*⟩:[⟨*Filename*⟩ (−−):]: Wählt die Datei ⟨***Filename***⟩ an. Ansonsten wie ÜSE.

FILE, (−−): Legt den statischen Teil einer FCB an (Linkfield, Zeiger auf den dynamisch verwalteten und Dateinummer).

FILE (−−) ⟨*Name*⟩:⟨*Name*⟩ (−−): Erzeugt einen FCB mit den Namen ⟨***Name***⟩. Der FCB ist vorerst leer.

DIRECT (−−): Setzt die Isfile auf 0 und damit auf Direktzugriff.

.FILE (fcb −−): Gibt den Namen der Datei *fcb* aus (Name ist nicht gleich Dateiname!).

FILE? (−−): Gibt den Namen der aktuellen Datei aus.

29. High Level Massenspeicherfunktionen

COPY (from to −−): Kopiert den Block *from* aus FROMFILE in den Block *to* der Isfile. Der alte Block *to* der Isfile wird überschrieben.

CONVEY ([blk1 blk2] [to.blk −−): Kopiert die Blöcke [*blk*1 bis einschließlich *blk*2] aus der FROMFILE ab Block [*to.blk* in die Isfile.

INDEX (from to −−): Gibt die Indexzeilen der Blöcke von *from* bis *to* der aktuellen Datei (mit Blocknummer) aus. INDEX kann mit ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ abgebrochen werden, mit einer anderen Taste unterbrochen und wieder fortgesetzt. Die Indexzeile ist die erste Zeile eines Screens.

30. Dictionary-Pflege

FORTH ist ein dynamisches Environment-System. Dieses hochtrabende Wort bedeutet, daß in FORTH ein einmal compiliertes Wort nicht unwiederruflich im System bleibt, sondern auch wieder gelöscht werden kann - ohne daß dazu das System verlassen und neu gestartet werden muß.

Diese Dictionary-Pflege beschränkt sich darauf, das alles ab einem bestimmten Wort vergessen werden kann, also alle seit diesem Zeitpunkt definierten Wörter. Einzelne Wörter können nicht zwischendrin „vergessen“ werden, dies ist in dem Stackcharakter des Dictionaries begründet, man kann hier nicht einfach Seiten „herausreißen“.

DP! (addr −−): Wie DP !. Allerdings wird die Relocater-Info zwischen dem alten und dem neuen HERE gelöscht. DP! dient zum Zurücksetzen vom DP und wird von FORGET und EMPTY benutzt.

REMOVE (dic symb thread −− dic symb): Hängt die Teile einer verketteten Liste aus, die vergessen werden sollen. *thread* ist der Listenzeiger, alle Wörter zwischen *dic* und *symb* sollen entfernt werden. *dic* ist der unterste Bereich und liegt im Dictionary, *symb* liegt im Heap.

CUSTOM–REMOVE (dic symb −− dic symb): Deferred Word. Hier kann man später eigene Wörter einhängen, die eigene Strukturen entfernen. Auch hier muß alles, was zwischen *dic* und *symb* liegt, entfernt werden.

CLEAR (−−): Löscht den Heap. Das Dictionary wird nicht berührt.

(FORGET (addr −−): *addr* ist entweder die niedrigste Adresse im Dictionary oder die höchste im Heap, die noch behalten werden soll. (FORGET sucht alle Wörter heraus, die danach definiert wurden und vergißt sie.

FORGET (−−) ⟨*Name*⟩: Vergißt ab ⟨***Name***⟩ einschließlich alle Wörter, die später definiert wurden. Abgebrochen wird, wenn nur Symbole im Heap vergessen werden sollen (Fehler „is Symbol!“) oder wenn das Wort im geschützten Bereich des Systems liegt („protected“).

EMPTY (−−): Löscht alles bis auf den geschützten Bereich.

SAVE (−−): Löscht den Heap und setzt alles bisher Definierte als geschützten Bereich. Nach dem Start ist nur der Systemteil, der sofort vorhanden ist, geschützt, man kann also nur vergessen, was man nach dem Start und vor einem SAVE definiert hat.

31. Ein/Ausgabe

FORTH benutzt zur Ein/Ausgabe das Konzept des virtuellen Terminals. Die Ein/Ausgabeeinheit versteht einige standardisierte Befehle. Damit erreicht man eine Unabhängigkeit vom tatsächlich verwendeten Gerät. Die Ausgabe kann auf einen Drucker oder einen Bildschirm erfolgen, in eine Datei umgeleitet oder über serielle Schnittstelle auf einen anderen Rechner übertragen werden. Genauso muß die Eingabe nicht über Tastatur erfolgen, sondern könnte auch über serielle Schnittstelle o. ä. laufen.

Die CFAs der gerätespezifischen Wörter sind für Input und Output jeweils in einem Array zusammengefaßt, die beiden Uservariablen INPUT und OUTPUT zeigen auf diese Arrays. Sie stehen dort in der Reihenfolge, in der sie hier aufgelistet sind.

OUTPUT: (−−) ⟨*Name*⟩ {⟨*Wort*⟩ }(13) [:⟨*Name*⟩ (−−): Erzeugt ein Outputfeld. Hinter dem Namen müssen die folgenden 13 gerätespezifischen Wörter in der hier aufgezählten Reihenfolge stehen. Abgeschlossen wird die Definition mit [. Bei dem Aufruf von ⟨*Name*⟩ wird die Ausgabe auf dieses Gerät umgeleitet, d. h. die PFA von ⟨*Name*⟩ wird in der Uservariablen OUTPUT gespeichert.

Beispiel: Das System-Outputfeld DISPLAY wurde so definiert:

```
Output: DISPLAY
STemit STcr STtype STdel STpage STat STat? STform
STcuron STcuroff STcurleft STcurrite STclrline [
```

EMIT (char −−): Gibt das Zeichen *char* aus.

CR (−−): Wagenrücklauf. Beginnt eine neue Zeile. Am unteren Rand des Bildschirms wird gescrollt, am Ende der Druckerseite wird ein neues Blatt eingezogen.

TYPE (addr count −−): Ascii-Dump. Gibt alle *count* Zeichen aus, die im Puffer ab *addr* gespeichert sind.

DEL (−−): Löscht das letzte Zeichen (überschreibt es mit einem Leerzeichen) und rückt den Cursor (Druckkopf) um eins nach links.

PAGE (−−): Löscht den Bildschirm oder zieht ein neues Blatt ein.

AT (row col −−): Positioniert den Cursor (Druckkopf) in der Zeile *row* und der Spalte *col*. Für die linke obere Ecke ist *row*=0 und *col*=0.

AT? (−− row col): Legt die Cursor/Druckkopfposition auf den Stack.

FORM (−− rows cols): Gibt das Format an. Die Seite hat *rows* Zeilen und *cols* Spalten. Die rechte untere Ecke liegt bei $rows - 1$ und $cols - 1$.

CURON (−−): Schaltet den Cursor ein (sofern vorhanden).

CUROFF (−−): Schaltet den Cursor aus (wenn vorhanden).

CURLEFT (−−): Rückt den Cursor um eins nach links, vom Anfang einer Zeile wird zum Ende der vorhergehenden gegangen.

CURRITE (−−): Rückt den Cursor um eins nach rechts, am Ende einer Zeile wird an den Anfang der nächsten gegangen.

CLRLINE (−−): Löscht die Zeile, in der der Cursor steht. Soll nur angewendet werden, wenn der Cursor am Anfang der Zeile steht ($col = 0$).

INPUT: (−−) ⟨*Name*⟩ (4)⟨*Wort*⟩ [:⟨*Name*⟩ (−−): Erzeugt ein Inputfeld. Hinter dem Namen müssen die folgenden 4 gerätespezifischen Wörter in der hier aufgezählten Reihenfolge stehen. Abgeschlossen wird die Definition von [. Bei dem Aufruf von ⟨*Name*⟩ wird die Eingabe von dieses Gerät angenommen, d. h. die PFA von ⟨*Name*⟩ wird in der Uservariablen INPUT gespeichert.

Beispiel: Das System-Inputfeld KEYBOARD wurde so definiert:

```
Input: KEYBOARD STkey STkey? STexpect STdecode [
```

KEY (−− key): Liest ein Zeichen aus dem Tastaturpuffer. Ist der Puffer leer, so wird auf einen Tastendruck gewartet. Der zurückgegebene 16-Bit-Wert enthält im Lowbyte den Asciicode der Taste, im Highbyte den Scancode der ST-Tastatur (der natürlich nicht standardisiert ist).

KEY? (−− flag): Prüft, ob eine Taste gedrückt wurde. Wenn ja, wird true zurückgegeben. Der Tastaturpuffer wird nicht beeinflußt.

EXPECT (addr len −−): Liest von der Tastatur in den Puffer ab *addr* maximal *len* Zeichen ein. Ein RET beendet EXPECT. Die tatsächlich eingelesene Länge wird in der Variablen SPAN zurückgegeben.

DECODE (addr pos1 key −− addr pos2): Dekodiert das Zeichen *key*. DECODE wird von EXPECT benutzt. *addr* ist der Anfang des Puffers, die tatsächliche Länge steht in SPAN, die maximale in MAXCHARS. *pos*1 ist die Cursorposition im Text, *pos*2 die neue Cursorposition.

STOP? (−− flag): Liefert true, wenn ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ gedrückt wurde. Bei einer anderen Taste wartet STOP? einen weiteren Tastendruck ab. Auch hier wird wieder true geliefert, wenn ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ gedrückt wurde. Wurde keine Taste gedrückt, wird false geliefert. STOP? dient zum Ab- und Unterbrechen von längeren Ausgaben wie bei WORDS oder INDEX.

ROW (−− row): Gibt die Zeile zurück, in der der Cursor steht.

COL (−− col): Gibt die Spalte zurück, in der der Cursor steht.

ROWS (−− rows): Gibt die Anzahl der Zeilen des Schirms zurück.

COLS (−− cols): Gibt die Anzahl der Spalten des Schirms zurück.

?CR (−−): Bricht mit CR um, wenn von der Cursorposition weniger als 16 Zeichen bis zum rechten Rand sind. ?CR soll verhindern, daß mitten im Wort umgebrochen wird.

STANDARDI/O (−−): Setzt die Ein/Ausgabestruktur auf die Anfangswerte nach Systemstart zurück.

PUSHI/O (−−): Sichert die alte Ein/Ausgabestruktur auf dem Returnstack, sie wird nach dem Verlassen des Wortes wieder zurückgesetzt.

32. Systemstart

bigFORTH wird wie ein normales GEM-Programm gestartet. Nach dem Start muß also aller überflüssiger Speicher zurückgegeben werden. Das eigentliche FORTH-System belegt den Speicher von FORTHSTART bis LIMIT. Der Supervisormodus des Prozessors wird eingeschaltet, damit sämtliche Systemadressen angesprochen werden können. Dann wird der Relocater aufgerufen. COLD initialisiert Userarea, Stack und Returnstack sowie die Vokabulare und Tasks.

Für den Memory Heap wird Speicherplatz belegt. Dabei wird der freie Speicher ermittelt, RESERVED Bytes für das TOS übriggelassen, mindestens aber ACCBUF Bytes belegt und wenn das auch nicht geht, eben der ganze freie Restspeicher.

Die Maus wird versteckt, der Bildschirm gelöscht und die Startmeldung ausgegeben. Soweit vorhanden, wird eine vom System übergebene Kommandozeile als FORTH-Zeile interpretiert. Der Editor versucht die Kommandozeile als Datei zu interpretieren, dazu darf sie aber kein Leerzeichen enthalten und muß mit .SCR enden. Die Kommandozeile muß als counted String und CR-0-terminiert übergeben werden. Damit die Kommandozeile nicht zweimal interpretiert wird, löscht das System das CR, das nicht mehr zum eigentlichen String gehört und ohnehin bedeutungslos ist.

RESERVED (−− n): *n* Bytes müssen für das System übrigbleiben. Default: $10000=64 KBytes.

ACCBUF (−− n): *n* Bytes müssen mindestens für den Memory Heap belegt werden. Default: $12000=72 KBytes.

FORTHSTART (−− addr): Startadresse des FORTH-Systems. $100 Bytes vor FORTH-START beginnt die Basepage.

LIMIT (−− addr): Hier endet das System. In bigFORTH ist der Bereich von FORTH-START bis LIMIT der eigentliche Teil des FORTH-Systems, also Dictionary, Stack, Heap, User Area, Returnstack und eventuell Relocater-Bitstring. Der Memory Heap liegt hinter LIMIT.

MALLOC (n / -1 −− addr/0 / free): Belegt einen n Bytes großen Block. Ist soviel Speicher nicht frei, wird 0 zurückgegeben. −1 MALLOC gibt die Länge des größten zusammenhängenden freien Blocks zurück. MALLOC wird beim Systemstart benötigt, um den Memory Heap anzulegen, ansonsten sollte man es nicht benutzen, da das bigFORTH-eigene Memory Management wesentlich leistungsfähiger ist und zudem kaum Systemspeicher frei ist.

MFREE (addr −− 0 / -error): Gibt den Block mit der Startadresse *addr* frei. Bei korrekter Ausführung wird 0 zurückgegeben, andernfalls eine TOS-Fehlernummer.

RELOZ (−− addr): Adresse der Relocater-Routine. Wird von SAVESYSTEM benutzt.

SAVE'SSP (−− addr): Hier wird der alte Supervisorstackpointer des Systems gesichert. Dahinter werden alten Vektoren der von bigFORTH verbogenen Traps gesichert (Bus Error, Address Error, Illegal Instruction, Division by Zero, Trapv, Trap #3).

RELINFO (−− addr / 0): Legt die Adresse der Relocater-Info auf den Stack. Ist keine Relocater-Info vorhanden, wird 0 zurückgegeben.

?ISPRG (−− flag): Liefert true, wenn bigFORTH als Programm gestartet wurde, false, wenn es ein Accessory ist.

COLD (−−): Kaltstart des Systems. Stack, Returnstack, Userarea und das Dictionary werden (soweit es geht) auf den Stand bei Systemstart zurückgesetzt. Der Bildschirm wird gelöscht und die Einschaltmeldung angezeigt.

'COLD (−−): Deferred Word. Wird von COLD nach erfolgreicher Installation aufgerufen. Der Bildschirm ist noch nicht gelöscht, für GEM-Programmierer:
Die Maus ist noch eingeschaltet. 'COLD wird zum Starten von eigenen Applikationen verwendet.

RESTART (−−): „Lauwarmstart“ des Systems. Stack und Returnstack werden initialisiert. Außerdem wird (QUIT in 'QUIT eingehängt und damit auf alle Fälle der Interpreter aufgerufen. Der Vocabulary Stack wird mit ONLYFORTH gesetzt. Warmstart kann man RESTART nicht nennen, da ein Warmstart mit ABORT, ABORT“ bzw. ERROR“ ausgeführt wird.

'RESTART (−−): Deferred Word. Wird von RESTART aufgerufen. Hier hängt man Erweiterungen des Systems ein, die initialisiert werden müssen. In 'RESTART ist das FORTH-System selbst vollständig initialisiert.

33. Verlassen des Systems

'BYE (−−): Deferred Word. Es wird von BYE direkt vor dem eigentlichen Programmende aufgerufen. Hier muß man sich einhängen, wenn man Systemvektoren verbogen hat und die Vektoren zurücksetzen, da es sonst höchstwahrscheinlich einen Absturz gibt.

BYE (−−): Verläßt das System. Ist bigFORTH als Accessory gestartet, kann es nicht beendet werden, dann zeigt BYE keine Wirkung. Im Gegensatz zu volksFORTH darf BYE in bigFORTH nicht neu definiert werden, da sich zwei andere Wörter darauf verlassen, daß es der ursprünglichen Definition entspricht: GOODBYE und BADBYE. Unvermeidbares muß man in 'BYE einhängen.

BADBYE (−−): Verläßt das System und übergibt an den aufrufenden Prozeß die letzte Fehlernummer, falls diese 0 war, eine −1 („allgemeiner Fehler“). Dadurch wird RELOCATE.PRG vom Mißerfolg der Compilation informiert.

34. ST-Interface

Der betriebssystemabhängige Teil vom FORTH-Kernel ist ziemlich klein. „Nur“ die Ein/Ausgaben und die Massenspeicherzugriffe sind natürlich prinzipiell systemabhängig und müssen definiert werden.

Die Zeicheneinein/ausgabe wird mit den BIOS-Funktionen des TOS erledigt. (BIOS= Basic Input/Output-System). Das angewählte Gerät dev ist eine Nummer, die folgendermaßen belegt ist:

0=Centronics (Drucker)
1=RS 232
2=Bildschirm/Tastatur
3=MIDI
4=Tastaturprozessor
5=Bildschirm direkt (keine Steuerzeichenauswertung)

BCONSTAT (dev −− flag): Gibt true zurück, wenn *dev* ein Zeichen senden kann. Bei *dev*=2 wird abgefragt, ob ein Zeichen im Tastaturpuffer ist.

BCOSTAT (dev −− flag): Gibt true zurück, wenn *dev* empfangsbereit ist. Der Bildschirm nimmt immer Zeichen entgegen. Bei BCOSTAT sind die Nummern von MIDI und Tastaturprozessor vertauscht, also ist 3 BCOSTAT der Tastaturprozessorstatus und 4 BCOSTAT der MIDI-Status.

BCONIN (dev −− char): Liest von *dev* ein Zeichen ein. Von der Tastatur wird auch der Scancode zurückgegeben (im selben Format wie von KEY).

BCONOUT (char dev −−): Gibt das Zeichen *char* auf dem Gerät *dev* aus.

#BS (−− $08): Steuerzeichen Backspace (Ein Zeichen zurück).

#CR (−− $0D): Steuerzeichen Carriage Return (Wagenrücklauf).

#LF (−− $0A): Steuerzeichen Line Feed (Zeilenvorschub).

#ESC (−− $1B): Steuerzeichen Escape.

CON! (char −−): Gibt das Zeichen *char* an den Bildschirm aus. Steuerzeichen werden interpretiert.

WRAP (−−): Schaltet den automatischen Umbruch am Zeilenende ein. Buchstaben, die übers Zeilenende hinausgehen, werden in der nächste Zeile ausgegeben.

STEMIT (char −−): Gibt *char* auf dem Bildschirm aus. Steuerzeichen werden auch ausgegeben, nicht interpretiert.

STCR (−−): Carriage Return.

STDEL (−−): Löscht das Zeichen vor dem Cursor.

STPAGE (−−): Löscht den Bildschirm.

STAT (row col −−): Setzt den Cursor auf Zeile *row* und Spalte *col*.

STAT? (−− row col): Gibt die Cursorposition zurück. Sie wird aus den negativen Line-A-Variablen ausgelesen (Variablen mit negativem Offset zu A_BASE, s. Literatur zu Line-A).

STFORM (−− rows cols): Gibt die Bildschirmgröße zurück. Auch hier wird aus negativen Line-A-Variablen ausgelesen.

STTYPE (addr len −−): Gibt *len* ab *addr* gespeicherte Zeichen aus. Dabei muß *len* ein 16-Bit-Wert sein.

STCURON (−−): Schaltet den Cursor ein.
STCUROFF (−−): Schaltet den Cursor aus.
STCURLEFT (−−): Cursor nach links.
STCURRITE (−−): Cursor nach rechts.
STCLRLINE (−−): Löscht die Zeile, auf der der Cursor steht.
DISPLAY (−−): Standard-Output in bigFORTH.
STKEY? (−− flag): Gibt true zurück, wenn der Tastaturpuffer Zeichen enthält.
GETKEY (−− key / false): Liest ein Zeichen aus dem Tastaturpuffer, wenn vorhanden, sonst wird *false* zurückgegeben.
STKEY (−− key): Liest ein Zeichen aus dem Tastaturpuffer. Ist der leer, wird gewartet.
STDECODE (addr pos1 key −− addr pos2): DECODE für den ST.
MAXCHARS (−− useraddr): Enthält die maximale Länge des Puffers von EXPECT.
STEXPECT (addr len −−): EXPECT für den ST.
KEYBOARD (−−): Standard-Input in bigFORTH.
B/BLK (−− $400): Konstante: Ein Block hat $400=1024 Bytes=1 KByte.
DRIVE (n −−): Schaltet auf Laufwerk *n*. „A:“ ist Dr 0, „B:“ Dr 1 usw.
>DRIVE (blk drv −− blk'): Rechnet aus *blk* und *drv* die absolute Blocknummer aus. Damit kann direkt auf das Laufwerk *drv* zugegriffen werden.
DRV? (blk −− drv): Gibt zurück, auf welchem Laufwerk *blk* zu finden ist.
DRVINIT (−−): Deferred Word. Initialisiert den Laufwerkszugriff.
STR/W (file pos len addr r/wf −−): Liest von der Datei *file* ab *poslen* Bytes in den Puffer ab *addr* oder schreibt daraus zurück. STR/W kann nur auf Dateien zugreifen, der Direktzugriff wird später dazugeladen.
A: (−−): Setzt das aktuelle Laufwerk auf A:.
B: (−−): Setzt das aktuelle Laufwerk auf B:.
C: (−−): Setzt das aktuelle Laufwerk auf C:.
D: (−−): Setzt das aktuelle Laufwerk auf D:.
E: (−−): Setzt das aktuelle Laufwerk auf E:.
F: (−−): Setzt das aktuelle Laufwerk auf F:.
>REL (addr −− n): Wie FORTHSTART -. Rechnet den Offset von *addr* zum Start des Systems aus.
FORTH.SCR (−−): Sourcedatei des Kernels, Datei mit der Nummer 1.
FORTH–83 (−−):: Letztes Wort des Kernels von volksFORTH-83. Tat dort nichts. In bigFORTH ist dieses Wort nicht mehr vorhanden, deshalb ist es hier hell dargestellt. Dieses Wort kann man benutzen, wenn man sich darauf verläßt, daß ein FORTH-83-kompatibles FORTH benutzt wird. Als erste ausgewertete Zeile im Loadscreen kann man schreiben:

```
\needs FORTH-83 .( Ihr System ist nicht 100% F83-kompatibel!) \\
```

Solche Sources lassen sich mit bigFORTH nicht mehr laden - sie müssen angepaßt werden. So leid es uns tut: Aufgrund der Optimierungen und den Schwierigkeiten, ein FORTH-System auf dem ST passabel relokatibel zu machen, ist das System zwar weitgehend F83-kompatibel, aber eben „nur“ ein Dialekt.

6 Der 486er–Assembler

1. Die Intel–Architektur

Als IBM seinen ersten PC entwickelte, lies man — offensichtlich der Meinung, PCs könnten nur in Garagen entwickelt werden — ein kleines Team den Prozessor eines Fremdherstellers verwenden. Zufällig war das Intel mit seinem 8086, der den enormen Vorteil hatte, mit nur 40 Beinchen in einen Standard–Sockel zu passen und in der Gestalt des 8088 prima mit billiger 8–Bit Peripherie (auch Speicher) auszukommen.

Eine Revolution war er damals nicht gerade. Zwar hatte man beim Aufbohren des 8085 auf 16 Bit doch eine neue Befehlsarchitektur entworfen, aber die Segmentierung des Speichers war sicher nicht die ultimative Lösung. Da es die Konkurrenz damals (Z80 unter CP/M, Apple, Atari und Commodore mit 6502) auch nicht anders machte, war das halb so schlimm.

Schlimmer war, daß zwar CP/M ausstarb und Apple, Atari und Commodore sich bessere Prozessoren (den 68 000) für ihre Produkte aussuchten, IBM aber weiter auf die x86–Architektur setzte und aus Kompatibilitätsgründen wohl auch dabei bleiben mußte.

Zwar beseitigte Intel in der Freude, unverschuldet zum Marktführer erhoben worden zu sein, nach und nach die schwerwiegendsten Mängel des 8086; zuerst der fehlende Schutz des Betriebssystem vor den Anwendungen, mit dem 386 auch die Limitierung der Segmentgröße und der Wortbreite auf 64K bzw. 16 Bit. Die folgende Leistungssteigerung über den 486 zum Pentium zeigt, daß auch in einem betagten Konzept noch gewaltige Reserven stecken und daß der dadurch eigentlich hohe Preis durch Marktmasse durchaus ausgeglichen werden kann.

Trotzdem bleiben konzeptionelle Mängel zurück: Die 8 Register sind einfach zu wenig, viele Befehle nutzen zudem manche dieser Register als Spezialregister. Außerdem gibt es nur einen Stack, ein Umstand, der für FORTH nicht gerade förderlich ist. bigFORTH wechselt deshalb beide Stacks immer wieder aus.

Das Beharren auf Kompatibilität hat auch wirksam verhindert, daß sich bei Zeiten ein Betriebssystem etablieren konnte, das den 386 auch unterstützt. Zuviel Umstand im Protected Mode und ein zu stark verändertes Programmiermodell machen den 386 zu einem anderen Prozessor, der lediglich einen Kompatibilitätsmodus hat — und fast nur in diesem benutzt wird.

bigFORTH enthält einen kompletten 486–Inline–Assembler. Mit diesem werden Code–Wörter (Primitives) definiert. Sie bilden im Kernel die Basis, daß FORTH überhaupt lauffähig ist.

Zudem ist optimaler Code nur in Assembler möglich. Für extrem zeitkritische Aufgaben muß deshalb auch in bigFORTH Assembler verwendet werden. Allerdings ist der Unterschied hier nicht so groß wie beim 68k–bigFORTH, da der x86er nicht den Vorteil vieler Register ausspielen kann. Ab dem 486er (und erst recht beim Pentium) sind Speicherzugriffe über den Stack ohnehin mit Registerzugriffen vergleichbar.

Dieses Manual kann nicht als Ersatz für ein 486–Manual dienen, zu viele Informationen wären dazu nötig. Eine große Reihe an geeigneteren Büchern, die in die Programmierung des Intel 486 einführen, finden Sie im Fachbuchhandel. Für den Profi ist Intels eigene Dokumentation (etwa [2]) unerläßlich.

2. Syntax

Der Inline–Assembler ist kein klassischer Assembler mit Parser, sondern lediglich eine Wortsammlung im Vokabular ASSEMBLER. Die Notation entspricht daher nicht dem von Intel vorgeschlagenen Standard, sondern der FORTH–üblichen UPN. Ebenso wird von der etwas unüblichen Registereihenfolge bei Intel (⟨Ziel⟩, ⟨Quelle⟩) abgewichen und stattdessen zuerst Quelle und dann Ziel angegeben.

Statt über Längendefinitionen in Labels oder der Angabe von Pointergrößen wird explizit geschaltet. Die Schalter wirken dabei nur auf den nächsten Befehl, ohne Angabe gilt .D für Langwortzugriff, bzw. .W im 86–Mode.

Ebenso unterscheidet sich die Notation von Adressierungsarten von Intels Vorgabe. Hier eine Umformungstabelle:

R	$\longrightarrow$	R	Register direkt
$[R]$	$\longrightarrow$	R `)`	Register indirekt
$[R_1 + R_2]$	$\longrightarrow$	$R_1\ R_2$ `I)`	Registersumme indirekt
$[R_1 + R_2 * Sc]$	$\longrightarrow$	$R_1\ R_2\ *Sc$ `I)`	Registersumme mit Scaling indirekt
$[Disp + R]$	$\longrightarrow$	$Disp\ R$ `D)`	Register mit Offset
$[Disp32 + R]$	$\longrightarrow$	$Disp32\ R$ `L)`	Register mit 32–Bit–Offset
$[Disp + R * Sc]$	$\longrightarrow$	$Disp\ R\ *Sc$ `I#)`	Skalierter Register mit Offset
$[Disp + R_1 + R_2]$	$\longrightarrow$	$Disp\ R_1\ R_2$ `DI)`	Registersumme mit Offset
$[Disp + R_1 + R_2 * Sc]$	$\longrightarrow$	$Disp\ R_1\ R_2\ *Sc$ `DI)`	Registersumme mit Scaling & Offset
$[Address]$	$\longrightarrow$	$Address$ `#)`	Adresse
$Value$	$\longrightarrow$	$Value$ `#`	Immediate

3. Verwaltungsbefehle

ASSEM486.SCR (−−): Aus dieser Datei wird der Assembler geladen. In Screen 1 steht der normale Loadscreen, in Screen 2 ein Loadscreen, der den kompletten Assembler in den Heap lädt, wodurch er mit CLEAR oder SAVE komplett gelöscht wird und daher in einer eigenen Applikation keinen Platz mehr belegt.

Die folgenden Befehle sind sowohl im Vokabular ASSEMBLER als auch im Vokabular FORTH enthalten:

ASSEMBLER (−−) (VS voc −− ASSEMBLER): In diesem Vokabular sind die Befehle des Assemblers definiert.

CODE (−−) (VS voc −− ASSEMBLER) ⟨*Name*⟩: Erzeugt einen Wortheader und ruft ASSEMBLER auf. Leitet damit die Definition eines Primitives ein. Ein Assemblerwort muß mit END–CODE beendet werden.

>LABEL (addr −−) ⟨*Name*⟩:⟨*Name*⟩ (−− addr): Erzeugt auf dem Heap ein Makro, das **addr** auf den Stack legt. HERE wird nicht verändert. >LABEL legt eine Adreßkonstante während des Assemblierens an.

LABEL (−−) (VS voc −− ASSEMBLER) ⟨*Name*⟩:⟨*Name*⟩ (−− addr): Erzeugt auf dem Heap ein Makro, das den beim Erzeugen aktuellen HERE auf den Stack legt. LABEL legt eine Adreßmarke an.

Die folgenden Wörter sind nur im Vokabular ASSEMBLER zugänglich:

END–CODE (−−) (VS voc ASSEMBLER −− voc voc): Beendet die Assemblerdefinition und schaltet zurück auf das alte Vokabular.

[F] (−−) (VS voc −− FORTH) immediate: Wie FORTH, jedoch immediate. Schaltet innerhalb einer FORTH–Definition auf das Vokabular FORTH.

[A] (−−) (VS voc −− ASSEMBLER) immediate: Analog zu [F], schaltet jedoch auf das Vokabular ASSEMBLER. Diese beiden Wörter dienen beim Programmieren von Assembler–Makros zum oft benötigten Umschalten zwischen beiden Vokabularen und schaffen Klarheit.

>CODES (−− addr): Enthält einen Zeiger, der auf eine Tabelle der zur Ablage von Maschinencode benötigten Wörter zeigt. Durch das Umschalten dieser Tabelle kann der Assembler auch vom Target–Compiler oder evtl. auch als Down–Assembler benutzt werden (je nach Fantasie des Anwenders).

Das Feld besteht aus folgenden Worten:
, HERE ALLOT C! +REL
wobei , einen elementaren Assembleropcode, also ein Byte compiliert — es steht also für C,. +REL setzt an HERE eine Adreßmarke.

NONRELOCATE (−−): Schaltet den Assembler auf Codeerzeugung im FORTH–System (default).

.386 (−−): Schaltet auf 32–Bit–Adressierung und 32–Bit Daten; es werden Befehle für den 32–Bit Protected Mode erzeugt.

.86 (−−): Schaltet auf 16–Bit–Adressierung und 16–Bit Daten; es werden Befehle für den Real Mode erzeugt (allerdings weiterhin 486er–Befehle, auch Langwortverarbeitung ist weiterhin möglich).

USER' (−− offset) ⟨*Uservariable*⟩ immediate: Liest den Offset der nachstehenden Uservariblen. Im Programm compiliert es den Wert als Literal, sonst legt es den Wert auf den Stack. Auf Uservariablen kann dann mit USER' ⟨***Uservariable***⟩ UP D) zugegriffen werden.

;CODE (0 −−) (VS voc −− ASSEMBLER) immediate restrict: Wie DOES>, der Compiler wird aber abgeschaltet und auf Assembler geschaltet. Die PFA liegt noch auf dem Returnstack.

4. Die Register

Die Register sind als Konstanten vordefiniert, deren Werte für den Benutzer nicht von Bedeutung sind. Die meisten Register sind unter bigFORTH vorbelegt: AX enthält den Top of Stack, BX den Schleifenzähler, CX und DX sind noch frei (üblicherweise wird DX für das zweite Argument verwendet). SI wird für den zweiten Stack (normalerweise den Returnstack) verwendet, DI für den Objektpointer. BP steht für den Userpointer und SP den ersten Stack (normalerweise den Datenstack).

AX CX DX BX SP BP SI DI (−− c): Wort- und Doppelwortregister. Anders als bei Intel wird nicht durch ein vorangestelltes „e“ ein 32–Bit–Befehl angezeigt, sondern durch den eingestellten Operationsmodus (mit .D).

AL CL DL BL AH CH DH BH (−− c): Byteregister. Hier wird der nächste Befehl auf alle Fälle als Byte–Operation assembliert.

[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] [BP] [BX] (−− c): Adreßregister für Wortadressen. Anders als im 32–Bit–Adressen–Modus können im 16–Bit–Adressen–Modus leider nur wenige Register und Registersummen als Adreßregister verwendet werden. Die Verwendung eines dieser Register schaltet automatisch auf Wort–Adressen um. Sie können nur mit dem Adreß–Distanz–Operator D) kombiniert werden.

ES CS SS DS FS GS (−− c): Segmentregister.

RP (−− SI): Alias auf SI, der im Stack–Modus als Returnstack verwendet wird.
OP (−− DI): Alias auf DI, der als Objektpointer dient.
UP (−− BP): Alias auf BP, der als User Pointer verwendet wird.
LOOPREG (−− BX): Alias auf BX, der als Schleifenzähler verwendet wird.
LOOPLIM (−− mem): Adressiert das oberste Element des Returnstack, auf dem das Schleifenende liegt.

CR (n −− $\mathbf{cr}_n$): Kontrollregister **n**. Da die Kontrollregister selten gebraucht werden, gibt es nicht für jedes der 8 Register eine eigene Konstante.
CR0 (−− $\mathbf{cr}_0$): Kontrollregister 0.
DR (n −− $\mathbf{dr}_n$): Debugregister **n**. Die 8 Debugregister werden ebenso selten gebraucht.
TR (n −− $\mathbf{tr}_n$): Testregister **n**. Analog zu CR und DR.

ST (n −− sp(n)): Stackelement **n** des Fließkommaprozessors als Quelle des Befehls.
<ST (n −− sp(n)): Stackelement **n** des Fließkommaprozessors als Ziel des Befehls.
STP (n −− sp(n)): Stackelement **n** des Fließkommaprozessors als Ziel des Befehls. Gleichzeitig wird der Top of Stack gepoppt.

5. Adressierungsarten

#) (addr −− mem): Segmentrelative Adressierung. Es wird auf den Wert zugegriffen, der an **addr** steht.
SEG) (disp seg −− sega): Absolute Adressierung. Es wird auf den Wert zugegriffen, der **disp** Bytes vom Segmentanfang des Segments **seg** steht.
 # (imm −−): Konstante unmittelbar. Der Wert der Konstante wird direkt als Quelloperand im Befehl benutzt. Wenn möglich, wird die Konstante auf ein Byte verkürzt.
L# (imm −−): Konstante unmittelbar. Eine Verkürzung wie bei # findet nicht statt. Hilft, wenn das Displacement später gepatcht werden muß.
A# (addr −−): Adreßkonstante unmittelbar. Eine Verkürzung findet ebenfalls nicht statt. Die assemblierte Adresse wird markiert.
A: (−−): Schaltet die Adreßmarkierung für die nächste Distanz ein.
A#) (addr −− mem): Segmentrelative Adressierung mit Markierung der Adresse für den Relocater. Entspricht A: #).
ASEG) (addr seg −− sega): Absolute Adressierung mit Markierung der Adresse. Entspricht A: SEG).
) (reg −− mem): Register indirekt. Es wird auf den Wert zugegriffen, auf das der momentane Inhalt von **reg** zeigt.
D) (disp reg −− mem): Register indirekt mit Displacement. Es wird auf den Wert zugegriffen, auf den die Summe des Inhalts von **reg** und **disp** zeigt. Kurze Displacements werden zu einem Byte verkürzt.
L) (disp32 reg −− mem): Register indirekt mit langem Displacement. Entspricht D), allerdings entfällt die Verkürzung auf ein Byte. Dient u. a. zum Patchen des Displacements
REL) (addr −− mem): IP–relative Adressierung (nur CALL und JMP). Zieladresse ist die Summe aus IP des nächsten Befehl und Displacement. Die Umrechnung von Adresse **addr** in das Displacement nimmt der Assembler vor.
I) (reg idx −− mem): Register indirekt mit Index. Zugegriffen wird auf den Wert, der an der Summe von Register **reg** und dem evtl. skalierten Register **idx** steht.

I#) (disp idx −− mem): Index indirekt mit Displacement. Zugegriffen wird auf den Wert, der an der Summe vom (skalierten) Register **reg** und **disp** steht.

DI) (disp reg idx −− mem): Register indirekt mit Displacement und Index. Zugegriffen wird auf den Wert, der an der Summe von **reg**, **disp** und (skaliertem) **idx** steht.

∗2 (reg −− idx): Scaling um den Faktor 2.

∗4 (reg −− idx): Scaling um den Faktor 4.

∗8 (reg −− idx): Scaling um den Faktor 8.

CS: DS: SS: ES: FS: GS: (−−): Segment override: Es wird auf das angegebene Segment zugegriffen. GS ist unter GO32 das DOS–Segment.

6. Längenangabe

Der 386er kann auf Bytes, Wörter (16 Bit) und Doppelwörter (32 Bit) zugreifen. Anders als in einem Intel–Assembler bestimmt in bigFORTH nicht Registername (außer bei Byte–Registern) oder Labeldeklaration die Zugriffslänge, sondern ein Schalter. Die Default–Länge ist im 386–Mode Doppelwort, im 86–Mode Wort. Ähnliches gilt für die Adreßberechnung. Allerdings gibt es hier kaum einen Grund, von der Vorgabe abzuweichen.

.B (−−): Der nächste Befehl ist ein Byte–Befehl

.W (−−): Der nächste Befehl ist ein Wort–Befehl (16 Bit)

.D (−−): Der nächste Befehl ist ein Doppelwort–Befehl (32 Bit)

.WA (−−): Adreßlängenschalter: Der nächste Befehl nutzt 16–Bit–Adressierung

.DA (−−): Adreßlängenschalter: Der nächste Befehl nutzt 32–Bit–Adressierung

7. Die Befehle

Als CISC–CPU besitzt der 486er eine Unmenge Befehle, die obendrein noch verschiedene Parameterversorgungen kennen. Im Kern ist die CPU eine $1\frac{1}{2}$–Adreß–CPU, es gibt also eine Registeradresse und eine Speicher- oder Registeradresse, die jeweils entweder als Quelle oder Ziel dienen. Unäre Operationen haben meist eine volle Adresse. Andererseits benutzen viele Befehle Spezialregister und sind nur stark eingeschränkt konfigurierbar. So wird CX oder CL als Zählregister verwendet, AX und DX bilden zusammen ein Doppel(Quad)wortregister; SI und DI werden für Stringoperationen verwendet und SP ist der Stackpointer.

Daneben gibt es noch unmittelbare Konstanten als Quelle, aber natürlich auch nicht immer und durchschaubar. Um die Übersicht etwas zu erleichtern, sind die Operationsarten als Stackeffekt angegeben:

reg	Register
r/m	Register oder Speicheradresse
mem	Speicheradresse
imm	Wert unmittelbar, belegt keinen Stackplatz
CL/imm	Wert unmittelbar, falls nicht vorhanden, wird CL genutzt
cdt	Control, Debug oder Test–Register
st	Stack als Source oder Destination oder Destination mit pop
st/m	Stack wie oben oder Speicheradresse

ADD (r/m reg / reg r/m / imm r/m −−): Addiert Quelle und Ziel und speichert das Ergebnis in Ziel

ADC (r/m reg / reg r/m / imm r/m −−): Addiert Quelle und Ziel mit Carry und speichert das Ergebnis in Ziel

SUB (r/m reg / reg r/m / imm r/m −−): Subtrahiert Quelle von Ziel und speichert das Ergebnis in Ziel

SBB (r/m reg / reg r/m / imm r/m −−): Subtrahiert Quelle von Ziel mit Borrow und speichert das Ergebnis in Ziel

OR (r/m reg / reg r/m / imm r/m −−): Bildet bitweises Oder von Quelle und Ziel

AND (r/m reg / reg r/m / imm r/m −−): Bildet bitweises And von Quelle und Ziel

XOR (r/m reg / reg r/m / imm r/m −−): Bildet bitweise Exklusiv–Oder von Quelle und Ziel

CMP (r/m reg / reg r/m / imm r/m −−): Vergleicht Quelle und Ziel. Entspricht einem SUB, ohne daß das Ergebnis geschrieben wird.

ROL (r/m CL/imm −−): Rotiert **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach links (in Richtung MSB)

ROR (r/m CL/imm −−): Rotiert **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach rechts (in Richtung LSB)

RCL (r/m CL/imm −−): Rotiert **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach links mit Carry (in Richtung MSB)

RCR (r/m CL/imm −−): Rotiert **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach rechts mit Carry (in Richtung LSB)

SHL (r/m CL/imm −−): Schiebt **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach links und füllt mit 0–Bits auf

SHR (r/m CL/imm −−): Schiebt **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach rechts und füllt mit 0–Bits auf

SAL (r/m CL/imm −−): Schiebt **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach links und füllt mit 0–Bits auf. Bei Überlauf wird das Overflow–Flag gesetzt.

SAR (r/m CL/imm −−): Schiebt **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach links und füllt mit dem MSB auf

Stringbefehle benutzen SI als Quelle und DI als Ziel. Nach der Operation werden abhängig von der Direction Flag im Statusregister die benutzten Register erhöht/erniedrigt. Stringbefehle können mit dem Präfix REP (bzw. REPE) CX mal wiederholt werden.

INS (−−): Input from Port to String. Liest einen Wert vom in DX angegebenen Port und schreibt ihn nach DI.

OUTS (−−): Output to Port from String. Schreibt einen Wert vom String aus SI in den in DX angegebenen Port.

MOVS (−−): Liest einen Wert von String an SI und schreibt ihn nach DI

CMPS (−−): Vergleicht die beiden Werte der Strings an SI und DI miteinander. Bricht mit Präfix REP bei Gleichheit ab, mit REPE bei Ungleichheit.

STOS (−−): Speichert den Wert in AL bzw. AX in den String in DI

LODS (−−): Lädt den Wert vom String in SI nach AL bzw. AX

SCAS (−−): Vergleicht den Wert vom String in DI mit dem Wert in AL bzw. AX. Bricht mit Präfix REP bei Gleichheit ab, mit Präfix REPE bei Ungleichheit.

REP (−−): Repeat Präfix: Der folgende Befehl (muß ein String–Befehl sein) wird CX mal wiederholt, bei vergleichenden String–Befehlen solange das Zero–Flag gelöscht ist (REPNE)

REPE (−−): Repeat while Equal–Präfix: Der folgende Befehl (muß ein vergleichender String–Befehl sein) wird höchstens CX mal wiederholt, solange das Zero–Flag gesetzt ist.

MOV (r/m reg / reg r/m / imm r/m / cdt reg / reg cdt −−): Lädt Quelle und speichert sie in Ziel

NOT (r/m −−): Invertiert **r/m** bitweise

NEG (r/m −−): Bildet das Zweierkomplement von **r/m**

MUL (r/m −−): Multipliziert **r/m** vorzeichenlos mit AL bzw. AX und schreibt das doppelt genaue Ergebnis in AX bzw. DX:AX (DX enthält den höherwertigen Part).

IMUL (r/m /imm reg −−): Multipliziert **r/m** mit **reg** nach **reg** oder **r/m** mit AL bzw. AX nach AX bzw. DX:AX oder **r/m** mit einer Konstante nach **reg**. Nur bei der zweiten Variante wird auch der vollständige Wertebereich der Multiplikation ausgenutzt.

DIV (r/m −−): Dividiert AX bzw. DX:AX vorzeichenlos durch **r/m**. DX enthält vorher den höherwertigen Part des Divisors, nachher den Rest, AX den Quotienten; bei Byte–Operationen entspricht AH DX und AL AX.

IDIV (r/m −−): Dividiert AX bzw. DX:AX vorzeichenbehaftet durch **r/m**. DX enthält vorher den höherwertigen Part des Divisors, nachher den Rest, AX den nach 0 gerundeten Quotienten. Bei Byte–Operationen gilt dasselbe wie bei DIV.

INC (r/m −−): Erhöht **r/m** um 1.

DEC (r/m −−): Erniedrigt **r/m** um 1.

TEST (r/m reg −−): Vergleicht **r/m** und **reg** bitweise. Entspricht einem AND, bei dem das Ergebnis nicht geschrieben wird.

SHLD (r/m reg CL/imm −−): Schiebt **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach links, schiebt dabei reg von rechts herein

SHRD (r/m reg CL/imm −−): Schiebt **r/m** um CL oder einen unmittelbar angegebenen Wert bitweise nach rechts, schiebt dabei reg von links herein

BT (r/m reg/imm −−): Bit Test. Testet Bit **reg/imm** im Bitstring **r/m**. Das getestete Bit wird ins Carry Flag kopiert.

BTS (r/m reg/imm −−): Bit Test and Set. Testet und setzt Bit **reg/imm** im Bitstring **r/m**. Das ursprüngliche Bit wird ins Carry Flag kopiert.

BTR (r/m reg/imm −−): Bit Test and Reset. Testet und löscht Bit **reg/imm** im Bitstring **r/m**.

BTC (r/m reg/imm −−): Bit Test and Complement. Testet und invertiert Bit **reg/imm** im Bitstring **r/m**.

PUSH (r/m −−): Pusht **r/m** auf den Stack

POP (r/m −−): Poppt **r/m** vom Stack

Der x86 hat vom 8080 ein paar nette BCD–Befehle geerbt, die so überflüssig sind, daß ich auf ein entsprechendes Manual verweisen kann.

DAA (−−): Decimal Adjust after Addition

DAS (−−): Decimal Adjust after Subtraction

AAA (−−): ASCII Adjust after Addition

AAS (−−): ASCII Adjust after Subtraction

AAM (/imm −−): ASCII Adjust after Multiply

AAD (/imm −−): ASCII Adjust before Division

XLAT (−−): Table lookup. Entspräche `mov AL,[(e)BX+(unsigned)AL]`, wenn es diesen Befehl gäbe.

PUSHA (−−): Pusht alle Register in der Reihenfolge der CPU–internen Numerierung, also AX, CX, DX, BX, SP, BP, SI, DI. Dabei hat SP den Wert vor dem Pushen von AX.

POPA (−−): Poppt alle Register in umgekehrter Reihenfolge wie bei PUSHA, mit Ausnahme von SP, der ignoriert wird

NOP (−−): No Operation. Tut nichts.

CBW (−−): Convert Byte Word. Setzt alle Bits in AH auf das Vorzeichen in AL.

CWD (−−): Convert Word Double. Setzt alle Bits in DX auf das Vorzeichenbit von AX.

FWAIT (−−): Wartet die Ausführung des letzten Coprozessorbefehls ab

WAIT (−−): Alias für FWAIT

PUSHF (−−): Pusht die Flags auf den Stack

POPF (−−): Poppt die Flags vom Stack

SAHF (−−): Store AH into Flags. Lädt die unteren 8 Bit des Flag–Registers mit AH.

LAHF (−−): Load AH from Flags. Lädt AH mit den unteren 8 Bit des Flag–Registers.

INT (imm −−): Löst Interrupt **imm** aus

INT3 (−−): Interrupt 3, wird als Breakpoint verwendet

INTO (−−): Interrupt on Overflow

IRET (−−): Interrupt Return

LOCK (−−): Lock Präfix: Der folgende Befehl greift in einem nicht unterbrechbaren Zyklus auf den Speicher zu

HLT (−−): Halt. Hält die CPU an.

CMC (−−): Complement Carry. Invertiert das Carry–Flag.

CLC (−−): Clear Carry. Löscht das Carry–Flag.

STC (−−): Set Carry. Setzt das Carry–Flag.

CLI (−−): Clear Interrupt Flag. Bis zum nächsten STI werden keine Interrupts mehr registriert.

STI (−−): Set Interrupt Flag. Nach dem nächsten Befehl werden wieder Interrupts angenommen, es sei denn, das ist wieder ein CLI.

CLD (−−): Clear Direction. String–Operationen erhöhen SI oder DI.

STD (−−): Set Direction. String–Operationen erniedrigen SI oder DI.

Die x86–Architektur bietet zwar alle gewohnten Programmflußkontrollen (Call, Return, bedingte und unbedingte Sprünge) an, der segmentierte Speicher verlangt aber zwei Varianten, near und far. Ein near call, jump oder return entspricht dem flachen, unsegmentierten Speicher und wird auch in bigFORTH fast immer ausreichen. Ein far call, jump oder return braucht zusätzlich den Segment–Deskriptor des Ziels als Angabe. Ein far call legt auch zuerst das Segment, dann den IP auf den Stack und es muß mit einem far ret zurückgesprungen werden.

Anders als im Intel–Assembler wird auch hier nicht beim Label angegeben, ob near oder far gesprungen wird, sondern im Befehl. Da bigFORTH selbst nicht segmentiert ist, reicht das auch aus. Die Unmenge an Protekt- und Segmentstrategien, die Intel erlaubt, werden vom GO32 zum Glück kaum genutzt.

RET (/imm −−): Rücksprung aus einer Subroutine. Addiert optional nach dem Holen der Rücksprungadresse noch **imm** (ein 16–Bit–Wert) zum Stackpointer und bereinigt damit den Stack (vor allem für Pascal–Programme wichtig).
RETF (/imm −−): Rücksprung aus einer Subroutine, die mit einem far call angesprungen wird (Aufrufer in einem anderen Segment). Ansonsten wie RET.
CALL (addr −−): Springt nach **addr** und legt dabei den IP der nächsten Instruktion auf den Stack
CALLF (seg −−): Far Call. Springt an die Segmentadresse. Verschiedene Protektstrategien kommen zum Einsatz, wie Taskwechsel, Stackwechsel, Call Gates etc. Eine Rücksprungadresse für RETF wird bereitgestellt.
JMP (addr −−): Springt nach **addr**
JMPF (seg −−): Far Jump. Je nach Protektstrategie wird eine Rücksprungadresse erzeugt oder nicht (sehr kompliziert).

Bedingte Sprünge sind die elementaren Befehle für den Programmfluß. Da jeder Prozessorhersteller seine eigenen Abkürzungen für Bedingungen verwendet, werden in bigFORTH neben den herstellerspezifischen Kürzel auch FORTH–artige Bedingungscodes angegeben; mit UPN–Syntax, versteht sich.

VS (−− c): Overflow set
VC (−− c): Overflow clear
U< (−− c): Vorzeichenlos kleiner. Carry set.
U>= (−− c): Vorzeichenlos größer oder gleich. Carry clear.
0= (−− c): Gleich 0. Zero set.
0<> (−− c): Ungleich 0. Zero clear.
U<= (−− c): Vorzeichenlos kleiner oder gleich. $C \vee Z$
U> (−− c): Vorzeichenlos größer $\overline{C} \wedge \overline{Z}$
0< (−− c): Kleiner 0. Negative set.
0>= (−− c): Größer oder gleich 0. Negative clear.
PS (−− c): Parity even. Parity set.
PC (−− c): Parity odd. Parity clear.
< (−− c): Kleiner. $\overline{Z} \wedge (V \neq N)$.
>= (−− c): Größer oder gleich. $Z \vee (V = N)$.
<= (−− c): Kleiner oder gleich. $Z \wedge (V \neq N)$.
> (−− c): Größer. $\overline{Z} \wedge (V = N)$.

O (−− c): Overflow (VS)
NO (−− c): No Overflow (VC)
B (−− c): Below (Carry set, U<)
NB (−− c): Not Below (Carry clear, U>=)
Z (−− c): Zero (0=)
NZ (−− c): Not Zero (0<>)
BE (−− c): Below or Equal (U<=)
NBE (−− c): Not Below or Equal (U>)
S (−− c): Sign (0<)
NS (−− c): No Sign (0>=)
PE (−− c): Parity Even (PS)
PO (−− c): Parity Odd (PC)
L (−− c): Less (<)
NL (−− c): Not Less (>=)
LE (−− c): Less or Equal (<=)

NLE (−− c): Not Less or Equal (>)

JMPIF (addr c −−): Springt nach **addr**, wenn die Bedingung **c** wahr ist
JO (addr −−): Springt bei Overflow
JNO (addr −−): Springt wenn kein Overflow
JB (addr −−): Springt wenn vorzeichenlos kleiner bzw. Carry/Borrow gesetzt ist.
JNB (addr −−): Springt wenn vorzeichenlos größer oder gleich
JZ (addr −−): Springt wenn Null
JNZ (addr −−): Springt wenn ungleich Null
JBE (addr −−): Springt wenn vorzeichenlos kleiner oder gleich
JNBE (addr −−): Springt wenn vorzeichenlos größer
JS (addr −−): Springt wenn negativ
JNS (addr −−): Springt wenn positiv
JPE (addr −−): Springt wenn Parity gerade
JPO (addr −−): Springt wenn Parity ungerade
JL (addr −−): Springt wenn kleiner
JNL (addr −−): Springt wenn größer oder gleich
JLE (addr −−): Springt wenn kleiner oder gleich
JNLE (addr −−): Springt wenn größer

LOOPNE (addr −−): Decrementiert CX und springt solange nach **addr**, bis CX entweder 0 geworden ist, oder das Zero–Flag gesetzt ist
LOOPE (addr −−): Decrementiert CX und springt solange nach **addr**, bis CX entweder 0 geworden ist, oder das Zero–Flag gelöscht ist
LOOP (addr −−): Decrementiert CX und springt solange nach **addr**, bis CX 0 geworden ist
JCXZ (addr −−): Springt nach **addr**, wenn CX 0 ist (jump if CX zero)

Modernere Programmiersprachen können nicht nur bedingt verzweigen, sondern auch mit Flags rechnen. Solche Flags erzeugen kann der Befehl SETcc, wobei cc für den Condition Code steht. Leider erzeugt der Befehl nicht 0 oder −1 in einem Wort–Register, wie es für FORTH gut wäre, sonder 0 oder 1 in einem Byte–Register. Empfehlenswerte kürzeste Sequenz, dem abzuhelfen: Das Flag zur invertierten Bedingung erzeugen, `1 #` *reg* `and` um zu erweitern und *reg* `dec` um aus 0 −1 und aus 1 0 zu machen.

SETIF (r/m c −−): Setzt das Byte an **r/m** auf 1, wenn die Bedingung **c** wahr ist, sonst auf 0
SETO (r/m −−): Setzt **r/m** bei Overflow
SETNO (r/m −−): Setzt **r/m** wenn kein Overflow
SETB (r/m −−): Setzt **r/m** wenn vorzeichenlos kleiner bzw. Carry/Borrow gesetzt ist.
SETNB (r/m −−): Setzt **r/m** wenn vorzeichenlos größer oder gleich
SETE (r/m −−): Setzt **r/m** wenn Null
SETNE (r/m −−): Setzt **r/m** wenn ungleich Null
SETNA (r/m −−): Setzt **r/m** wenn vorzeichenlos kleiner oder gleich
SETA (r/m −−): Setzt **r/m** wenn vorzeichenlos größer
SETS (r/m −−): Setzt **r/m** wenn negativ
SETNS (r/m −−): Setzt **r/m** wenn positiv
SETPE (r/m −−): Setzt **r/m** wenn Parity gerade
SETPO (r/m −−): Setzt **r/m** wenn Parity ungerade
SETL (r/m −−): Setzt **r/m** wenn kleiner

SETGE (r/m −−): Setzt **r/m** wenn größer oder gleich
SETLE (r/m −−): Setzt **r/m** wenn kleiner oder gleich
SETG (r/m −−): Setzt **r/m** wenn größer

XCHG (r/m reg / reg r/m −−): Vertauscht den Inhalt von **r/m** und **reg**
MOVSX (r/m reg −−): Erweitert **r/m** vorzeichenbehaftet und schreibt das Ergebnis nach **reg**
MOVZX (r/m reg −−): Erweitert **r/m** vorzeichenlos und schreibt das Ergebnis nach **reg**
ENTER (imm imm8 −−): Legt ein Stackframe der Größe **imm** an. Dabei werden aus dem alten Stackframe optional **imm8** Pointer zu anderen Stackframes kopiert. Statt `0 ENTER` sollte man lieber `BP push SP BP mov imm # SP add` verwenden.
LEAVE (−−): Bereinigt ein Stackframe
ARPL (reg r/m −−): Paßt das RPL–Feld eines Selektors in **r/m** an das in **reg** an. Dieser Befehl ist nur für segmentorientierte Systemsoftware nötig.
BOUND (mem reg −−): Stellt fest, ob **reg** innerhalb von [**mem**] und [**mem**+size] liegt. Wenn nicht, wird Interrupt 5 ausgelöst.
BSF (r/m reg −−): Sucht in **r/m** vom LSB nach dem ersten gesetzten Bit und schreibt dessen Nummer nach **reg**
BSR (r/m reg −−): Sucht in **r/m** vom MSB nach dem ersten gesetzten Bit und schreibt dessen Nummer nach **reg**
CLTS (−−): Löscht die Taskswitch–Flag in CR0. Diese Flag erlaubt es, nach einem Taskwechsel bei Bedarf die Fließkommaregister zu sichern; danach muß mit CLTS die Flag gelöscht werden.

Die folgenden Befehle stehen nur auf dem 486er zur Verfügung, sie sind auf dem 386er nicht implementiert:

INVD (−−): Invalidate Cache. Markiert alle Einträge des internen Caches als ungültig und signalisiert externen Caches, sich ebenfalls zu löschen.
WBINVD (−−): Write Back and Invalidate Cache. Analog wie INVD, bewegt aber Write–Back–Caches dazu, den Inhalt vorher zurückzuschreiben.
CMPXCHG (reg r/m −−): Vergleicht AL bzw. AX mit **r/m**. Wenn beide gleich sind, wird **reg** nach **r/m** gespeichert, ansonsten **r/m** nach AL bzw. AX.
BSWAP (reg −−): Konvertiert einen 32–Bit–Wert im Register **reg** von little/big endian nach big/little endian
XADD (r/m reg −−): Addiert **r/m** und **reg** zusammen und schreibt das Ergebnis nach **r/m**. Der ursprüngliche Wert von **r/m** wird nach **reg** geladen.

Die weiteren Befehle stehen auch auf dem 386er zur Verfügung:

IN (/imm −−): Liest vom Port in DX oder **imm** nach AL bzw. AX
OUT (/imm −−): Schreibt AL bzw. AX auf den Port in DX bzw. in **imm**

SLDT (r/m −−): Store Local Descriptor Table Register. Speichert den LDTR in den 16–Bit–Wert **r/m**.
LLDT (r/m −−): Load Local Descriptor Table Register. Lädt den LDTR aus **r/m**.
STR (r/m −−): Store Task Register. Speichert das Task Register in den 16–Bit–Wert **r/m**.
LTR (r/m −−): Load Task Register. Lädt das Task Register aus **r/m**.
VERR (r/m −−): Testet, ob das Segment in **r/m** lesbar ist und setzt das Zero–Flag, wenn ja

VERW (r/m −−): Testet, ob das Segment in **r/m** beschreibbar ist und setzt das Zero–Flag, wenn ja
SGDT (r/m −−): Store Global Descriptor Table Register. Schreibt den GDTR nach **r/m**.
LGDT (r/m −−): Load Global Descriptor Table Register. Lädt den GDTR von **r/m**.
SIDT (r/m −−): Store Interrupt Descriptor Table Register. Schreibt den IDTR nach **r/m**.
LIDT (r/m −−): Load Interrupt Descriptor Table Register. Lädt den IDTR nach **r/m**.
SMSW (r/m −−): Store Machine Status Word. Speichert die untere Hälfte des CR0 nach **r/m** und ist nur zur Rückwärtskompatibilität zum 80286 vorhanden.
LMSW (r/m −−): Load MSW. Lädt **r/m** in die untere Hälfte von CR0.
INVLPG (mem −−): Invalidate Page. Streicht einen Eintrag aus der TLB, wenn **m** in dieser Seite liegt.
LAR (r/m reg −−): Lädt Zugriffsrechte vom Selektor an **r/m** nach **reg**. Führt dabei im wesentlichen eine Ausmaskierung von Adreßteilen durch.

LEA (mem reg −−): Lädt die Adresse **mem** in das Register **reg**
LDS (mem reg −−): Lädt die segmentrelative Adresse **mem** nach **reg** und DS
LES (mem reg −−): Lädt die segmentrelative Adresse **mem** nach **reg** und ES
LSS (mem reg −−): Lädt die segmentrelative Adresse **mem** nach **reg** und SS
LFS (mem reg −−): Lädt die segmentrelative Adresse **mem** nach **reg** und FS
LGS (mem reg −−): Lädt die segmentrelative Adresse **mem** nach **reg** und GS

8. Die Befehle der Fließkommaeinheit

Neben der Integer–CPU gibt es für die x86–Architektur auch eine Fließkommaeinheit. Bis zum 386 ist das ein eigener Chip, der 387. Ab dem 486 ist dieser Coprozessor in den Prozessorchip gewandert (außer beim 486SX). Trotzdem bleibt es vom Konzept her ein Coprozessor mit einem eigenen Registersatz, den die CPU mit Befehlen und Speicherzugriffen versorgt.

Der Coprozessor verwaltet seine 8 Register als Stack und ist damit ausnahmsweise für die Implementierung einer FORTH–Fließkommabibliothek gut geeignet. Nur falls sein interner Stack überläuft, verhält er sich weniger kooperativ. Trotzdem reichen die 8 Stackplätze schon ganz gut für die meisten praktischen Zwecke aus.

Ausgezeichnet ist auch die Genauigkeit der Operationen: 80 Bit ist eine Fließkommazahl lang, also IEEE–extended — und dies ist nicht nur die interne Darstellung; man kann diese Zahlen auch abspeichern und laden. Da die ganze FPU dem IEEE–854–Standard genügt, bleiben kaum Wünsche offen, allenfalls die Zahl der Register ist wieder mal etwas zu knapp, aber das ist man von Intel ja schon gewohnt.

Die Operandengrößen beim Speicherzugriff sind nicht Prefixes, wie bei den Integerbefehlen, sondern Schalter, die ihren Zustand behalten. Sie spielen nur bei den Lade– und Speicher–Operationen eine Rolle.

.FS (−−): Short Float: 1 Bit Vorzeichen, 8 Bit Exponent und 23 Bit Mantisse
.FL (−−): Long Float: 1 Bit Vorzeichen, 13 Bit Exponent und 52 Bit Mantisse
.FX (−−): Extended Float: 1 Bit Vorzeichen, 15 Bit Exponent und 64 Bit Mantisse (mit immer gesetztem 1.–Bit)
.FW (−−): Word: 16 Bit Integer

.FD (−−): Double Word: 32 Bit Integer
.FQ (−−): Quad Word: 64 Bit Integer

Da die Fließkommaoperationen auf den Stack operieren, der auch von bigFORTH als Fließkommastack verwendet wird, sind ihre Laufzeiteffekte auch als Stackeffekt angegeben.

FNOP (−−) (FS −−): Fließkomma NOP
FCHS (−−) (FS f −− −f): Ändert das Vorzeichen
FABS (−−) (FS f −− |f|): Bildet den Betrag von **f**
FTST (−−) (FS f −− f): Vergleicht **f** mit 0.0 und schreibt das Ergebnis in das Statusregister
FXAM (−−) (FS f −− f): Findet den Typ von **f** heraus: C3, C2 und C0 der Reihe nach: Unsupported, NaN, Normal, Infinity, Zero, Empty und Denormal. C3, C2, C0 = 111 ist nicht angegeben.
FLD1 (−−) (FS −− 1.0): Lädt 1.0 auf den Fließkommastack
FLDL2T (−−) (FS −− lb(10)): Lädt $\log_2 10$ auf den Fließkommastack
FLDL2E (−−) (FS −− lb(e)): Lädt $\log_2 e$ auf den Fließkommastack
FLDPI (−−) (FS −− π): Lädt π auf den Fließkommastack
FLDLG2 (−−) (FS −− lg(2)): Lädt $\log_{10} 2$ auf den Fließkommastack
FLDLN2 (−−) (FS −− ln(2)): Lädt $\log_e 2$ auf den Fließkommastack
FLDZ (−−) (FS −− 0.0): Lädt 0.0 auf den Fließkommastack
F2XM1 (−−) (FS f −− $2^{\mathbf{f}}-1$): Berechnet $2^{\mathbf{f}}-1$. Dies erlaubt eine genaue Berechnung von e^x auch in der Umgebung von 1. Allerdings muß zuerst x auf den 2er–Exponent abgeglichen (also mit $\log_2 e$ multipliziert) werden.
FYL2X (−−) (FS y x −− y ∗ $\log_2$ x): Berechnet $\mathbf{y} * \log_2 \mathbf{x}$, also den generellen Logarithmus.
FPTAN (−−) (FS f −− tan f 1.0): Berechnet den Tangens von **f**. Die 1.0 wird anschließend auf den Stack gepusht, um Berechnungen wie z. B. den Cotangens zu erleichtern — und aus Kompatibilitätsgründen. Insbesondere ist FPTAN die inverse Operation von FPATAN.
FPATAN (−−) (FS x y −− tan $\frac{\mathbf{x}}{\mathbf{y}}$): Berechnet den Tangens von $\frac{\mathbf{x}}{\mathbf{y}}$. Die zusätzliche Division erlaubt zudem noch die einfache Berechnung anderer trigonometrischer Umkehrfunktionen, wie z. B. Arcus Sinus: $\sin^{-1} x = \tan^{-1} \frac{x}{\sqrt{1-x^2}}$. Außerdem ist FPATAN auch die Umkehrfunktion von FSINCOS.
FXTRACT (−−) (FS f −− e s): Teilt **f** in seinen Exponenten **e** und seine Signifikant **s** inclusive Vorzeichen
FPREM1 (−−) (FS x y −− x y%x): Berechnet den partiellen Rest der Division $\frac{\mathbf{y}}{\mathbf{x}}$. Dieser ist betragskleiner als die Hälfte des Betrags des Dividends. Dabei wird iterativ vorgegangen. Bei jedem Schritt wird der Exponent von **y** höchstens um 64 reduziert. Wenn die Funktion erfolgreich beendet ist, wird C2 gelöscht und in C3, C1 und C0 die letzten 3 Bits des Quotienten gespeichert, ansonsten ist C2 = 1.
FPREM (−−) (FS x y −− x y%x): Berechnet den parziellen Modulo der Division $\frac{\mathbf{y}}{\mathbf{x}}$. Dieser hat dasselbe Vorzeichen wie der Dividend und ist garantiert betragskleiner als der Dividend. Ansonsten wird wie FPREM1 vorgegangen.
FDECSTP (−−) (FS f0 .. f7 −− f7 f0 .. f6): Rotiert den Fließkommastack in Richtung Stackboden
FINCSTP (−−) (FS f0 .. f7 −− f1 .. f7 f0): Rotiert den Fließkommastack in Richtung Top of Stack

FYL2XP1 (−−) (FS y x −− y∗log$_2$x+1): Berechnet den generellen Logarithmus von **x**+1 und erreicht damit auch in der Gegend um 0 eine hohe Genauigkeit, ähnlich F2XM1

FSQRT (−−) (FS f −− $\sqrt{f}$): Zieht die Wurzel aus **f**

FSINCOS (−−) (FS f −− sin f cos f): Berechnet Sinus und Cosinus von **f**

FRNDINT (−−) (FS f −− i): Rundet **f** zu einem ganzzahligen Wert entsprechend dem RC–Feld im FPU Control Word.

FSCALE (−−) (FS e s −− s∗2^e): Skaliert **s** um den Exponenten **e** und ist damit die Umkehrfunktion von FXTRACT. FSCALE bietet einen schnelle Multiplikation oder Division einer ganzzahligen Potenz von 2 an.

FSIN (−−) (FS f −− sin f): Berechnet den Sinus von **f**

FCOS (−−) (FS f −− cos f): Berechnet den Cosinus von **f**

FADD (st/m −−) (FS fn .. f1 −− fn+f1 .. f2 / fn+f1 .. f1 / fn .. f1+fn / fn .. f1+[r/m]): Addiert zwei Fließkommazahlen auf dem Stack oder eine Fließkommazahl aus dem Hauptspeicher zum Top of Stack.

FMUL (st/m −−) (FS fn .. f1 −− fn∗f1 .. f2 / fn∗f1 .. f1 / fn .. f1∗fn / fn .. f1∗[r/m]): Multipliziert zwei Fließkommazahlen auf dem Stack oder eine Fließkommazahl aus dem Hauptspeicher mit dem Top of Stack.

FCOM (st/m −−) (FS fn .. f1 −− fn .. f1 / fn .. f2): Vergleicht zwei Fließkommazahlen auf dem Stack oder eine Fließkommazahl aus dem Hauptspeicher mit dem Top of Stack.

FCOMP (st/m −−) (FS fn .. f1 −− fn .. f2 / fn .. f3): Vergleicht zwei Fließkommazahlen auf dem Stack oder eine Fließkommazahl aus dem Hauptspeicher mit dem Top of Stack und entfernt anschließend den Top of Stack.

FSUB (st/m −−) (FS fn .. f1 −− f1−fn .. f2 / f1−fn .. f1 / fn .. fn−f1 / fn .. [r/m]−f1): Subtrahiert zwei Fließkommazahlen auf dem Stack oder den Top of Stack von einer Fließkommazahl aus dem Hauptspeicher.

FSUBR (st/m −−) (FS fn .. f1 −− fn−f1 .. f2 / fn−f1 .. f1 / fn .. f1−fn / fn .. f1−[r/m]): Subtrahiert zwei Fließkommazahlen auf dem Stack oder eine Fließkommazahl aus dem Hauptspeicher vom Top of Stack. Das Ergebnis entspricht dem negierten von FSUB.

FDIV (st/m −−) (FS fn .. f1 −− f1/fn .. f2 / f1/fn .. f1 / fn .. fn/f1 / fn .. [r/m]/f1): Dividiert zwei Fließkommazahlen auf dem Stack oder eine Fließkommazahl aus dem Hauptspeicher durch den Top of Stack.

FDIVR (st/m −−) (FS fn .. f1 −− fn/f1 .. f2 / fn/f1 .. f1 / fn .. f1/fn / fn .. f1/[r/m]): Dividiert zwei Fließkommazahlen auf dem Stack oder den Top of Stack durch eine Fließkommazahl aus dem Hauptspeicher. Das Ergebnis ist reziprok zu dem von FDIV.

FCOMPP (−−) (FS f1 f2 −−): Vergleicht die obersten beiden Fließkommazahlen auf dem Stack und entfernt sie. Entspricht 1 STP FCOMP.

FBLD (mem −−) (FS −− f): Lädt eine BCD–codierte Fließkommazahl von **mem** auf den Stack

FBSTP (mem −−) (FS f −−): Speichert eine Fließkommazahl BCD–codiert an **mem** ab

FFREE (st −−) (FS f1 .. fi .. fn −− f1 .. empty .. fn): Setzt den "tag" von **st** auf leer. Damit kann auf **st** nicht mehr zugegriffen werden.

FSAVE (mem −−) (FS f1 .. fn −−): Sichert den akutellen Status der FPU in den 108 Byte (PM, im Real Mode 94 Byte) ab **mem** und reinitialisiert die FPU

FRSTOR (mem −−) (FS −− f1 .. fn): Liest den an **mem** von FSAVE abgespeicherten Status wieder in die FPU ein
FINIT (−−) (FS −−): Initialisiert die FPU
FXCH (st −−) (FS fn .. f1 −− f1 .. fn): Tauscht **st** und den Top of Stack aus
FSTENV (mem −−) (FS −−): Speichert das Environment, also Status–Wort, Kontroll–Wort, tag–Wort und die Error–Pointer in einem 28 (PM) bzw. 14 (RM) Byte großen Feld ab
FLDENV (mem −−) (FS −−): Lädt das mit FSTENV abgespeicherte Environment von **mem** zurück in die FPU
FSTCW (mem −−) (FS −−): Speichert das Kontroll–Wort (2 Byte) an **mem** ab
FLDCW (mem −−) (FS −−): Lädt das Kontroll–Wort von **mem**
FUCOM (st −−) (FS fn .. f1 −− fn .. f1 / fn .. f2): Vergleicht **st** mit dem Top of Stack und setzt die Bedingungsbits wie FCOM. Wenn ein Operand ein QNaN ist, werden C0, C2 und C3 auf 1 gesetzt.
FUCOMPP (−−) (FS f2 f1 −−): Vergleicht und popt die oberen beiden Werte des Stacks wie FUCOM
FNCLEX (−−) (FS −−): Löscht die Floating–Point exception Flags ohne auf Fehler zu prüfen
FCLEX (−−) (FS −−): Wie FNCLEX, prüft aber zuerst auf Fehler
FSTSW (AX/m −−) (FS −−): Speichert das Statuswort in AX oder **m**
FLD (st/m −−) (FS −− f): Lädt eine Fließkommazahl **f** von der Stackposition **st** oder von **mem**
FST (st/m −−) (FS f −− f): Kopiert die Fließkommazahl **f** nach **st** oder **m**
FSTP (st/m −−) (FS f −−): Speichert die Fließkommazahl **f** nach **st** oder **m** und popt sie

9. Conditionals

Assembler müssen nicht unstrukturiert sein. Neben den Labels und Sprüngen, die mehr an einen konventionellen Assembler erinnern, gibt es auch die üblichen FORTH–Kontrollstrukturen. Sie erlauben auch Vorwärtssprünge, die bei einem Einpassassembler nicht so leicht zu realisieren und mit Labels deshalb nicht möglich sind. Als "Flags" werden die Bedingungscodes des Prozessors verwendet. Es müssen also bei Vergleichen auch noch die entsprechenden Vergleichsbefehle, bei Tests ein Vergleich mit 0 oder ein Rx Rx TEST erfolgen.

IF (cond −− addr): Springt hinter ELSE bzw. THEN, wenn **cond** true ist. IF kann maximal über 128 Bytes springen.
THEN (addr −−): Löst eine Referenz von IF auf
AHEAD (−− addr): Springt unbedingt zum nächsten THEN bzw. ELSE
ELSE (addr −− addr'): Löst ein IF auf und assembliert ein AHEAD
BEGIN (−− addr): Legt HERE auf den Stack
DO (−− addr): Legt HERE auf den Stack und bereitet damit alles für ein LOOP/-LOOPE/LOOPNE vor, das zu DO zurückspringt
WHILE (addr cond −− addr' addr): Springt hinter REPEAT, wenn **cond** nicht erfüllt ist
UNTIL (addr cond −−): Springt solange zurück nach **addr**, solange **cond** nicht erfüllt ist
AGAIN (addr −−): Springt zurück nach **addr**

REPEAT (addr' addr −−): Springt zurück nach **addr** und löst einen WHILE–Sprung and **addr'** auf

?DO (−− addr' addr): Legt einen JCXZ an, der mit einem THEN hinter der zugehörigen LOOP–Anweisung aufgelöst wird. Damit wird die Schleife übersprungen, wenn sie nie augeführt werden muß.

BUT (addr' addr −− addr addr'): Vertauscht die oberen beiden Kontroll–Adressen

YET (addr −− addr addr): Verdoppelt die oberste Kontroll–Adresse

MAKEFLAG (cond −−): Setzt AX auf TRUE, wenn **cond** erfüllt ist, auf FALSE sonst

;C: (−−) (VS voc ASSEMBER −− voc voc): Geht vom Assembler in Hochsprachdefinition über

>C: (−−) (VS voc – ASSEMBLER) immediate: Geht von Hochsprachdefinition in Assembler über

R: (−−): Schaltet auf Return–Stack–Modus um. Im SP steht der Returnstack, in SI der Datenstack.

S: (−−): Schaltet auf Stack–Modus (default) um. Im SP steht der Datenstack, im SI der Returnstack.

:R (−−): Setzt auf Return–Stack–Modus. Die Stacks werden nicht ausgetaucht, :R sagt nur dem Assembler, in welchem Modus man gerade ist.

:S (−−): Setzt auf Stack–Modus. Ansonsten analog zu :R.

NEXT (−−): Makro zum Beenden eines Primitives. NEXT sichert den Modus, schaltet auf Return–Stack–Modus und assembliert ein RET.

7 Das File-Interface GEMDOS-, BIOS- und XBIOS-Library

1. Interna

Der Kern des File-Interfaces ist schon im Kernel enthalten. Im Kapitel 4 wurden die wichtigsten Worte dazu erklärt. Natürlich beschränkt man sich im Kernel auf die wesentlichen Funktionen, eine komfortablere Arbeitsumgebung kann später bei Bedarf nachgeladen werden. Diese findet man in FILEINT.SCR.

bigFORTH bietet Möglichkeiten, Dateien zu erzeugen, zu verlängern und zu löschen. Die Ordnerverwaltung des TOS wird unterstützt, Ordner können gewechselt, erzeugt und gelöscht werden. Zudem werden Environment-Pathes unterstützt, in denen Dateien gesucht werden, die im aktuellen Verzeichnis nicht gefunden wurden.

Basisstruktur des File-Interfaces ist der File Control Block (FCB), der die Dateien beschreibt. Dieser FCB ist zweigeteilt. Der statische Teil ist ein FORTH-Wort, von FILE definiert. Beim Aufruf wird diese Datei zur Isfile, zur aktuellen Datei, auf die zugegriffen werden kann.

Der dynamische Teil wird bei Bedarf im Heap angelegt. Da hier auch der Dateiname und gegebenenfalls der komplette Pfad steht, kann man die Länge des Bereichs nicht genau vorhersehen. Eine statische Reservierung wäre entweder Platzverschwendung oder zu klein. Deshalb ist es angebracht, diesen Teil in den Heap zu legen.

FCB-Struktur (Body eines mit FILE definierten Wortes):

|Link Field|Pointer Field|Number Field (16 Bit)|

Das Pointer Field zeigt auf folgende Struktur:

|Size (32 Bit)|Handle (16 Bit)|Open# (16 Bit)|Name: |0-Byte|

Nun noch ein paar Details zur Dateiverwaltung des TOS. Das TOS betrachtet jedes Laufwerk als eigenes Medium, auch die Partitions einer Festplatte. 16 solcher Medien kann es verwalten. Jedes hat ein Wurzelverzeichnis. Unterverzeichnisse kann man in sogenannten “Ordnern” (Directories) anlegen. Datei- und Ordnernamen dürfen höchstens 8 Buchstaben und einen durch Punkt vom Namen abgetrennten Suffix (maximal 3 Buchstaben) haben.

Dateinamen müssen folgende Syntax aufweisen:

Dateiname::=$\langle Path\rangle\langle Datei\rangle.\langle Suffix\rangle$

Path::=[$\langle Drive\rangle$:][\]{$\langle Ordner\rangle$\\}

Drive::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P

Ist ein Laufwerk angegeben, so wird in diesem Laufwerk gesucht, ansonsten im aktuellen. Steht dann ein Backslash, so wird der Pfad vom Wurzelverzeichnis aus gesucht, sonst vom aktuellen Verzeichnis. Hinter jedem Ordnernamen muß ein Backslash stehen. In das nächsthöhere Verzeichnis kommt man mit dem Ordnernamen “..”. Das TOS wandelt bei Datei- und Ordnernamen Kleinbuchstaben grundsätzlich in Großbuchstaben um. Umlaute werden dabei nicht gewandelt.

Beispiele:

CD	Aktueller Pfad	+ Dateiname	⟶ Resultierender Pfad
F:	F:\BIGFORTH	+ FORTH.SCR	⟶ F:\BIGFORTH\FORTH.SCR
F:	F:\BIGFORTH	+ GEM\AES.SCR	⟶ F:\BIGFORTH\GEM\AES.SCR
F:	F:\BIGFORTH	+ \FORTH.SCR	⟶ F:\FORTH.SCR
F:	F:\BIGFORTH\GEM	+ ..\FORTH.SCR	⟶ F:\BIGFORTH\FORTH.SCR
F:	F:\BIGFORTH\GEM	+ AES.SCR	⟶ F:\BIGFORTH\GEM\AES.SCR
F:	A:\	+ A:FORTH.SCR	⟶ A:\FORTH.SCR
F:	C:\AUTO	+ C:AHDI.PRG	⟶ C:\AUTO\AHDI.PRG

(CD meint Current Drive, also aktuelles Laufwerk. Jedes Laufwerk hat seinen eigenen aktuellen Pfad)

Beim Öffnen einer Datei weist TOS dieser ein Handle zu. Dieses Handle ist eine Nummer von 6 an aufwärts. TOS legt zu dem Handle eine Struktur an, aus der hervorgeht, in welchem Ordner die Datei ist, auf welchem Laufwerk sie an welchem Block beginnt, wie lang sie ist und wann sie erzeugt wurde. Zudem gibt es einen Schreib-Lese-Zeiger, der auf die Stelle zeigt, von der beim nächsten Lesebefehl gelesen bzw. beim nächsten Schreibbefehl geschrieben wird.

Diese Struktur muß natürlich zurückgegeben werden, wenn man die Datei nicht mehr braucht. Die Datei muß dann geschlossen werden. Dabei leert bigFORTH alle zur Datei gehörenden Puffer und schreibt veränderte zurück. Auch das TOS sichert seine Dateipuffer und gibt die Dateistruktur frei.

Im Open#-Field wird gezählt, wie oft die Datei mit OPEN geöffnet wurde. Endgültig geschlossen wird sie erst, wenn genausoviele CLOSE-Aufrufe auf sie erfolgt sind. Dadurch ist ein shared Use (geteilte Benutzung) möglich, mehrere Tasks können gleichzeitig auf die Datei zugreifen. Jeder öffnet die Datei ordentlich mit OPEN und schließt sie ordentlich mit CLOSE. Erst wenn kein Task oder in hierarchischen Strukturen kein Wort mehr den Zugriff auf die Datei beansprucht, wird sie tatsächlich geschlossen.

Beim Suchen nach Dateien erlaubt das TOS Wildcards. Das ? ersetzt ein beliebiges Zeichen, ∗ eine beliebige Zeichenkette. Konkret wird bei Verwendung des Sterns der Rest des Dateinamens bzw. Suffix mit Fragezeichen aufgefüllt, d. h. FILE∗X.SUF wird als FILE????.SUF interpretiert und findet auch Dateien, deren Namen nicht mit einem X endet.

Das TOS verteilt an seine Dateien Atribute. Sechs Bits sind dabei von Bedeutung: ADVSHR. A ist das Archive-Bit. Es wird gesetzt, wenn auf eine Datei ein Schreibzugriff stattgefunden hat. Ein Archivierungsprogramm kann an dem Archive-Bit eine veränderte Datei erkennen, nur veränderte Dateien kopieren und das Archive-Bit löschen. Erst das TOS 1.2 unterstützt das fehlerfrei.

D ist das Directory-Bit. Ein solcher Eintrag ist keine Datei, sondern ein Ordner. V ist das Volume-Bit, der “Dateiname” gibt den Diskettennamen an. Mit S markierte Dateien sind System-Dateien, diese Markierung hat aber keine Konsequenz. H ist das Hidden-Bit, versteckte Dateien werden vom Desktop nicht angezeigt. R schließlich ist das Read-Only-Bit, Dateien mit diesem Bit können nicht beschrieben werden. Allerdings erlaubt das TOS auch Schreibzugriffe auf nur zum Lesen geöffnete Dateien, somit ist die Schutzfunktion dieses Bit nur eingeschränkt.

Eine komfortable Arbeitsumgebung ist ohne Environment-Pathes nicht denkbar. Zur Ordnung vieler Dateien ist die Verwendung mehrerer Verzeichnisse einfach unumgänglich. Auf die wichtigsten Dateien will man aber ohne Pfadangabe zugreifen können. Das TOS selbst unterstützt nur die Suche in einem Verzeichnis. Klammert man das AES aus, das mit SHEL_FIND auch in allen Directories sucht, die im Environment-String hinter PATH= angegeben sind (normalerweise nur das Rootdirectory des Boot-Laufwerks), unterstützt TOS nur die Suche im aktuellen Directory.

Die vom TOS zur Verfügung gestellten Befehle sind also unbrauchbar, deshalb stellt bigFORTH eigene Environment-Pathes zur Verfügung. Die Pfade müssen komplett angegeben werden (also mit Backslash am Ende) und werden durch einen Strichpunkt getrennt. Bei der Suche nach Dateien wird zuerst das aktuelle Verzeichnis durchsucht (damit werden auch Dateien mit komplettem Pfad gleich gefunden), dann von vorn nach hinten die Environment-Pathes. Erst wenn auch hier die Suche erfolglos bleibt, wird abgebrochen.

Die Dateiverwaltung vor allem des TOS 1.0 ist etwas problematisch. Hier werden nach dem Ende eines Prozesses manchmal noch geöffnete Dateien nicht geschlossen. Mit bigFORTH ist das belanglos, da bigFORTH seine Dateien selbst schließt.

Zweitens kann das TOS 1.0 nur 40 Ordner verwalten. Alle Ordner, die im Laufe der Zeit gefunden werden, reiht es in seine Ordnerverwaltung ein. Sie bleiben im Speicher, bis entweder das Medium gewechselt oder der Computer ausgeschaltet wird. Leider tritt dieser Fehler meist schon dann auf, wenn von 40 Ordnern keine Spur ist, im Prinzip kann er sogar bei nur einem Ordner auftreten.

Das TOS "sammelt" die Ordner nämlich bei jedem Directory-Zugriff "ein". Leider gibt es auch hier einen Fehler, der bewirkt, daß nur ein vollständig durchsuchtes Directory nicht zu einer neuen Ordnerquelle wird. Greift man erneut auf ein bereits halb durchsuchtes Verzeichnis zu, so werden alle dabei gefundenen Ordner nochmal in die Ordnerverwaltung aufgenommen, doppelt und dreifach schließlich. Klar, daß irgendwann kein Speicher mehr vorhanden ist.

bigFORTH durchsucht deshalb jedes Verzeichnis bis zum Ende durch, auch wenn das etwas länger dauert, da gerade die gezielte Suche auf bekannte Dateien (wie beim Öffnen einer Datei) sehr fehlerträchtig ist.

Da das TOS auch Diskettenwechsel nicht immer korrekt verarbeitet, empfielt es sich, vor einem Diskettenwechsel mit FLUSH den Puffer zu leeren. Ansonsten werden die zur ausgeworfenen Diskette gehörenden Dateien geschlossen und sind nicht mehr ansprechbar. Da ihre Handles aber neu vergeben werden können, kann es durchaus passieren, daß man den veränderten Block einer Datei in einer anderen sichert - deshalb: FLUSH vor jedem Medienwechsel!

2. Die Top-Level-Befehle

Zu einer komfortablen Umgebung gehört, daß man Dateien und Ordner erzeugen und wieder löschen kann. Der aktuelle Verzeichnis-Pfad sollte gewechselt werden können, der Inhalt der Verzeichnisse auflistbar sein — sonst müßte man sich alle Namen merken. Dies alles steht im Vokabular FORTH, damit man nicht für diese wichtigen Funktionen das Vokabular wechseln muß.

FILEINT.SCR (−−): Die Zusätze des File-Interfaces werden aus der Datei FILEINT.SCR geladen.

>LEN (C$ −− addr count): Berechnet die Länge eines 0-terminierten Strings, wie er in der Programmiersprache C und im TOS verwendet wird.

PATH (−−) [⟨*Pfad*⟩{;⟨*Pfad*⟩}]: Ohne Parameter werden die aktuellen Environment-Pathes ausgegeben. Jeder Pfad ist im TOS-Format notiert, einzelne Pfade werden durch einen Strichpunkt abgetrennt. Im Feld PATHES für die Environment-Pathes haben maximal 128 Zeichen Platz.

EOF (−− flag): End Of File. Gibt true zurück, wenn der Schreib-Lese-Zeiger der aktuellen Datei das Dateiende anzeigt.

CREATEFILE (fcb −−): Erzeugt eine Datei mit dem in **fcb** gespeicherten Namen. Die Datei ist dann zum Schreiben und Lesen geöffnet und hat vorerst eine Länge von 0 Bytes. War in **fcb** vorher eine Datei geöffnet, so wird sie vor dem Erzeugen der neuen Datei geschlossen, beim Erzeugen dann normalerweise gelöscht, da die neue Datei ja mit demselben Namen erzeugt wird.

MAKE (−−) ⟨***Filename***⟩**:** Erzeugt eine Datei des Namens ⟨***Filename***⟩.

MAKEFILE (−−) ⟨***Filename***⟩**:**⟨***Filename***⟩ **(−−):** Erzeugt einen FCB mit dem Namen ⟨***Filename***⟩, kreiert eine neue Datei gleichen Namens, die zum Schreiben und Lesen geöffnet ist.

EMPTYFILE (−−): Setzt die Länge der aktuellen Datei auf 0 Bytes zurück, indem mit CREATEFILE eine Datei gleichen Namens erzeugt wird.

(MORE (n −−): Hängt **n** Blöcke an die aktuelle Datei an. Um die Verlängerung zu fixieren, muß die Datei aber geschlossen werden.

MORE (n −−): Hängt **n** Blöcke an die aktuelle Datei an und schließt sie. Dadurch wird die neue Länge fixiert.

RENAME (−−) ⟨***Alter Name***⟩ ⟨***Neuer Name***⟩**:** Dateien umbenennen. Der alte Name kann ein üblicher TOS-Suchstring sein, der neue muß ausformuliert sein (keine Wildcards). Ist der alte Name in irgendeinem FCB enthalten, wird er dort allerdings nicht verändert.

FROM (−−) ⟨***Filename***⟩**:[**⟨***Filename***⟩ **(−−):]:** Wechselt die Datei in FROMFILE, ohne die Isfile zu ändern. Ansonsten wie USE.

FILES (−−): Gibt alle Dateien und Ordner des aktuellen Verzeichnisses auf dem aktuellen Laufwerk aus. Zuerst werden die einzelnen Attribut-Bits mit Bezeichnung (A, D, V, S, H, R) in einem 6 Zeichen großen Feld rechtsbündig ausgegeben, nach einem Leerzeichen der Name in 15 Zeichen linksbündig, dahinter die Länge in 10 Zeichen rechtsbündig, 4 Leerzeichen, Uhrzeit im Format HH:MM:SS, zwei Leerzeichen und Datum (im FORTH-Format: DDmonJJ). Die Ausgabe kann mit ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ gestoppt, mit allen anderen Tasten unterbrochen und fortgesetzt werden.

Die ausgegebenen Ordner . und .. sind "Geisterordner", die auch in MS-DOS als erste Ordner in jedem Unterverzeichnis stehen, sie sind aus Kompatibilitätsgründen nötig.

FILES" (−−) ⟨***Suchpfad***⟩**":** Wie FILES, nur wird im angegebenen Suchpfad mit angegebenem Dateinamen gesucht (Wildcards sind möglich).

FREE? (−−): Gibt für das aktuelle Laufwerk die Anzahl der gesamten und freien Blöcken und Bytes aus.

KILLFILE (−−) ⟨***Filename***⟩**:** Löscht die Datei ⟨***Filename***⟩. Dabei sind Wildcards erlaubt. Vor jedem Löschvorgang wird die tatsächlich zu löschende Datei ausgegeben und nachgefragt, ob sie gelöscht werden soll. Nur wenn Sie die J- oder die Y-Taste drücken, wird wirklich gelöscht.

KILLDIR (−−) ⟨***Directory***⟩**:** Löscht das Verzeichnis ⟨***Directory***⟩. Dazu darf es keine Dateien mehr enthalten. Da ein versehentliches Löschen dann problemlos rückgängig gemacht werden kann, gibt es keine Sicherheitsabfrage.

MAKEDIR (−−) ⟨***Directory***⟩**:** Erzeugt das Verzeichnis ⟨***Directory***⟩.

DIR (−−) [⟨***Directory***⟩]**:** Gibt ohne Argument das aktuelle Laufwerk und Verzeichnis aus, mit Argument wird Laufwerk und/oder Verzeichnis neu gesetzt. Im Gegensatz zu DSETPATH verarbeitet DIR auch das Laufwerk.

(VIEW (%fffffffbbbbbbbbb −− blk'): Rechnet aus den Daten des View-Fields den Block aus und speichert die Datei in ISFILE.

FILER/W (file pos len addr r/w −−): Liest ab der Position **pos** der Datei **file len** Bytes nach **addr** (wenn **r/w**=0) oder schreibt sie von **addr** in die Datei (wenn

r/w=1). Wird in BLOCKR/W eingehängt. FILER/W erlaubt auch den Direktzugriff. Als Laufwerksauswahl wird die höchstwertige Hexziffer von **pos** benutzt. Damit sind bis zu 256 MByte direkt zugreifbar, das ist auch die maximale Größe von Medien, die AHDI 3.0 korrekt verwalten kann. Gelesen werden kann nur aus dem Datenbereich des Laufwerks, die Verwaltungsbereiche sind nicht zugänglich. FILER/W ist in BLOCKR/W eingehängt.

.BLK (−−): Hängt in .STATUS. Gibt " `Blk` " und die gerade geladene Blocknummer aus, wenn die nicht gerade 0 ist (TIB-Interpretation). Ändert sich die Datei, aus der gelesen wird, oder wird aus einer neuen Datei geladen, so wird in der nächsten Zeile am Anfang der Dateiname ausgegeben. Dadurch kann man verfolgen, aus welcher Datei gerade welcher Block geladen wird.

3. FCB-Struktur

Die folgenden Wörter sind im Vokabular DOS enthalten:

FILESIZE (fcb −− addr): Berechnet die Adresse des Size-Fields (4 Bytes) im File Control Block **fcb**.

FILEHANDLE (fcb −− addr): Berechnet die Adresse des Handle-Fields (2 Bytes).

FILEOPEN# (fcb −− addr): Berechnet die Adresse des Open#-Fields (2 Bytes).

FILENO (fcb −− addr): Berechnet die Adresse des Filenumber-Fields (2 Bytes). Hier wird die Nummer der Datei gespeichert. Die älteste Datei (FORTH.SCR) hat die Nummer 1.

FILENAME (fcb −− addr): Berechnet die Adresse des Dateinamens (Länge beliebig). Der Dateiname wird als 0-terminated String im Format des Betriebssystems (C-Format) gespeichert.

HANDLE (−− handle): Gibt das Handle der aktuellen Datei zurück.

4. Dateien öffnen und schließen

!FILES (fcb −−): Speichert **fcb** in ISFILE und FROMFILE. Dadurch kann voll auf die Datei zugegriffen werden.

!FCB (addr count fcb −−): Speichert den Dateinamen **addr count** im File Control Block **fcb**. Da die Länge des Dateinamens flexibel ist, darf nur mit !FCB ein neuer Dateiname gespeichert werden, andere Wege bringen das Memory Management aus dem Konzept (d. h. zum Absturz).

CLOSEFILE (handle −− 0 / -error): Deferred Word. Schließt die TOS-Datei handle und gibt bei Erfolg 0, ansonsten die übliche TOS-Fehlernummer zurück.

NOHANDLE (-error −− flag): Gibt true zurück, wenn TOS ein ungültiges Handle meldet.

(CLOSE (fcb −−): Schließt die Datei **fcb** und entfernt alle zu ihr gehörenden Blöcke aus dem Dateipuffer, nachdem die veränderten gesichert wurden.

OPENFILE (C$ −− len handle / -error): Deferred Word. Öffnet die Datei mit dem TOS-Namen **C$**. Es wird zuerst das aktuelle Directory durchsucht, dann alle in PATHES angegebenen. Zurückgegeben wird die Dateilänge **len** und das Handle, bei Mißerfolg die TOS-Fehlernummer (negativ).

(OPEN (fcb −−): Versucht die Datei, die durch den Dateinamen in FCB bezeichnet wird, zu öffnen.

(CAPACITY (fcb −− n): Berechnet aus der Dateilänge in **fcb** die Länge der Datei in Blöcken (KBytes). Es wird aufgerundet.

>PATH.FILE (C$ −− path\C$): Deferred Word. Sucht die Datei **C$** erst im aktuellen Directory, dann in allen in PATHES angegebenen. Der komplette Pfad, unter dem die Datei gefunden wurde, wird zurückgegeben.

(OPENFILE (C$ −− len handle / -error): Hängt in OPENFILE. Wandelt **C$** mit >PATH.FILE in den eigentlichen Dateinamen, öffnet diese Datei mit FOPEN und bestimmt ihre Länge mit 0 **handle** 2 FSEEK. Dadurch steht der Dateizeiger direkt nach dem Öffnen am Dateiende. Dateien mit einer zerstörten Struktur können an einer Länge −1 erkannt werden. Da diese Länge als vorzeichenlose Zahl betrachtet wird, kann trotzdem auf alle noch intakten Teile zugegriffen werden.

5. Fehlerausgabe

Das TOS gibt bei Mißerfolg seiner Aktionen Fehlernummern zurück. Diese sind negativ, damit können sie leicht von den anderen Rückgaben unterschieden werden, die immer positiv sind.

Natürlich informiert so eine Nummer den Benutzer nicht sehr und ein ständiges Blättern im Handbuch ist sicher nicht das Optimum an Benutzerfreundlichkeit. Deshalb wandelt bigFORTH die Fehlernummern auch in Klartext um. Diese Meldungen sind aussagekräftiger. Wer mehr wissen will, dem sei eine ausführliche TOS–Beschreibung ans Herz gelegt, wie die Artikelserie “Auf der Schwelle zum Licht” (ST-Computer 12/87-2/89).

>DISKERROR (−error −− string): Rechnet die negative TOS-Fehlernummer in eine Klartextmeldung um. Diese wird als counted String zurückgegeben.

.DISKERROR (-error −−): Gibt die TOS-Fehlernummer als Klartextmeldung aus.

?DISKABORT (-error / 0 −−): Bricht mit einer Klartextfehlermeldung ab, wenn eine Zahl ungleich null übergeben wird, ansonsten wird das Programm wie gewohnt fortgesetzt. Beim Abbruch verhält es sich wie ABORT“ ⟨***Meldung***⟩”.

(DISKERR (error# string −−): Hängt in DISKERR. Im Gegensatz zu (DISKERR des Kernels wird die Fehlernummer in Klartext gewandelt.

6. Directory-Verwaltung und File-Interface-Tools

DTA (−− addr): Gibt die Adresse der FORTH-eigenen DTA (Disk Transfer Area) zurück. In diesem Feld legen FSFIRST und FSNEXT ihre Informationen ab (siehe dort).

POSITION (offset handle −− false / -error): Setzt den Schreib-Lese-Zeiger der Datei mit dem TOS-Handle **handle** auf die Position **offset** (vom Anfang an gerechnet). Bei Erfolg wird 0 zurückgegeben, sonst die Fehlernummer.

POSITION? (handle −− offset): Gibt die Position des Schreib-Lese-Zeigers der Datei handle zurück.

?FCB (fcb / 0 −− fcb): Bricht mit der Fehlermeldung “Not for direct access” ab, wenn 0 übergeben wird (der FCB für Direktzugriff), ansonsten bleibt **fcb** auf dem Stack erhalten.

.FCB (fcb −−): Gibt Handle, Länge, FORTH-Name und Dateiname des File Control Block **fcb** aus.

PATHES (−− addr): Gibt die Adresse der Environment-Pathes zurück. Dieses Feld umfaßt 128 Zeichen. Die Environment-Pathes sind alle zusammen als counted String abgelegt, durch Strichpunkte getrennt. Sie müssen durch einen Backslash abgeschlossen sein.

.PATHES (−−): Gibt die aktuellen Environment-Pathes aus. Wie PATH ohne Parameter.

SETPATH (addr count −−): Speichert **addr count** als neue Environment-Pathes in PATHES.

(SEARCHFILE (fcb −− false / C$ true): Sucht den Dateinamen von **fcb** im aktuellen Verzeichnis und in allen Environment-Pathes. Gibt **false** zurück, wenn die Suche erfolglos war, den eigentlichen Dateinamen **C$** (mit Pfad) und **true**, wenn die Suche erfolgreich war.

SEARCHFILE (fcb −− C$): Bricht mit "File not found" ab, wenn der Dateiname von **fcb** von (SEARCHFILE nicht gefunden wurde.

>DATE (date −− addr count): Wandelt das Datum **date** vom TOS-Format in das FORTH-Text-Format um. In den ersten zwei Ziffern wird der Tag ausgegeben, dann folgen drei Buchstaben mit der Monatsbezeichnung (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec), anschließend die letzten zwei Ziffern der Jahreszahl (ab 80 heißt 19xx, unter 80 bedeutet 20xx).

.DTA (−−): Gibt die Informationen der DTA im selben Format wie bei FILES aus (.DTA wird von FILES verwendet).

(DIR (attr addr count −−): Gibt alle Dateien des aktuellen Directories mit dem Atribut **attr** aus, die auf den Suchstring im Feld **addr count** passen. Bei **attr**=8 werden nur Volume-Namen ausgegeben, ansonsten wird jede Datei ausgegeben, die entweder das Atribut 0 hat, oder in mindestens einem Bit mit **attr** übereinstimmt. (DIR ist die Subroutine von FILES.

FORTHFILES (−−): Gibt alle Dateien des FORTH-Systems aus. Es hangelt sich dabei durch die Kette von der Uservariablen FILE-LINK aus. Die Dateien werden mit .FCB ausgegeben, jede Datei in eine neue Zeile. Auch hier kann mit ⟨Ctrl⟩⟨C⟩ bzw. ⟨Esc⟩ gestoppt, mit jeder anderen Taste unterbrochen und fortgesetzt werden.

7. Der Direktzugriff

Eigentlich sollte der Direktzugriff nur blockweise möglich sein, schließlich geht das Konzept von FORTH von blockorientierten Massenspeichern aus. Aber das Memory Management (siehe Kapitel 8) betrachtet Massenspeicher als Dateien einer bestimmten Länge, aus denen ein beliebiger Teil (byteweise positioniert) ausgelesen werden kann. Deshalb kann der Direktzugriff auch zwischen den Sektorgrenzen starten und enden. Natürlich braucht der Zugriff dann länger, es muß zusätzlich noch ein Puffer eingerichtet werden.

Die Basis für den Laufwerkzugriff bildet das BIOS. Mit RWABS können Sektoren vom Laufwerk gelesen oder auf das Laufwerk geschrieben werden. Die Belegung der Sektoren steht im BIOS Parameter Block (BPB).

BPBS (−− addr): In diesem Puffer sind die BPBs (BIOS Parameter Block) der Laufwerke untergebracht. Sie sind zur Berechnung des Zugriffes erforderlich. Ausgewertet werden die Nummer des ersten Datensektors, die Sektorlänge und die Anzahl der Sektoren. Laufwerke, auf die noch nicht zugegriffen wurde, sind hier durch einen Zeiger auf NIL markiert. Da sich die Adresse des BPB üblicherweise nicht ändert, ist diese Speicherung erlaubt, außerdem stellt es die einzige Möglichkeit dar, Laufwerkswechsel

korrekt zu behandeln, auch wenn schon von anderer Seite (vom TOS) auf das Laufwerk zugegriffen wurde.

B/DRV (−− n): Gibt die Anzahl der Bytes pro Laufwerk (Byte per Drive) zurück. Dabei wird bei Laufwerkswechsel und beim ersten Zugriff des Systems der BPB geholt.

(BLK/DRV (−− n): Gibt die Blöcke pro Laufwerk (Blocks per Drive) zurück. Benutzt dabei B/DRV.

R/WBUFFER (−− addr): Schreib-Lese-Puffers für einzelne Sektorzugriffe.

(DRVINIT (−−): Löscht den BPB-Puffer. Dadurch muß bei einem Direktzugriff der BPB neu geholt werden. Hängt in DRVINIT.

8. TOS-Befehle

Dieses Kapitel soll keine detaillierte Beschreibung der TOS-Routinen darstellen. Das Thema kann dicke Bücher füllen ("Atari ST Intern" (Data Becker) oder "Das Atari ST Profibuch" (Sybex Verlag)). Die hier gegebenen Informationen mögen Anwendern genügen, die ihr Wissen bereits aus solchen Quellen gewonnen haben, oder über genügend Experimentierfreudigkeit verfügen, es sich selbst anzueignen. Dieses Kapitel wurde aus den Serien "ST-Betriebssystem" (ST-Computer 4/86-2/87), "Auf der Schwelle zum Licht" (ST-Computer 12/87-12/88) und dem Handbuch von Omikron.BASIC zusammengestellt.

8.1. GEMDOS

GEMDOS (GEM Disk Operation System) ist, wie der Name sagt, das Betriebssystem für GEM auf dem Atari ST. Da GEM für MS-DOS-Rechner geschrieben wurde, ist GEMDOS funktionell eine Kopie von MS-DOS. Parameter und sogar Funktionsnummern der Routinen stimmen mit den entsprechenden MS-DOS-Routinen überein.

GEMDOS lehnt sich an das File-System von UNIX an: Ein hierarchisches Dateisystem mit zeichenorientierten Dateien und Ein/Ausgabeumleitung. Nur Multitasking gibt es leider nicht.

FREAD (addr len handle −− #Bytes / -error): Liest **len** Bytes der Datei **handle** in den Puffer ab **addr**. Zurückgegeben wird die Anzahl tatsächlich gelesener Bytes oder eine TOS-Fehlernummer. Ist **#Bytes** kleiner als **len**, so war die Datei kürzer als die angeforderte Länge.

FWRITE (addr len handle −− #Bytes / -error): Schreibt **len** Bytes ab **addr** in die Datei **handle**. Zurückgegeben wird die Anzahl tatsächlich geschriebener Bytes oder eine TOS-Fehlernummer. Ist **#Bytes** kleiner als **len**, so war nicht mehr genügend Platz auf dem Laufwerk.

FSEEK (offset0 handle modus −− offset1 / -error): Setzt den Schreib/Lesezeiger der Datei handle. **modus** hat folgene Bedeutung:

modus=0: **offset0** vom Dateianfang an gerechnet.

modus=1: **offset0** relativ von der aktuellen Zeigerposition gerechnet.

modus=2: **offset0** vom Dateiende an gerechnet. **offset0** muß dann negativ sein.

Zurückgegeben wird entweder die neue Position des Schreib/Lesezeigers (bezogen auf den Dateianfang) oder eine Fehlernummer.

FCREATE (C$ −− handle / -error): Erzeugt eine Datei mit dem Namen **C$**. Zurückgegeben wird das Dateihandle. Die Datei ist dann zum Schreiben und Lesen geöffnet.

FDELETE (C$ −− 0 / -error): Löscht die Datei **C$**. Zurückgegeben wird 0, wenn die Ausführung geglückt ist, sonst eine TOS-Fehlernummer.

FOPEN (C$ −− handle / -error): Öffnet die Datei **C$** zum Lesen und Schreiben. Zurückgegeben wird das Handle oder eine TOS-Fehlernummer. Es können auch die "Dateien" "CON:" (Bildschirm), "AUX:" (serielle Schnittstelle) und "PRN:" (Drucker über Centronics) geöffnet werden.

FCLOSE (handle −− 0 / -error): Schließt die Datei **handle**. Bei Erfolg wird 0 zurückgegeben, sonst die TOS-Fehlernummer.

FGETDTA (−− addr): Gibt die Disk Transfer Area zurück. Die DTA ist der Übergabepuffer bei der Dateisuche. Ihr Aufbau:

Länge	Offset	Bedeutung
12	0	Suchname von FSFIRST (Format: NNNNNNNNSSS, N für Name, S für Suffix)
1	12	Suchattribut
4	13	letzte Suchposition
4	17	Zeiger auf den Directory Deskriptor des Suchdirectories

Die obigen Daten sind TOS-intern, sie können (sollen!) sich in zukünftigen TOS-Versionen ändern. Die folgenden Daten sind zugesichert:

1	21	gefundenes Attribut
2	22	gefundene Zeit
2	24	gefundenes Datum
4	26	gefundene Länge
14	30	gefundener Dateiname

FSETDTA (addr −−): Setzt die Disk Transfer Area. Dies ist notwendig, da nach dem Start von bigFORTH der Puffer für die Kommandozeile als DTA gesetzt ist (vom TOS) und dieser nicht überschrieben werden darf — zumindest solange die Kommandozeile interpretiert wird.

FSFIRST (C$ attr −− false / -error): Sucht nach der Datei **C$** (Wildcards möglich). Alle Dateien mit dem Attribut 0, außerdem Dateien, deren Attribut mit **attr** in mindestens einem Bit übereinstimmt, und Dateien, deren R- und A-Bit gesetzt sind, werden gefunden. Ausnahme: **attr**=8 findet nur alle Arten von Diskettennamen. Die erste gefundene Datei wird in der DTA gespeichert. Wurde die Datei gefunden, wird **false** übergeben, sonst eine TOS-Fehlernummer.

FSNEXT (−− false / -error): Sucht mit dem Argument des letzten FSFIRST weiter. Auch hier dient die DTA als Übergabepuffer zur Aufname der gefundenen Dateien. Solange **false** übergeben wird, kann weitergesucht werden.

FRENAME (C$old C$new −− false / -error): Benennt die Datei **C$old** in **C$new**. Dabei kann der neue Name auch in einem anderen Verzeichnis stehen, muß aber auf demselben Laufwerk liegen. Bei korrekter Ausführung wird **false** zurückgegeben, sonst eine TOS-Fehlernummer.

DCREATE (C$ −− 0 / -error): Erzeugt einen Ordner mit dem Namen **C$**.

DDELETE (C$ −− 0 / -error): Löscht den Ordner mit dem Namen **C$**. Der Ordner darf dabei keine Dateien mehr enthalten, sonst wird mit einer TOS-Fehlernummer abgebrochen.

DSETPATH (C$ −− 0 / -error): Setzt den aktuellen Pfad auf **C$**. Eine eventuelle Laufwerksangabe wird nicht berücksichtigt, der Pfad kann also nur für das aktuelle Laufwerk gesetzt werden.

DGETPATH (buffer drive+1 −− false / -error): Holt den aktuellen Pfad des Laufwerks **drive** in den Puffer ab **buffer**. Das **drive** −1 (der Übergabeparameter 0) ist das aktuelle Laufwerk.

DFREE (drive+1 −− total_units free_units b/unit): Berechnet den totalen und den freien Speicherplatz des Laufwerks **drive**. Das **drive** −1 ist auch hier das aktuelle

Laufwerk. Durch einen Fehler im TOS sind bereits auf einem ganz leeren Medium zwei Einheiten ("Units" oder "Cluster") verbraucht. Normalerweise ist ein Cluster ein KByte groß, ab AHDI 3.0 sind auch größere Cluster erlaubt.

DSETDRV (drive −−): Setzt das aktuelle Laufwerk auf **drive**. Dabei ist **drive** 0=A:, **drive** 1=B: usw.

DGETDRV (−− drive): Gibt die Nummer des aktuellen Laufwerks zurück.

TGETTIME (−− time): Holt die aktuelle Zeit. Das 16-Bit-Wort hat folgendes Format: (MSB) HHHHHMMMMMMSSSSS (LSB). HHHHH=Stunden im 24-Stunden-Format, MMMMMM=Minuten und SSSSS=Sekunden∗**2**.

TGETDATE (−− date): Holt das aktuelle Datum. Das 16-Bit-Wort hat folgendes Format: JJJJJJJMMMMTTTTT. JJJJJJJ=Jahr ab 1980. MMMM=Monat, TTTTT=Tag.

TSETTIME (time −−): Setzt die Uhrzeit.

TSETDATE (date −−): Setzt das Datum.

In DOS.SCR definierte Befehle (DOS.SCR muß nachgeladen werden!):

Alle mit C beginnenden GEMDOS-Befehle sollte man von bigFORTH aus nicht benutzen, da es elegantere Möglichkeiten gibt. Zudem besteht bei einigen Befehlen die Gefahr, daß nach einer Eingabe von ⟨Ctrl⟩⟨C⟩ bigFORTH mit pterm(−32) verlassen wird — da die Vektoren dabei nicht zurückgebogen werden, eine gefährliche Angelegenheit.

Die Ein/Ausgabe erfolgt über die logischen Ausgabekanäle von GEMDOS. Diese haben die Nummern von 0 bis 3. Diese Kanäle können umgeleitet werden, müssen also nicht auf das Default-Device zeigen. Die Bedeutung:

Handle (Kanal)	Zweck	Default-Device
0	Standardeingabe	CON:
1	Standardausgabe	CON:
2	Standard-Hilfs-Device	AUX:
3	Standarddrucker	PRN:

Die Default-Devices haben folgende Handles:

Handle (Device)	Name	Gerät
−1	CON:	Tastatur/Bildschirm
−2	AUX:	Serielle Schnittstelle RS 232
−3	PRN:	Drucker, Centronics-Port

CCONIN (−− key): Liest ein Zeichen vom Kanal 0. Es wird solange gewartet, bis eines vorhanden ist. Das Zeichen wird auf demselben Kanal noch einmal ausgegeben.

CCONOUT (char −−): Gibt das Zeichen **char** auf Kanal 1 aus. TAB wird zu Leerzeichen expandiert.

CAUXIN (−− char): Ein Zeichen wird vom Kanal 2 gelesen. Dabei wird solange gewartet, bis eines vorhanden ist.

CAUXOUT (char −−): Das Zeichen **char** wird auf Kanal 2 geschrieben. Dabei wird solange gewartet, bis das Device annahmebereit ist.

CPRNOUT (char −− flag): Das Zeichen **char** wird auf Kanal 3 geschrieben. Dabei wird solange gewartet, bis das Device annahmebereit ist. Bei einem Timeout ist **flag** ungleich 0.

CRAWIO (char / $FF −− key / false): Liest ein Zeichen von Kanal 0 ein, wenn $FF übergeben wird. Ist keines vorhanden, so wird 0 zurückgegeben. Wird ein anderer Wert als $FF übergeben, so wird **char** auf Kanal 1 ausgegeben.

CRAWCIN (−− key): Liest ein Zeichen von der Standardeingabe ein. Es wird gewartet, bis eines vorhanden ist, es erfolgt kein Echo.

CCONWS (C$ −−): Schreibt den 0-terminated String **C$** auf Kanal 1.

CCONRS (buffer −−): Liest eine Zeichenkette von Kanal 0 in den Puffer **buffer**. Es stehen primitive Editiermöglichkeiten zur Verfügung:

⟨BS⟩ ⟨DEL⟩:letztes eingegebens Zeichen löschen.
⟨TAB⟩ :Tabulator.
⟨Ctrl⟩⟨C⟩ :Abbruch des Prozesses mit pterm(-32) (darf in bigFORTH nicht :passieren! ⟨Ctrl⟩⟨C⟩ auf keinen Fall eingeben!).
⟨Ctrl⟩⟨X⟩ :alle bisher eingegebenen und die im GEMDOS-Puffer wartenden Zeichen löschen.
⟨Ctrl⟩⟨U⟩ :"#" ausgeben, Cursor genau eine Zeile unter die alte Anfangsposition :setzen, die bisher eingegebenen Zeichen vergessen.
⟨Ctrl⟩⟨R⟩ :Wie ⟨Ctrl⟩⟨U⟩ und dann bisherige Eingabe dorthin kopieren.

CCONIS (−− flag): Ermittelt den Eingabestatus von Kanal 0. true bedeutet, daß ein Zeichen anliegt, false bedeutet keine Eingabe.

CCONOS (−− flag): Ermittelt den Ausgabestatus von Kanal 1. true bedeutet, daß das Gerät empfangsbereit ist, false, daß es nicht empfangsbereit ist.

CAUXIS (−− flag): Ermittelt den Eingabestatus von Kanal 2.

CAUXOS (−− flag): Ermittelt den Ausgabestatus von Kanal 2.

CPRNOS (−− flag): Ermittelt den Ausgabestatus von Kanal 3.

SVERSION (−− version): Gibt die TOS-Versionsnummer zurück. Diese ist im Format byte reversed, binary fixed oder auch Intel binary fixed gespeichert (Quelle: Bernd Rosenlechner, TOS DATEN, ST-Computer 1/90, S. 122 ff.). Das heißt, man muß das High-Byte des 16-Bit-Wertes als Low-Byte interpretieren und umgekehrt. Zur Veranschaulichung die Wandlungsroutine in FORTH:

```
: .SVERSION ( sversion -- )
$100 /mod swap $100 * + 0 <# # # Ascii . hold #S #> type ;
```

FATTR (attr flag C$ −− attr / -error): Setzt oder liest das Attribut der Datei **C$**. Gelesen wird, wenn **flag**=0, sonst wird gesetzt. Bei Erfolg wird das Attribut (bei **flag**=0 das alte, bei **flag**=1 das neue) zurückgegeben, sonst die TOS-Fehlernummer.

FDUP (physcan −− handle): Erzeugt **handle** (6-80), mit dem auf dieselbe Datei/denselben Kanal wie mit **physcan** zugegriffen werden kann.

FFORCE (physcan logcan −−): Legt **logcan** auf **physcan**. **logcan** wird dabei eines der Standardgeräte sein (0-3). **physcan** ist entweder ein Default-Device (−1, −2 oder −3) oder ein Dateihandle.

FDATTIME (flag handle addr −−): Der "timestamp" der Datei **handle** wird gelesen (**flag**=0) bzw. geschrieben (**flag**=1). **addr** ist ein Zeiger auf einen 4 Byte langen Puffer. An **addr** steht die Zeit, an **addr**+2 das Datum (beide im GEMDOS-Format). Der Pufferinhalt sollte nach dem Schreiben nicht mehr benutzt werden, da das TOS den Inhalt in das Intel-Format gewandelt hat.

MSHRINK (len addr 0 −−): Setzt den Block **addr** auf die Länge **len**. 0 ist ein Dummy. Der Speicherblock wurde mit MALLOC angefordert. Die Länge des Blockes kann nur verkürzt werden. MSHRINK wird in der Regel nach dem Programmstart eingesetzt.

PEXEC (environment command name mode −− rwert): Dient zum Laden und Starten eines GEMDOS-Prozesses. Dabei ist **environment** ein Zeiger auf den Environment-String. 0 heißt hier, daß der Environmentstring des Parent-Prozesses übernommen wird. **command** ist ein Zeiger auf die Kommandozeile. Diese ist ein counted String, CR-0-terminiert. Das sieht so aus:

|Count|Count Bytes...|$0D|$00|

name ist der Pfadname des zu startenden Prozesses im GEMDOS-Format. **mode** kann folgende Werte annehmen:

0: Starten und laden. **rwert** ist dann der Rückgabewert des Programms. Nur die unteren 16 Bit sind signifikant.

3: Nur laden. Environmentstring und Programmspeicher werden dann unter dem PD (Prozess Deskriptor) des Parent-Prozesses angelegt, also nach Beendigung des aufgerufenen Programms nicht zurückgegeben. **rwert** ist die Adresse des PD des geladenen Programms.

4: Nur starten. **environment** und **command** werden nicht ausgewertet. Statt **name** wird der von PEXEC #3 zurückgegebene PD-Zeiger übergeben. **rwert** ist der Rückgabewert des aufgerufenen Programms.

5: Basepage anlegen. Es wird eine leere Basepage angelegt. **name** wird ignoriert. Zurückgegeben wird der Zeiger auf die Basepage (PD-Zeiger).

6: Nur starten. Im Gegensatz zu Modus 4 werden Environmentstring und Programmspeicher dem PD des aufgerufenen Prozeß zugeordnet und deshalb nach dessen Beendung freigegeben. Erst ab TOS 1.4 verfügbar.

Ein gutes Anwendungsbeispiel ist das Wort RUN“:

```
: RUN" ( -- rwert ) \ <Kommando>" <Name>
  Ascii " parse pad place $0D00 pad capitalize count + w!
  0 pad name 0 over count + c! 1+ 0 pexec wextend ;
```

RUN“ ist in DOS.SCR definiert, allerdings im Vokabular FORTH, da es die Anwendung von PEXEC leicht zugänglich macht:

RUN“ (−− rwert) ¡Kommando¿” ¡Name¿: Startet das Programm **¡Name¿** mit der Kommandozeile **¡Kommando¿**. **rwert** ist der Rückgabewert, normal 0. Eine negative Nummer weist auf einen Fehler hin. BIGFORTH.PRG kann selbst mit diesem Befehl gestartet werden. Allerdings muß man dann zuerst für genügend Platz sorgen - mindestens $50000 Bytes (320 KBytes). Dazu gibt man folgende Zeile ein:

INCLUDE␣RELOCATE.SCR␣$50000␣RESERVE␣BIGFORTH.PRG␣BYE⟨RET⟩

Anschließend startet man BIGFORTH.PRG neu.

RUN"␣DOS.SCR"␣BIGFORTH.PRG⟨RET⟩

startet BIGFORTH.PRG aus bigFORTH heraus. Dort wird dann die Datei DOS.SCR (Screen 1) zum Editieren angeboten.

RUN"␣include␣STARTUP.SCR␣savesystem␣BIGFORTH.PRG"␣FORTHKER.PRG⟨RET⟩

startet das Kernel. Dieses lädt STARTUP.SCR und sichert das Ergebnis anschließend als BIGFORTH.PRG.

8.2. BIOS

Das BIOS (Basic Input Output System) verwaltet zeichen- und blockorientierte Geräte (Bildschirm, Schnittstellen bzw. Laufwerke). Die zeichenorientierten Befehle sind schon im Kernel (FORTH.SCR) definiert und erklärt. Sie werden deshalb hier nicht nochmals aufgelistet.

RWABS (drive begsec #sec buf r/w −− ret): Liest aus dem Laufwerk **drive** vom Sektor **begsec** an **#sec** Sektoren in den Puffer ab der Adresse **buf**, wenn Bit 0 von **r/w** 0 ist, ansonsten wird geschrieben. Ist Bit 1 gesetzt, so werden Laufwerkswechsel nicht

berücksichtigt, ansonsten wird bei einem Laufwerkswechsel mit einer Fehlernummer abgebrochen.

MEDIACH (drive −− flag): Stellt fest, ob die Diskette **drive** gewechselt wurde: 0=nein, 1=vielleicht, 2=ja. Die Rückgabe 1 kommt vor allem bei schreibgeschützten Disketten vor, oder bei Systemen mit nur einem Laufwerk, da hier ein Laufwerkwechsel ja auch bedeuten könnte, daß "Diskette B:" im Laufwerk A: liegt. Ab AHDI 3.0 können auch Wechselplatten während des Rechnerbetriebs gewechselt werden.

GETBPB (drive −− bpb): Holt die Adresse des BIOS Parameter Block. Der Block besteht aus neun Integer-Worten und acht Bytes:

Bytes pro Sektor, Sectors pro Cluster, Bytes pro Cluster, Sektoren des Rootdirectories, Sektoren je FAT, Sektornummer des zweiten FATs, erster Datensektor, Anzahl der Datensektoren.

Das LSB des ersten Bytes ist bei 12-Bit-FATs gelöscht, bei 16-Bit-FATs gesetzt. Alle weiteren Bytes sind bislang reserviert und auf 0 gesetzt.

In DOS.SCR definierte Befehle:

SETEXC (vecaddr #vec −− vecaddr): Setzt den Vektor **#vec** auf **vecaddr**. Ist **vecaddr**= −1, so wird nur ausgelesen. Zurückgegeben wird die alte Adresse. Die Systemvektoren sind Zeiger ab der Adresse 0. SETEXC kann also durch 4∗ ! oder 4∗ @ ersetzt werden, da bigFORTH ohnehin im Supervisormodus läuft und uneingeschränkt Zugriff auf die geschützten Bereiche hat.

TICKCAL (−− time): Gibt die Zeit zwischen zwei Timer-Aufrufen in Millisekunden zurück.

DRVMAP (−− map): Gibt eine Bitmap zurück, in der alle ansprechbaren Laufwerke eingetragen sind. Dabei steht das LSB für Laufwerk A:, die höheren dann für die weiteren Laufwerke. Beispiel: Bei einem ST ohne Festplatte wird 3 (%11) zurückgegeben, mit RAM-Disk D: 11 (%1011). Das BIOS kann 32 Laufwerke verwalten, das GEMDOS aber nur 16, also ist der Rückgabewert zwar ein 32-Bit-Wert, aber nur 16 Bit sind signifikant.

KBSHIFT (status0 −− status1): Liest den Status der Tastatur aus oder setzt ihn. Der Wert hat folgende Bedeutung: Bit 0: Rechte Shifttaste gedrückt, Bit 1: Linke Shifttaste gedrückt, Bit 2: Control-Taste gedrückt, Bit 3: Alternate-Taste gedrückt, Bit 4: Caps on. Um den Status abzufragen, muß −1 übergeben werden, ansonsten wird der Status neu gesetzt — sinnvoll ist das sicher nur für Caps on, da man hier (wie z . B. bei 1st Wordplus) mit einen Button und per Mausklick auf Großschrift umschalten kann.

8.3. XBIOS

Das XBIOS (eXtended BIOS) verwaltet Atari-spezifische Geräte und ist so sehr hardwarenah. Es ist eine Erweiterung des BIOS.

FLOPFMT (init $87654321 int side track sec# drv ∗int buf −− 0 / -error): Formatiert die Spur **track** auf der Seite **side** (0 oder 1, auf einseitigen Disketten nur 0) des Laufwerks **drv** (nur A: oder B:) mit **sec#** Sektoren. **buf** ist ein Zeiger auf einen 10 KByte großen Puffer. **init** sind zwei Bytes, mit denen die Sektoren initialisiert werden. **int** ist der Interleavefaktor (normalerweise 1). ∗**int** hat eine ähnliche Funktion (aber erst ab TOS 1.2): Es zeigt auf eine 16-BitTabelle, in der die Reihenfolge der logischen Sektornummern steht. Damit kann man z . B. mit einem Spiralisierungsfaktor formatieren. Dazu muß **int**=-1 sein. Ansonsten wird **int** benutzt und ∗**int** muß 0 sein.

Als zweiter Parameter muß die Konstante $87654321 übergeben werden, nur dann wird formatiert (Schutz vor Datenverlust durch Programmfehler).

RANDOM (−− 24b): Berechnet eine 24-Bit-Zufallszahl.

CURSCONF (rate mode −− rwert): Cursor Configuration. **mode** hat die Bedeutung:

0: Cursor aus

1: Cursor ein

2: Cursor blinkend

3: Cursor stabil

4: Blinkgeschwindigkeit setzen. Nur hier hat **rate** eine Bedeutung.

5: Blinkgeschwindigkeit abfragen. Nur hier hat **rwert** eine Bedeutung.

KBRATE (delay0 speed0 −− delay1 speed1): Setzt Geschwindigkeit (**speed0**) und Verzögerung (**delay0**) der Tastaturwiederholung und gibt die alten Werte zurück. Alle Zeiten werden in fünfzigstel Sekunden gemessen, übergeben Sie für **delay0** oder **speed0** −1, wird der entsprechende Wert nicht verändert.

In DOS.SCR definierte Befehle:

Viele XBIOS-Befehle werden nur zur Initialisierung des STs nach dem Kaltstart benötigt und sind nachher wertlos bzw. unsinnig. Auf die Verwendung sollte daher verzichtet werden.

INITMAUS (rout tab mode −−): Initialisiert die Maus. **mode** hat die Bedeutung:

0: Maus ausschalten

1: Maus einschalten, relativer Modus

2: Maus einschalten, absoluter Modus

4: Maus einschalten, Tastaturmodus (Mausbewegungen werden in Cursorbewegungen übersetzt).

Die Mausroutine **rout** steht an KBDVBASE+16 - man sollte immer die TOS-Routine benutzen (beim Aufruf von INITMAUS mit KBDVBASE 16 + @ auslesen!). **tab** zeigt auf ein Bytefeld, das nur bei Modus 1 und 2 ausgewertet wird und deren Werte folgende Bedeutung haben:

Topmode: =0 Y-Achse von unten nach oben, =1 Y-Achse von oben nach unten.
Buttons: Bit 0: Bei Drücken Mausposition melden
Bit 1: Bei Loslassen Mausposition melden
Bit 2: Bei Drücken Tastencodes melden
x-Teilung (im relativen Modus) | xmax (im absoluten Modus)
y-Teilung (im relativen Modus) | ymax (im absoluten Modus)
xstart (absoluter Modus)
ystart (absoluter Modus)

Das TOS intitialisiert mit $01,$03,$01,$01, im relativen Modus. Die Teilung bedeutet, daß nur Schritte über der Teilgröße gemeldet werden, bestimmt also die Größe

der Schritte, die die Maus macht, verändert aber nicht die Relation von Handweg zu Mausweg auf dem Bildschirm.

PHYSBASE (−− pbase): Ermittelt die Anfangsadresse des tatsächlich dargestellten Bildschirms.

LOGBASE (−− lbase): Ermittelt die Anfangsadresse des Bildschirms, auf den gerade ausgegeben wird.

GETREZ (−− rez): Ermittelt die Bildschirmauflösung:

$0 = 320 * 200$ Punkte, 16 Farben (niedrig, Farbmonitor)

$1 = 640 * 200$ Punkte, 4 Farben (mittel, Farbmonitor)

$2 = 640 * 400$ Punkte, 2 Farben (hoch, S/W-Monitor)

$4 = 640 * 480$ Punkte, 16 Farben (TT mittel, nur auf TT)

$6 = 1280 * 960$ Punkte, 2 Farben (TT hoch, nur auf TT)

$7 = 320 * 480$ Punkte, 256 Farben (TT niedrig, nur auf TT)

SETSCREEN (rez pbase lbase −−): Setzt die Bildschirmauflösung und die Adressen. **rez** ist die Auflösung, **pbase** der dargestellte Bildschirm und **lbase** der, auf den ausgegeben wird. Ein Parameter -1 bedeutet, daß der alte Wert erhalten bleibt. Bei Änderung der Auflösung werden die GEM-Variablen nicht mit angepaßt!

SETPAL (tabaddr −−): Setzt Farben. **tabaddr** zeigt auf einen Speicherbereich, in dem 16 Integer-Werte stehen. Dabei hat ein Integer folgende Aufteilung (in Halbbytes): $0RGB, wobei die Bit-Wertigkeit eines Halbbytes 0321 ist. Beim ST wird das oberste Bit nicht benutzt, beim STE und TT wird es als LSB betrachtet, damit die Erweiterung der Farbpalette von 512 auf 4096 Farben aufwärtskompatibel ist. Beispiel: Weiß auf dem ST ist $0777, Rot ist $0700. Auf dem STE und TT gibt es ein "noch weißeres" Weiß: $0FFF. Dort wird eine Farbintensität so gezählt: 0, 8, 1, 9, 2, 10 usw.

SETCOLOR (col numb −−): Setzt die Farbe **numb** mit dem Farbwert **col**. Dabei hat **col** dieselbe Aufteilung wie die Integerwerte bei SETPAL.

FLOPRD (sec# side track sec drv 0 buffer −−): Liest vom Laufwerk **drv** (nur A: oder B:) **sec#** Sektoren ab dem Sektor **sec** vom Track **track** in den Puffer ab buffer. Über das Spurende hinaus kann nicht gelesen werden.

FLOPWR (sec# side track sec drv 0 buffer −−): Schreibt Sektoren. Parameter wie FLOPRD.

FLOPVER (sec# side track sec drv 0 buffer −− flag): Verifiziert Sektoren. Parameter wie bei FLOPRD. Stimmen die Daten auf Diskette mit denen im Puffer überein, wird 0 zurückgegeben, sonst eine negative Zahl. Der Puffer **buffer** enthält in der ersten Hälfte die Daten, die auf der Diskette stehen sollen, die zweite Hälfte wird für die auf Diskette stehenden Daten benutzt.

MIDIWS (addr count −−): Sendet Daten über die MIDI-Schnittstelle ab.

MFPINT (addr n −−): Installiert eine Interruptroutine für die Nummer **n**:

0 = Centronics Busy
1 = RS 232 DCD
2 = RS 232 CTS
4 = Timer D
5 = Timer C
6 = Tastatur- und MIDI-ACIAs
7 = FDC- und DMA-Chip
8 = Timer B
9 = RS 232 Sendefehler
10 = RS 232 Sendepuffer leer
11 = RS 232 Empfangsfehler
12 = RS 232 Empfangspuffer voll
13 = Timer A
14 = RS 232 Ring Indicator
15 = Monochrom detect

Die Interruptroutinen mit den Nummern **n**=0 bis 7 müssen Bit n von $FFFA11 löschen, die mit den Nummern **n**=8 bis 15 Bit n-8 von $FFFA09, um die Interrupts niedrigerer Priorität wieder freizugeben.

IOREC (dev −− buffer): Ermittelt den Zeiger auf die Eingabepuffer des Gerätes **dev**. Dabei bedeutet:

dev=0 RS 232, Ausgabepuffer schließt an

dev=1 Tastatur

dev=2 MIDI

Der Puffer hat folgenden Aufbau:

.L Pufferadresse

.W Länge

.W Offset für neue Daten (Head)

.W Offset fur herauszunehmende Daten (Tail)

.W “Low water mark”

.W “High water mark”

RSCONF (Scr Tsr Rsr Ucr handshake baud −− ret): Setzt Werte für die RS 232-Schnittstelle. Eine Übergabe von -1 bedeutet, daß der alte Wert nicht verändert wird. **baud** ist ein Wert von 0 bis 15, die Werte stehen der Reihe nach für folgende Baudraten:

19200,9600,4800,3600,2400,2000,1800,1200,600,300,200,150,134,110,75,50

In **handshake** steht Bit 0 für XON/XOFF und Bit 1 für RTS/CTS. **Ucr**, **Rsr**, **Tsr** und **Scr** setzen die gleichnamigen Register des MFPs:

UCR	Bit 0	: unbenutzt
	Bit 1	: 0=odd parity, 1=even parity
	Bit 2	: 1, wenn parity
	Bit 3+4	: 0=syncron, 1=1 Stopbit, 2=1,5 Stopbit, 3=2 Stopbit
	Bit 5+6	: 0=8 Bit, 1=7 Bit, 2=6 Bit, 3=5 Bit
	Bit 7	: 0=Frequenz nicht teilen (nur synchron), 1=Frequenz teilen
RSR	Bit 0	: 1, wenn RS 232-Empfänger ein
	Bit 1	: 1, wenn SCR-Zeichen mit übertragen
	Bit 2-7	: sind nur abfragbar
TSR	Bit 0	: 1, wenn RS 232-Sender ein
	Bit 1+2	: 0=Ausgang hochohmig, 1=H, 2=L, 3=Ausgang am Eingang
	Bit 3	: 1=Break senden (nur asynchron)
	Bit 5	: 1=Empfänger einschalten, wenn Zeichen fertig gesendet
	Bit 4,6,7	: nicht setztbar
SCR		: Enthält Synchronisationsbyte für Synchron-Betrieb

Wird für **baud** −2 übergeben, so ist **ret** die alte eingestellte Baudrate. Ansonsten werden in **ret** die vier Register Scr, Tsr, Rsr und Ucr zurückgegeben (in dieser Reihenfolge vom High-Byte bis zum Low-Byte), die den gleichnamigen Übergabeparametern entsprechen.

KEYTABL (key KEY keycaps −− tabblk): Setzt die Tastaturtabellen. **key** ist für einfache Tasten, **KEY** in Verbindung mit der Shift-Taste und **keycaps**, wenn Caps on ist und die Shift-Taste nicht gedrückt wird. Jedes Zeichen steht dabei an der Position, die dem Scancode der entsprechenden Taste entspricht. **tabblk** ist ein Zeiger auf die Tabelle mit den drei Zeigern (in der Reihenfolge der Übergabe).

BIOSKEY (−−): Setzt die Tastaturtabellen zurück (auf den Einschaltzustand).

PROTOB (execute typ serial# buffer −−): Erzeugt in **buffer** einen Bootsektor. **serial#** ist eine Seriennummer, bei −1 wird eine Zufallszahl berechnet. Ist **execute**=1, so wird ein ausführbarer Bootsektor erzeugt, bei **execute**=0 nicht. Es muß dann allerdings auch ein wirklich ausführbares Programm im Bootsektor stehen. **typ** steht für den Disktyp (-1 bedeutet nicht verändern):

Bit 0: 0=Single Sided, 1=Double Sided

Bit 1: 0=40 Tracks, 1=80 Tracks (normales ST-Format)

SCRDMP (−−): Löst eine Hardcopy aus (wie Alternate+HELP).

XSETTIME (time date −−): Setzt Datum und Uhrzeit. Werte im TOS-Format. Es wird die Uhrzeit im Tastaturprozessor gesetzt.

XGETTIME (−− time_date): Holt Datum und Uhrzeit aus dem Tastaturprozessor. Dabei ist **date** im Highword, **time** im Low-Word codiert.

IKBDWS (addr count −−): Schickt einen String an den Tastaturprozessor.

JDISINT (interupt# −−): Sperrt einen Interrupt des MFP. Siehe MFPINT.

JENABINT (interupt# −−): Gibt einen Interrupt des MFP frei. Siehe MFPINT.

GIACCESS (reg date −− date): Liest/schreibt ein Register des Soundchips. **reg**-Bit 7 = 1 heißt schreiben. Bedeutung der Register siehe DOSOUND.

OFFGIBIT (nbit −−): Löscht ein Bit im Port A des Soundchips. Dabei bedeutet:

Bit 0: Floppy Seite 0/Seite 1

Bit 1: Laufwerk A anwählen

Bit 2: Laufwerk B anwählen

Bit 3: RTS-Signal für RS 232

Bit 4: CTR-Signal für RS 232

Bit 5: Strobe-Signal für Centronics

Bit 6: Für allgemeine Ausgabe

Bit 7: unbenutzt

ONGIBIT (nbit −−): Setzt ein Bit im Port A des Soundchips.
XBTIMER (addr dat con timer −−): Startet einen MFP-Timer. **timer** geht von 0 bis 3 und steht für Timer A-D:

Timer A: Für Anwender reserviert, Taktrate 2 457 600 = &200 ∗ $3000 Hz.

Timer B: Horizontale Synchronisation

Timer C: System-Timer, Taktrate wie Timer A.

Timer D: kontrolliert Baudrate.

Die Werte von **con** (Control) bedeuten:
0 = Timer aus
1-7 = Vorteiler teilt durch 4/10/16/50/64/100/200
8 = Event Count Mode (nur Timer A, B)
9-15 = Pulsweiten-Mode, Vorteiler 4/10/16/50/64/100/200 (nur A, B)
Für Timer C ist **con** mit 16 zu multiplizieren.
dat ist der Wert, auf den der Timer nach Ablauf gesetzt wird.

DOSOUND (soundstring −−): Spielt eine vorgegebene Klangfolge ab. Die Datenbytes haben folgende Bedeutung:
0-15 + Wert: Register n setzen. Die Register haben folgende Bedeutung:
Register 0 und 1 bestimmen die Periodendauer des Tons von Kanal A
Register 2 und 3 bestimmen die Periodendauer des Tons von Kanal B
Register 4 und 5 bestimmen die Periodendauer des Tons von Kanal C
Register 6 schaltet den Rausch-Generator ein
Register 7 kontrolliert die Vorgänge: Bit 0: Kanal A (gesetzt bedeutet ein)
Bit 1: Kanal B
Bit 2: Kanal C
Bit 3: Rauschgenerator A
Bit 4: Rauschgenerator B
Bit 5: Rauschgenerator C
Bit 6: Ein/Ausgang Port A
Bit 7: Ein/Ausgang Port B
Register 8 bestimmt die Lautstärke von Kanal A
Register 9 bestimmt die Lautstärke von Kanal B
Register 10 bestimmt die Lautstärke von Kanal C
Register 11 und 12 bestimmen die Periodendauer der Hüllkurve
Register 13 bestimmt die Kurvenform der Hüllkurve
Register 14 bildet Port A des Soundchips (für allgemeine Zwecke)

Register 15 bildet Port B des Soundchips (Centronics-Port)

128 + Startwert: Setzt Startwert für Kommando 129

129 + 0-15 + Inc + Endwert: Addiert Inc zum Startwert, schreibt ihn ins Soundregister (0-15) und wiederholt dies alle 1/50 Sekunde, bis der Endwert erreicht wird

255 + Delay: Wartet Delay/50 Sekunden

255 + 0: Ende

SETPTR (6b −−): Stellt Daten für den Drucker (für die Hardcopy) ein. Bietet dieselben Möglichkeiten wie der Teil "Drucker einstellen" des Kontrollfeld-Accessory:

Bit 0: 0=Matrixdrucker, 1=Typenraddrucker

Bit 1: 0=S/W-, 1=Farb-Drucker

Bit 2: 0=Atari-, 1=Epson-Drucker

Bit 3: 0=Draft, 1=Final Quality

Bit 4: 0=Centronics, 1=RS 232

Bit 5: 0=Endlos, 1=Einzelblatt

KBDVBASE (−− tab): Gibt die Adresse auf eine Tabelle der Routinen zurück, die die Werte des Tastaturprozessors auswerten. Die Zeiger haben folgende Andordnung: MIDI Eingabe, Tastatur-Fehler, MIDI-Fehler, IKBD-Status, Maus-Routinen, Uhrzeit-Routine, Joystick-Routine.

PRTBLK (blktab −−): Gibt eine Hardcopy aus. **blktab** ist ein Zeiger auf eine Tabelle mit folgendem Aufbau:

.L Startadresse des zu druckenden Ausschnitts .W Bitoffset zur Startadresse .W Breite des Ausschnitts in Pixeln .W Höhe des Ausschnitts in Pixeln .W Linker Rand .W Rechter Rand .W Bildschirmauflösung (GETREZ) .W Druckerauflösung (Bit 3 von SETPTR) .L Zeiger für Farbpalette .W Druckertyp (Bit 2 von SETPTR) .W Druckerport (Bit 4 von SETPTR) .L Zeiger auf Halbtonmaske (0=Systemhalbtonmaske)

VSYNC (−−): Wartet bis zum nächsten Vertical Blank Interrupt.

BLITMODE (par −− rwert): Fragt ab, ob der Blitter vorhanden ist und schaltet ihn gegebenfalls ein oder aus. Dabei bedeutet **par**:

−1: Abfragen

0: Blitter ausschalten

1: Blitter einschalten

rwert hat folgende Bedeutung:

Bit 0: 1, wenn Blitter eingeschaltet

Bit 1: 1, wenn Blitter vorhanden

Da die XBIOS-Erweiterung BLITMODE die Nummer $40 hat, wird auf alten STs, deren TOS diese Routine noch nicht enthält, $40 zurückgegeben, also sind weder Bit 0, noch Bit 1 gesetzt.

Für Erweiterungen von GEMDOS, BIOS und XBIOS in späteren TOS-Versionen ist ein direkter Aufruf möglich:

GEMDOS (p1 .. pn number n+1 bset −− D0.l): Ruft GEMDOS (Trap #1) mit den Parametern **p1 .. pn number** auf. **number** ist die Nummer der GEMDOS-Routine. **n+1** ist die Zahl der Parameter, inklusive **number**. **bset** ist ein Wortgroßes Bitset, das angibt, ob ein Wert 16 oder 32 Bit groß ist. Dabei steht für jedes Langwort (32 Bit) ein gesetztes Bit. Die Bits werden vom LSB aus gewertet, das LSB steht für **p1**, die weiteren Bits dann für **p2** bis **pn**. Rückgabewert steht in D0, es wird einfach das ganze Register auf den Stack gelegt.

Beispiel: FREAD (addr len hande −− #Bytes / -error) wird durch folgenden Aufruf ersetzt:

```
&63 4 %11 GEMDOS
```

BIOS (p1 .. pn number n+1 bset −− D0.l): Parameter wie GEMODS, ruft aber das BIOS (Trap #13) auf.

XBIOS (p1 .. pn number n+1 bset −− D0.l): Parameter wie GEMDOS und BIOS, ruft das XBIOS (Trap #14) auf.

8 Tools

1. Das Memory Management

1.1. Memory-Management-Theorie

BigFORTH verfügt über ein eigenes Memory Management. Grund der Implementation ist das Memory Management des TOS, das in der Version 1.0 etwa 200 Malloc-Aufrufe zuläßt und dann schlappmacht. Für eine wirklich dynamische Speicherverwaltung ist das TOS ohnehin völlig untauglich, da nichts gegen eine zunehmende Speicherfragmentierung unternommen wird.

In einem Programm gibt es drei Datenarten, die sich in ihrer Belegungsstrategie grundsätzlich unterscheiden: Zum einen die Programmdaten selbst. Dazu gehört nicht nur der Code, sondern z. B. auch die Texte, die ein Programm ausgibt. Allgemein faßt man diese Daten als Resourcen eines Programms zusammen. Sie sind statisch, da sie schon vom Compiler erzeugt werden. Der Platz dafür wird beim Programmstart belegt. Auch für globale Variablen ist der Platz schon belegt.

Zweitens die lokalen Variablen, die während des Programmablaufes Platz brauchen. Sie finden ihn auf dem Stack. Hier hat jedes Wort nach oben Platz, den Stack unten außerhalb seines Einflußbereiches kann es nicht ändern. Stackvariablen sind wirklich nur lokal, haben keine Dauerhaftigkeit.

Für größere Datenmengen wie Strings oder andere Datenstrukturen wird natürlich auch Platz benötigt. Ihn statisch zu reservieren, wäre sicher ein Fehler, denn der vorhandenen Platz soll so gut wie möglich genutzt werden können. Leider gibt es für solche Daten keine vorhersehbare Belegungsstrategie wie für den Stack. Speicheranforderungen und Freigaben werden bei Bedarf vorgenommen. Diese Belegungsstrategie ist "ungeordnet" zu nennen. Dieser Speicherbereich wird deshalb "Heap" (engl. "Haufen") genannt. Er soll im folgenden Memory Heap genannt werden, um Verwechslungen mit dem Wortheader-Heap von bigFORTH zu vermeiden.

Das Programm kann einen Bereich mit einer beliebigen Länge anfordern, er wird im bislang freien Speicherplatz des Memory Heaps reserviert, die Adresse zurückgegeben. Der Bereich ist nun unverrückbar belegt, bis er wieder freigegeben wird.

Nun kann man natürlich auch nicht vorhersehen, wann welcher Teil des Speichers wieder freigegeben wird. Die Folge davon ist eine zunehmende Fragmentierung des Speichers: Teile sind freigegeben, dazwischen wieder ein paar Blöcke belegt. Schließlich kann dann eine Forderung nach einem Speicherbereich irgendwann nicht mehr erfüllt werden, obwohl insgesamt durchaus noch genügend Speicher vorhanden wäre, nur nicht zusammenhängend.

Hier müßte aufgeräumt werden. Allerdings muß der "Besitzer" des Speichers, also das Programm, das ihn angefordert hat, von den Aufräumarbeiten informiert werden. Schließlich hat es ja keine Ahnung von den Vorgängen in der Speicherverwaltung. Diese soll so transparent wie möglich sein.

Die Lösung ist schon bekannt: Man gibt dem Programm nicht die Adresse des angeforderten Blocks zurück, sondern einen Zeiger auf diese Adresse, ein "Handle" oder einen "Master Pointer". Der Ort dieses Zeigers bleibt fest, es ist auch kein Problem, ihn zu "recyclen", er hat ja eine konstante Länge. Der Block, auf den dieser Zeiger deutet,

kann beliebig verschoben werden, damit können alle Lücken zusammengefügt werden. Das Problem der Fragmentierung ist gelöst.

Der Aufräumprozeß wird "Garbage Collection" genannt (engl. "Müllabfuhr", Abkürzung GC). Es ist nicht wünschenswert, daß man von einer GC überrascht wird, da das System dann eine deutlich merkbare Zeit stehenbleibt (sekundenlang), bis es weitermachen kann. Deshalb sollte die GC im Hintergrund laufen. Auch dieses Konzept kann in bigFORTH verwirklicht werden, ein eigener Task kümmert sich darum. Er geht den Heap zyklisch durch und sammelt dabei alle freien Teile ein. Dadurch wird erreicht, daß in den meisten Fällen sofort der benötigte Speicher belegt werden kann — auch das ist ein notwendiger Aspekt, da FORTH ja den Anspruch erhebt, realtimefähig zu sein.

Das Memory Management von bigFORTH lehnt sich stark an dem des MacIntoshs an. Die Namen sind dieselben wie die entsprechenden Toolbox-Routinen. Der interne Aufbau ist natürlich anders als beim Mac. Zudem wurde noch eine Verbesserung implementiert: Nichtverschiebbare Blöcke werden von "unten" (niedrigen Adressen) angelegt, verschiebbare von oben. Damit wird das Problem umgangen, daß beim Mac die nichtverschiebbaren Blöcke wie Klippen im Heap liegen und die Garbage Collection bei ihrer Arbeit behindern.

Die festen Blöcke werden nach einem Fit-First-Algorithmus belegt (Fit-First: Was zuerst paßt, wird genommen). Damit wird gewährleistet, daß freie Stellen bald wieder aufgefüllt werden, die verschiebbaren werden aus dem freien Pool zwischen festen und verschiebbaren Blöcken genommen, das geht schneller. Es wird dem Benutzer nahegelegt, feste Blöcke nicht mehr freizugeben, sondern statisch zu benutzen. Der Memory Manager belegt selbst feste Blöcke, um Platz für die Master Pointers zu bekommen und gibt diese auch nicht mehr zurück.

In Anlehnung an den Mac gibt es auch ein "Virtual Memory", das den Inhalt von Dateien im Speicher abbildet. Diese Blöcke können bei Platzbedarf aus dem Speicher gelöscht werden. In bigFORTH wird dieser Teil benutzt, um das Blockkonzept zu inplementieren. Das VM ist eigentlich mächtiger, es kann nämlich ein beliebiger Teil einer Datei in einem Block stehen. Zu jedem dieser löschbaren (purgeable) Blöcke gehört eine Struktur, die PurgeInfo-Struktur. Diese Struktur ist als verkettete Liste verbunden, deren Start in PREV gespeichert ist.

1.2. Internes

Jeder Block beginnt mit einer Längenangabe (Langwort), danach folgt der Zeiger auf den Master Pointer, dahinter der freie Bereich und als Abschluß wieder ein Langwort, das nochmals die Länge enthält. Damit kann man sich von vorn nach hinten und von hinten nach vorn durch den Memory Heap durchhangeln. Als Anfang und Ende des Heaps (Vor HEAPSTART und hinter HEAPEND) steht ein 0-Langwort, aus dem zweifelsfrei ersichtlich ist, daß Anfang bzw. Ende des Heap erreicht ist, denn der kürzestmögliche Block ist 12 Bytes lang und besteht dann nur aus Verwaltungsinformationen.

Bei freien Blöcken zeigt der Rückzeiger nicht auf den MP, sondern auf Nil. Feste Blöcke haben einen Zeiger auf -1, geLOCKte Blöcke (vorrübergehend "festgenagelt") haben einen negativen Zeiger, dessen Betrag auf ihren MP zeigt.

Blöcke vorrübergehend "festnageln" ist durchaus sinnvoll. Auf einen Block darf ja nur zugegriffen werden, wenn man weiß, daß er sich während des Zugriffs nicht verschiebt. Solange diese Gefahr besteht, darf man auf ihn nur über den MP zugreifen. Dieser Umweg ist manchmal nicht möglich, manchmal nicht erwünscht. Dann nagelt man den Block fest, seine Lage ist vom Memory Manager nicht mehr veränderbar. Auch die löschbaren (purgeable) Blöcke des VM können im festgenagelten Zustand nicht gelöscht werden.

Bei löschbaren Blöcken zeigt der Rückzeiger nicht auf den MP, sondern auf die Purgeinfo. Deren erste Zelle ist die Listenkette, der zweite der eigentliche Rückzeiger auf den MP, dahinter steht der FCB, die Position und Länge in der Datei, deren Inhalt in dem Block steht und zuletzt ein Wort-Feld, in dem die Update-Flag steht. Bei ihr ist nur das MSB signifikant, der Rest könnte für andere Zwecke verwendet werden.

1.3. Die Befehle

MEMORY (−−) (VS voc −− MEMORY): Die Wörter des Memory Managers befinden sich im Vokabular MEMORY. Alle weiteren Wörter sind in diesem Vokabular zu finden.

HEAPSTART (−− addr): Adresse des ersten Blocks im Heap.

HEAPEND (−− addr): Endadresse des letzten Blocks im Heap.

HEAPSEM (−− addr): Semaphor. Wenn HEAPSEM locked ist, darf kein anderer Task den Heap verändern. Man kann dann sicher sein, daß der Heap so bleibt, wie er ist, vorrausgesetzt, man benutzt nicht selbst den Memory Manager.

SHIFT? (−− addr): Semaphor. Der Garbage Collector setzt SHIFT? während eines Durchgangs für sich locked. Will man den Durchgang abwarten, muß man SHIFT? für sich selbst locken und gleich wieder freigeben (SHIFT? UNLOCK).

FULL? (block −− flag): Gibt 0 zurück, wenn der Block mit der Startadresse der Verwaltungsinformation **block** frei ist, sonst den Rückzeiger. Etwas schneller als 4+ @.

PREVBLOCK (block −− prevblock): Gibt die Adresse des Blocks, der vor **block** steht, zurück.

NEXTBLOCK (block −− nextblock): Gibt die Adresse des Blocks, der nach **block** steht, zurück.

MEMERR (−− addr): In dieser Variable steht im Fehlerfall die Fehlernummer, sonst 0. Die Bedeutung der Nummern:

1: "**Kein Speicher mehr frei**"
2: "**Keine gültige Adresse**"
3: "**Kein gültiges Handle**"
4: "**SetPtrSize nicht möglich**"

MEMERR$ (−− addr): Enthält die Strings für die Fehlermeldung. An die Stringadresse kommt man mit der Sequenz MEMERR$ MEMERR @ 0 ?DO COUNT + LOOP.

.MEMERR (−−): Deferred Word. Dient zur unmittelbaren Fehlerausgabe.

?MEMERR (−−): Gibt die Fehlermeldung aus und löscht MEMERR. ?MEMERR hängt in .MEMERR.

DISKDISPOSE (−− addr): FindMP speichert hier die Adresse eines neu angelegten Blockes, der noch geladen werden muß. Tritt während des Ladens ein Fehler auf, so muß der Block von DISKERR mit DISPOSHANDLE wieder zurückgegeben werden, da sonst später die Ungültigkeit des Puffers nicht festgestellt werden kann. In dem Wort, das in DISKERR hängt, muß folgende Zeile stehen:

```
DiskDispose @ ?dup  IF  DisposHandle DiskDispose off  THEN
```

GETMP (addr −− MP/0): Findet aus der Blockadresse den MP heraus. Gibt es keinen, wird 0 zurückgegeben. Beispiel: Um den MP eines Disketten-Blocks zu bekommen, muß man `<n> BLOCK GETMP` aufrufen.

SHIFT>ALL (−−): Komplette Garbage Collection.

INITHEAP (len −−): Legt einen neuen Heap mit der Länge **len** an. Es kann nur ein Heap auf einmal vorhanden sein, der alte muß also zurückgegeben worden sein.

NOHEAP (−−): Gibt den Heap zurück.

PUSHHEAP (−−): Tut so, als wäre der Heap zurückgegeben, wirkt sich also auf den Systembereich genauso wie NOHEAP aus. Nach Verlassen des Aufrufers von PUSHHEAP ist der Heap wieder da. PUSHHEAP wird von SAVESYSTEM benutzt.

FREEMEM (−− len): Gibt die Länge des freien Pools zwischen festen und verschiebbaren Blöcken zurück.

MAXMEM (−− len): Löst die Garbage Collection aus und gibt anschließend die Länge des größten freien Blocks zurück.

MOREMASTERS (−−): Legt 512 neue Master Pointer an.

MOREPURGEINFOS (−−): Legt 128 neue PurgeInfos an.

NEWPTR (len −− Ptr): Legt einen nicht verschiebbaren Block der Länge **len** an der Adresse **Ptr** an. Der Inhalt ist vorerst undefiniert.

DISPOSPTR (Ptr −−): Gibt den nichtverschiebbaren Block **Ptr** zurück.

NEWHANDLE (len −− MP): Legt einen Block der Länge **len** an und weist ihm das Handle **MP** zu. Der Block ist verschiebbar, es darf also nur über das Handle zugegriffen werden.

(NEWHANDLE (MP len −−): Legt einen verschiebbaren Block der Länge **len** an und speichert seine Adresse in **MP**. Dieser Befehl dient der Zuweisung von Blöcken an Variablen oder Strukturen.

DISPOSHANDLE (MP −−): Gibt den dem Handle **MP** zugewiesenen Block und das Handle zurück.

EMPTYMP (MP −−): Wie DISPOSHANDLE, nur wird ein Purgeable-Block gesichert, wenn seine Update-Flag gesetzt ist.

GETPTRSIZE (Ptr −− len): Gibt die Länge des für den Block **Ptr** reservierten Platz zurück. Dies kann unter Umständen mehr sein, als ursprünglich reserviert wurde.

SETPTRSIZE (Ptr len −−): Setzt die Länge des Blocks **Ptr** auf die Länge **len**. Wachstum ist nur möglich, wenn hinter **Ptr** genügend freier Platz ist. Wird um weniger als 12 Bytes geschrumpft, wirkt sich das auf den reservierten Platz nicht aus.

GETHANDLESIZE (MP −− len): Wie GETPTRSIZE, nur für Handles.

SETHANDLESIZE (MP len −−): Wie SETPTRSIZE, nur für Handles. Wachstum ist immer möglich, wenn noch Platz frei ist.

HLOCK (MP −−): "Nagelt" den Block "fest", der am Handle **MP** "hängt". Er ist dann nicht verschiebbar.

HUNLOCK (MP −−): Hebt das Lock auf den Block auf, der am Handle **MP** hängt.

HPURGE (file pos len MP −−): Macht den Block **MP** purgeable (löschbar). Dazu müssen ihm die Datei **file**, Position **pos** und Länge **len** zugewiesen werden.

PURGE@ (MP −− file pos len / 0): Liest die Purgeinfo aus. Ist sie nicht vorhanden, wird 0 zurückgegeben.

HNOPURGE (MP −−): Hebt die Eigenschaft "Purgeable" des Blocks **MP** auf. Die zugehörige PurgeInfo wird freigegeben.

HUPDATE (MP −−): Setzt die Update-Flag des purgeablen Blocks **MP**. Ist er nicht purgeable, geschieht nichts.

BACKUPMP (MP −−): Sichert **MP**, wenn er purgeable und die Update-Flag gesetzt ist.

FINDMP (file pos len −− MP): Sucht den purgeable Block der Datei **file** ab Position **pos** mit der Länge **len**, ist er nicht vorhanden, wird ein entsprechend langer Speicherbereich angelegt, nachgeladen und der **MP** zurückgegeben.

Die folgenden Wörter sind nicht im Kernel definiert, sondern in FILEINT.SCR:

HANDTOHAND (MP1 −− MP2): Verdoppelt den Block am Handle **MP1** und gibt das Handle des zweiten Blocks **MP2** zurück.

PTRTOHAND (Ptr −− MP): Kopiert den festen Block an der Adresse **Ptr** und gibt das Handle der Kopie **MP** zurück.

PTRTOXHAND (Ptr MP −−): Weist dem Handle **MP** eine Kopie des Inhalts des festen Blocks **Ptr** zu. Der vorherige Block an **MP** wird freigegeben.

HANDANDHAND (MP1 MP2 −−): Hängt den Inhalt vom Block **MP1** hinten an Block **MP2** an.

PTRANDHAND (Ptr MP −−): Hängt den Inhalt von Block **Ptr** an Block **MP** hinten an.

.HEAP (−−): Heapdump. Es wird bei Heapstart begonnen, Startadresse, Länge, bei verschiebbaren Blöcken das Handle, bei purgeablen Dateiname, Position und Länge und ein "x" für gesetzte Update-Flag, bei festen Blöcken noch ein "locked" ausgegeben. .HEAP läßt sich mit ⟨Ctrl⟩⟨C⟩ oder ⟨Esc⟩ abbrechen, mit jedem anderen Tastendruck unterbrechen und wieder fortsetzen.

.BLOCKS (−−): Blockpufferdump. Geht von PREV die Blöcke der Reihe nach durch, gibt Datenadresse, Datei, Blocknummer (pos/len) und ein "updated" für gesetzte Update-Flag aus. .BLOCKS kann ebenfalls mit ⟨Ctrl⟩⟨C⟩ und ⟨Esc⟩ abgebrochen, mit jedem anderen Tastendruck unterbrochen und fortgesetzt werden.

SHIFTTASK (−− Taddr): Garbage Collection Task.

DOSHIFT (−−): Startet die Garbage Collection im Hintergrund.

2. SAVESYSTEM

Im Gegensatz zu anderen Compilern produziert der FORTH-Compiler ausschließ- lich direkt ausführbaren Code, keine von Diskette startbaren Programme. Um zu einem von Diskette startbaren System ("Image") oder einer eigenen Applikation zu kommen, muß man das System nach beendeter Compilation mit SAVESYSTEM sichern.

SAVESYS.SCR (−−): Aus dieser Datei wird SAVESYSTEM geladen.

(SAVESYS (start len handle −− #Bytes / -error): Relokiert das System (von **start len** Bytes) auf Adresse 0. Es steht dann so im Speicher, wie es gesichert werden soll und ist in diesem Zustand nicht mehr lauffähig. Danach wird es in der Datei **handle** gesichert. Anschließend wird an die Adresse RELOZ gesprungen, um das System wieder lauffähig zu machen. Zurückgegeben wird die Anzahl der geschriebenen Bytes oder eine Fehlernummer.

'SAVE (−−): Deferred Word. Wird von SAVESYSTEM vor dem eigentlichen Sichern aufgerufen. Hier werden noch nötige Aufräumarbeiten durchgeführt.

(SAVE (−−): Hängt in 'SAVE. Führt wichtige Aufräumarbeiten durch. Der HEAP wird mit PUSHHEAP ungültig gemacht etc. Ein später dazudefiniertes (SAVE hat üblicherweise folgenden Aufbau:

```
: (SAVE ( -- ) r> {<Word> } (SAVE >r ;
```

(SAVE soll die zurückzusetzende Werte mit PUSH o. ä. sichern, damit nach dem Ende von SAVESYSTEM dieselbe Situation wie vor dem Aufruf vorgefunden wird.

SAVESYSTEM (−−) ⟨*Name*⟩: Sichert das System in der Datei ⟨*Name*⟩. Alle Einstellungen bleiben erhalten. Das System ist in der Regel (soweit kein Fehler gemacht wurde) auch an einer anderen Adresse startbar. Strukturen außerhalb des Bereichs FORTHstart bis HERE und der Userarea sind nicht dauerhaft, Wörter, die darauf zugreifen, müssen erkennen, daß sie nach dem Neustart nicht mehr vorhanden sind.

Dazu muß ein entsprechendes Wort (SAVE in 'SAVE eingehängt werden, das diese Strukturen als ungültig markiert.

GOODBYE (−−): Bereitet das System so auf, wie es nach dem Laden im Speicher steht, mit einem Unterschied: Es ist schon relokiert. Danach wird mit BYE das System verlassen. Auch GOODBYE ruft 'SAVE auf. GOODBYE wird von RELOCATE.PRG benutzt, um ein sicherbares Image von bigFORTH im Speicher vorzufinden.

3. Strings

STRINGS.SCR (−−): Diese Datei enthält weitere String-Befehle.

CAPS (−− addr): Schalter. Wenn CAPS gesetzt ist, werden beim Vergleich mit COMPARE Groß- und Kleinbuchstaben nicht unterschieden. Ist CAPS gelöscht, findet die Unterscheidung statt.

−TEXT (addr1 len addr2 −− -n / 0 / n): Stringvergleich. Stimmen **len** Bytes ab **addr1** und **addr2** überein, so wird 0 zurückgegeben. Ansonsten wird die Differenz der ersten nicht übereinstimmenden Werte zurückgegeben, eine negative Zahl bedeutet, daß der Text an **addr1** "kleiner" ist, eine positive, daß der Text an **addr2** "kleiner" ist.

COMPARE (addr1 len addr2 −− -n / 0 / n): Stringvergleich. Parameter wie -TEXT, COMPARE wertet aber CAPS aus. Groß- und Kleinbuchstaben werden nicht unterschieden, wenn CAPS on ist. Des weiteren gilt: Ä=AE, Ö=OE, Ü=UE und ß=SS. Dadurch ist eine Sortierung wie z. B. im Telefonbuch möglich.

SEARCH (text textlen buf buflen −− offset flag): Sucht im Puffer **buf buflen** den String **text textlen**. Bei Erfolg wird die relative Position im Puffer **offset** und true zurückgegeben, sonst **buflen-textlen** und false.

DELETE (buffer size count −−): Löscht **count** Bytes in einem **size** großen Puffer. Der Pufferinhalt wird nach vorne geschoben, von hinten her werden **count** Bytes mit Leerzeichen aufgefüllt.

INSERT (text len buffer size −−): Der String **text len** wird in den Puffer eingefügt. Der Pufferinhalt wird nach hinten geschoben, **len** Bytes am Pufferende "fallen hinten hinaus".

REPLACE (text len buffer size −−): Der String **text len** wird an den Anfang des Puffers geschrieben und überschreibt die dort stehenden Bytes.

$SUM (−− addr): In dieser Variable wird die Adresse des Summenstrings gespeichert.

$ADD (addr count −−): Addiert den String **addr count** zum Summenstring. Der String wird hinten angehängt und das Countbyte des Summenstrings um **count** erhöht.

Anwendungsbeispiel:

```
pad ␣$sum␣ !␣pad␣off⟨RET⟩  ok
"␣Dies␣ist"␣count␣$add⟨RET⟩  ok
"␣␣ein␣Text"␣count␣$add␣"␣."␣count␣$add⟨RET⟩  ok
pad␣count␣type⟨RET⟩ Dies ist ein Text. ok
```

C>0" (addr −−): Wandelt einen counted String in einen 0-terminated String.

0>C" (addr −−): Wandelt einen 0-terminated String in einen counted String.

CPUSH (addr len −−): Sichert den Puffer **addr len** auf dem Returnstack. Wie bei PUSH wird er beim Verlassen des Wortes wiederhergestellt.

4. Der Disassembler

Der Disassembler wandelt die Befehle in Motorola-Mnemonics. Es wird also im üblichen Format ausgegeben:

```
nnnnnn:   CCCCPPPPPPPPPPPPPPPP  opcode  ea[,ea]
```

nnnnnn ist die Adresse, CCCC der Code hexadezimal, danach folgen ggf. weitere Hexwörter für die Parameter, danach Opcode und Adressierungsart.

DISASS.STR (−−): Aus dieser Datei wird der Disassembler geladen.

(DISLINE (−−): Disassembliert eine Zeile (ohne Ausgabe der Adresse).

ADDR! (addr −−): Speichert die Startadresse zum Disassemblieren mit (DISLINE.

DIS (addr −−): Disassembliert ab **addr**. Das Listing kann mit ⟨Esc⟩ und ⟨Ctrl⟩⟨C⟩ gestoppt, mit jeder anderen Taste angehalten und fortgesetzt werden.

DISW (−−) ⟨*Word*⟩: Disassembliert ⟨***Word***⟩. Bei jedem RTS wird auf einen Tastendruck gewartet, man kann hier mit ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ die Ausgabe beenden. Ansonsten gilt dasselbe wie für DIS.

DISLINE (addr −− addr'): Disassembliert den Befehl an **addr** und gibt die Adresse des nächsten Befehls zurück.

5. Decompiler

Bekanntlich wird ein großer Teil der Arbeit bei der Softwareentwicklung für Wartung und Fehlerkorrektur aufgewendet. Ein Debugger ist somit ein sehr wichtiges Werkzeug. Viele Debugger erlauben nur, das ablauffähige Programm, so wie es der Compiler erzeugt, zu verfolgen — also Maschinencodedebugging. Source Level Debugger, die den auszuführenden Befehl und seine Auswirkung zeigen, sind für Sprachen wie C oder Modula rar, teuer und noch nicht lange erhältlich — zumindest auf kleinen Systemen wie dem Atari ST.

FORTH-83 ist decompilierbar. Aus den Adressen kann man die Namen der Routinen ermitteln. Auch die Auswirkungen lassen sich leicht zeigen, im wesentlichen muß man nur einen Stackdump ausgeben.

Dagegen erzeugt bigFORTH optimierten Maschinencode — wie ein moderner C- oder Modula-Compiler. Ist bigFORTH auch decompilierbar? Ja, auch hier kann man die Herkunft des Codes rekonstruieren. Unterprogrammaufrufe mit jsr oder bsr lassen sich ähnlich einfach wie bei FORTH-83 decompilieren.

Makros bereiten wesentlich mehr Schwierigkeiten. Sämtliche Makros müssen mit dem Codesegment verglichen werden. Das Ende eines Makros kann durch die Optimierungen weggefallen sein — an seiner Stelle steht dann ein Codestück, aus dem der Übergang rekonstruierbar ist — dieses Codestück kann auch leer sein. Diesen Übergang muß man auswerten, denn aus ihm muß geschlossen werden, wie das nächste Macro anfängt. Da ein "Backtracking" in FORTH leider nur sehr schwer verwirklichbar ist, muß im ersten Durchgang das richtige Wort gefunden werden, das ist nicht immer möglich, deshalb wird nicht immer korrekt decompiliert.

Aufsetzpunkte sind die Stellen, die auch per Programm angesprungen werden können. Dazwischen wird solange Assembler ausgegeben, bis wieder ein Aufsetztpunkt gefunden wird. Auch nicht mehr decompilierbare Stellen werden als Assembler übersetzt, ebenso wie Code-Wörter, wenn sie nicht Makros sind.

Der Debugger erlaubt das Tracen eines Wortes, es können beliebig viele Breakpoints gesetzt werden. Allerdings müssen diese beim Compilieren erzeugt werden, während das Debuggen eines Wortes keine Codeänderungen erfordert.

Der Tracer beruht auf dem Trace-Modus des 68000. Nach der Ausführung eines Befehls wird eine Exception ausgelöst und der Trace-Vektor angesprungen. Dadurch wird die Ausführung der FORTH-Befehle etwa um den Faktor 10 verlangsamt. "Debugging" heißt wörtlich übersetzt "Entwanzen". Historischer Grund dieses Wortes soll ein Fehler in einer Computeranlage sein, dessen Ursache Schaben im Rechner gewesen waren. "Tracing" heißt "(eine Spur) verfolgen". Man versteht darunter die schrittweise Ausführung eines Programms. Nach jedem Schritt werden ein paar Informationen ausgegeben und eine Möglichkeit offengehalten, den Ablauf zumindest zu stoppen.

TOOLS.SCR (−−): Aus dieser Datei wird der Decompiler und der Tracer geladen.

TOOLS (−−) (VS voc −− TOOLS): Für die Wörter des Decompilers und des Tracers wird ein eigenes Vokabular angelegt.

SEE (−−) ⟨*Word*⟩: Decompiliert ⟨***Word***⟩. Es wird dabei versucht, so nahe wie möglich an den ursprünglichen Source heranzukommen. SEE erzeugt teilweise recompilierbaren Code. Variablen, Konstanten, User-Variablen, Vokabulare und Dateien werden erkannt, der Inhalt wird auch ausgegeben. Beim Decompilieren von Colon-Wörtern und Code-Wörtern kann mit ⟨Esc⟩ und ⟨Ctrl⟩⟨C⟩ abgebrochen werden, jeder andere Taste unterbricht und setzt wieder fort.

D' (−−) ⟨*Word*⟩: Decompiliert ⟨***Word***⟩, dabei wird ein Code erzeugt, wie ihn der Tracer ausgibt. Er entspricht in etwa einem traditionellen Disassemblerlisting: `nnnnnn: WORD` , wobei nnnnnn die Adresse ist, WORD das Ergebnis der Decompilation. Ebenso wie SEE kann D' mit ⟨Esc⟩ und ⟨Ctrl⟩⟨C⟩ abgebrochen und mit jeder anderen Taste unterbrochen und fortgesetzt werden.

DUMP (addr len −−): Gibt einen Dump aus. In jeder Zeile werden 16 Bytes gelistet. **addr** wird zu diesem Zweck auf die nächste durch 16 teilbare Zahl abgerundet. Die eigentliche Startposition wird durch "" (Backslash-Slash) und "V" in der Titelzeile markiert. Es wird sowohl eine Hex- als auch ein ASCII-Dump ausgegeben. Ganz links wird die Adresse der Zeile ausgegeben. In der Titelzeile steht die Nummer der Bytes. DUMP kann mit ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ abgebrochen, mit jeder anderen Taste unterbrochen und fortgesetzt werden.

DU (addr −− addr+$40): Gibt einen Dump über $40=64 Bytes aus. **addr** wird um diese $40 Bytes erhöht zurückgegeben.

DL (line# −−): Gibt einen Dump über die Zeile **line#** des aktuellen Screens aus.

.VOCS (−−): Gibt die Liste aller Vokabulare aus.

Die folgenden Wörter befinden sich im Vokabular TOOLS:

((SEE (cfa −−): Decompiliert **cfa**. Sonst wie SEE.

N (IP −− IP'): Decompiliert eine Zeile wie D'. N hat einen Nebeneffekt: Es speichert die Informationen, um an **IP'** aufzusetzen. Aus diesem Grund kann N nur dann zur stückweise Decompilation eingesetzt werden, wenn nur an Aufsetzpunkten abgebrochen wird. Der Tracer benutzt N in der hier geforderten Weise.

S (IP −− IP'): Stringdump. Gibt die Adresse, die Länge und den String aus. Format: `nnnnnn: len String`

D (addr n −− addr+n): Gibt einen Dump von **n** Bytes ab **addr** aus. Zuerst wird die Adresse ausgegeben, dann die **n** Bytes (rechtsbündig in einem je 3 Zeichen großen Feld), und schließlich die **n** Bytes als ASCII-Zeichen.

C (addr −− addr+1): Wie 1 D. Gibt einen Dump von einem Byte aus.

? (addr −−): Gibt den Inhalt von **addr** in einem 9 Zeichen großen Feld aus.

Nun zum Debugger, die wichtigsten Wörter sind wieder im Vokabular FORTH:

>DEBUG (cfa −−): Schaltet den Debugger ein. Wird **cfa** später aufgerufen, wird es schrittweise abgearbeitet.

DEBUG (−−) ⟨*Word*⟩: Schaltet den Debugger für ⟨***Word***⟩ ein. Ansonsten wie >DEBUG.

TRACE' (⟨*input*⟩ −− ⟨*output*⟩) ⟨*Word*⟩: ⟨***Word***⟩ wird schrittweise abgearbeitet. Der nächste abzuarbeitende Befehl wird mit N angezeigt, es wird mit .DUMP gewöhnlich ein Stackdump ausgegeben, vom Benutzer kann eine Zeile eingegeben werden, die dann interpretiert wird. Schließlich wird der Befehl ausgeführt. Nach Beendung des Wortes wird der Debugger wieder ausgeschaltet.

BP: (−−) ⟨*Breakpoint*⟩ immediate restrict: Erzeugt einen vorerst inaktiven Breakpoint. Der Breakpoint selbst wird auf dem Heap erzeugt, im Programm steht ein jsr zu dem Wort ⟨***Breakpoint***⟩. Solange Breakpoints eingesetzt werden, darf der Heap nicht mit CLEAR gelöscht werden.

BP'ON (−−) ⟨*Breakpoint*⟩: Aktiviert einen Breakpoint. Wird er erreicht, so wird der Debugger für das Wort, aus dem er aufgerufen wurde, eingeschaltet.

BP'OFF (−−) ⟨*Breakpoint*⟩: Deaktiviert einen Breakpoint.

Die folgenden Wörter befinden sich im Vokabular TOOLS:

MACRO> (−− addr): Hier wird die Adresse des nächsten zu tracenden Befehls gespeichert.

MACRO>! (addr −−): Speichert die Adresse des nächsten zu tracenden Befehls. 0 MACRO>! liefert einen Trick: Es können auch Makros getracet werden, da MACRO>! nur dann **addr** in MACRO> speichert, wenn dort nicht 0 steht. Der Debugger hält dann bei jedem Assemblerbefehl an. Um diesen Zustand wieder rückgängig zu machen, ist auch MACRO> sichtbar. Mit MACRO> ON werden wieder alle Assemblerbefehle eines Makros übersprungen.

TD0 (−− addr): Hier werden die Register D0-A4 und der SR gespeichert. Die Register sind einzeln ansprechbar, sie können wie Variablen behandelt (und natürlich auch geändert) werden:

D0 D1 D2 D3 D4 D5 D6 D7 A0 A1 A2 A3 A4 SR (−− addr): Die Adressen der gesicherten Register. Alle Register außer SR (16 Bit) sind 32-Bit-Werte.

#REGS (−− n): Gibt die Länge des Registerfelds ($36=54 Bytes).

.SR (−−): Gibt das Statusregister (SR) aus. Format:

```
T-S--210---XNZVC
x0x00xxx000xxxxx
```

Die Belegung wird binär ausgegeben. Die Binärzahl wird immer direkt unter der Titelzeile ("T-S−−210−−-XNZVC") ausgegeben.

.DUMP (−−): Deferred Word. Gibt den Dump bei jedem Trace-Schritt aus. Standardbelegung: .S.

DUMPREGS (−−): Gibt alle gesicherten Register, SR und einen Stackdump aus. Kann alternativ zu .S in .DUMP eingehängt werden, wenn Code-Wörter debugged werden müssen.

@TOS (−− addr): In dieser Variable steht der Modus, mit dem das letzte Makro seinen Wert auf den Stack gelegt hätte, wäre es nicht verkürzt worden.

DO–TRACE (−−): Schaltet den Debugger ein. Das Trace-Bit im SR wird gesetzt.

END–TRACE (−−): Schaltet den Debugger aus. Es wird auch das Tracebit im gespeicherten SR auf 0 gesetzt. So kann man aus dem Debugger aussteigen und ein gerade getracetes Wort mit normaler Geschwindigkeit zu Ende laufen lassen.

UNBUG (−−): Notbremse: Schaltet den Debugger aus, macht alle Änderungen rückgängig und beendet die Ausführung des gerade getraceten Worts. Der Stack wird auch gelöscht. UNBUG wird beim Erkennen des Fehlers eingesetzt.

GO (−−): Die Benutzerinteraktion beim Tracen eines Wortes wird abgeschaltet. Man muß nicht mehr zumindest RET drücken, um den nächsten Befehl auszuführen. Die Befehle laufen durch, mit ⟨Esc⟩ oder ⟨Ctrl⟩⟨C⟩ kann wieder in den normalen Zustand zurückgeschaltet werden, mit jeder anderen Taste kann angehalten und fortgesetzt werden.

ENDLOOP (−−): Dient zum Überspringen von fehlerfreien Schleifen. Beim Erreichen des Schleifenende-Befehls wird ENDLOOP eingegeben, die Ausgabe des Tracers wird erst wieder eingeschaltet, wenn die Schleife beendet wurde.

NEST (−−): Schaltet bei einem Unterprogrammaufruf den Debugger auf das Unterprogramm um. Es können damit Programmablaufwege in hierarchischen Programmen verfolgt werden. Wird das Unterprogramm beendet, wird wieder zurück auf das aufrufende Wort geschaltet.

NESTALL (−−): Schaltet automatisch bei jedem nächsten Unterprogrammaufruf auf das Unterprogramm um. Es werden dann alle Unterprogramme getracet.

NONEST (−−): Schaltet das automatische Tracen von Unterprogrammen wieder ab.

UNNEST (−−): Beendet das Tracen des aktuellen Wortes. Der Debugger wird nicht abgeschaltet, bei der Rückkehr zum aufrufenden Wort wird wieder weitergetracet.

6. Der Tasker

bigFORTH ist multitaskingfähig. Die Basiswörter sind schon im Kernel definiert. Doch PAUSE im Kernel ist noch wirkungslos (PAUSE ist ein deferred Word). Der eigentliche Tasker muß nachgeladen werden. Jeder Task hat seine eigene Task Area. Sie besteht aus HERE, PAD, Stack, User Area und Returnstack. Konflikte mit anderen Tasks sind weitgehend ausgeschlossen. Es muß natürlich darauf geachtet werden, daß keine “dirty” Variablen verwendet werden (Lokale Variablen, die nicht auf dem Stack, sondern an einer festen Adresse im Hauptspeicher liegen). Außerdem muß die Zugriffsberechtigung auf nicht teilbare Resourcen mit Semaphoren geregelt sein.

Ein Semaphor ist ein Schloß, das Zugriffe auf bestimmte Resourcen regelt (im täglichen Leben ist das Schloß am stillen Örtchen ein häufig benutztes Semaphor). Semaphore werden gesetzt, wenn der Task seine Ruhe braucht. Während der Floppycontroller seine Daten überträgt darf z. B. kein anderer Task auf ihn zugreifen — sonst wäre die Datenübertragung sehr gefährtet. Auch der Memory Manager kann nur hinter Schloß und Riegel seinen Speicher umbauen, ein Zugriff während des Umbaus würde ins Leere gehen.

Tasks laufen nicht von alleine los, sie müssen “initiiert” werden. Hier verzweigt das Programm in zwei Äste, die quasi gleichzeitig und voneinander unabhängig laufen. Genauer: Ein Wort kann sich von einem Task in den nächsten “schieben”, der Aufrufer dieses Wortes kann weitermachen, obwohl das Wort seine Arbeit nicht beendet hat.

Tasks haben ihre eigene Fehlerbehandlung. Die Fehlermeldung wird in der letzten Zeile ausgegeben, “Task Error:” wird davorgesetzt und ein Signalton ertönt. Der Task wird nach einem Fehler angehalten.

TASKER.SCR (−−): Aus dieser Datei wird der Tasker nachgeladen.

STOP (−−): Hält den aktuellen Task an. Er ist dann im "schlafenden" Zustand und kann von einem anderen Task an der Stelle hinter STOP wieder geweckt werden.

SINGLETASK (−−): Schaltet auf Singletasking um. PAUSE wechselt nicht mehr aus dem aktuellen Task. Auf Singletasking kann in kritischen Situationen umgeschaltet werden, wenn keine vernünftige Semaphor-Strategie zur Verfügung steht.

MULTITASK (−−): Schaltet auf Multitasking. PAUSE schaltet wieder in einen anderen Task um.

ACTIVATE (Taddr −−): Aktiviert den Task **Taddr**. Das Wort, in dem ACTIVATE steht, wird im Task **Taddr** weiter ausgeführt, im initiierenden Task wird zum Aufrufer dieses Wortes zurückgekehrt.

PASS (n1 .. nm m Taddr −−) (−− n1 .. nm): Aktiviert den Task **Taddr**. Es werden die **m** Parameter **n1 .. nm** vom Stack des initiierenden Tasks genommen und im gestarteten Task auf den Stack gelegt. Sonst wie ACTIVATE.

AUTOSTART (Taddr −− Taddr): Beim Neustart des Systems können Tasks nur kontrolliert neugestartet werden. Es wird daher die in TSTART gespeicherte Adresse mit der Taskadresse aufgerufen. Das Wort AUTOSTART setzt nun TSTART des zu aktivierenden Tasks mit der Returnadresse, die es auf dem Returnstack findet. Dadurch wird beim Systemstart das Wort nach AUTOSTART aufgerufen.

Typische Anwendung:

```
<Taddr> AUTOSTART ACTIVATE
```

SLEEP (Taddr −−): Deaktiviert den Task **Taddr**. Er wird bei einem Taskwechsel übersprungen und ist damit im schlafenden Zustand.

WAKE (Taddr −−): Weckt den Task **Taddr**. Er wird an der Stellte fortgesetzt, an der er mit SLEEP von außen oder STOP von innen angehalten wurde.

TIMER@ (−− timer): Liest den 200 Hz-Zählers des Systems aus. **timer** ist die Anzahl der Timer-Interrupts seit dem Einschalten des Rechners.

SYNCTIME (−− useraddr): In dieser Uservariable wird der Zählerstand gespeichert, bei dem der Task nach einem Aufruf von SYNC fortgesetzt wird.

SYNC! (millisec −−): Teil 1 der zeitlichen Tasksynchronisation. Es wird in SYNCTIME die Zeit zum Wiederanlauf gespeichert. Die Millisekunden werden auf den nächsten durch 5 teilbaren Wert aufgerundet, da mit den Systemtimer nur 200stel Sekunden gezählt werden können.

SYNC (−−): Wartet, bis der Systemtimer den Stand von SYNCTIME erreicht hat und läßt dann den Task weiterlaufen.

Dieses System der Zeitsynchronisation erlaubt es, Aktionen alle n Millisekunden auszuführen, auch wenn die Aktion selbst eine umbestimmte Zeit dauert (solange diese unter n Millisekunden liegt). Vor der Aktion wird mit SYNC! die Dauer gespeichert, nach der Aktion mit SYNC auf das Erreichen dieses Zeitpunktes gewartet. Dies ermöglicht eine wesentlich genauere Synchronisation als mit einem einfachen WAIT-Befehl.

TASK (rlen slen −−) ⟨*Name*⟩:⟨*Name*⟩ (−− Taddr): Legt eine Taskarea unter dem Namen ⟨*Name*⟩ an. **rlen** ist die Größe des Puffers für Returnstack und Userarea. Da bigFORTH den Supervisorstack als Returnstack nutzt, muß berücksichtigt werden, daß Interrupts und Systemaufrufe ihre Werte auch auf den Returnstack legen. Er sollte mindestens $200 bis $300 Bytes groß sein. **slen** ist die Länge des Puffers zwischen HERE und S0. Here und PAD zusammen benötigen $164 Bytes, also ist es auch vorteilhaft, $200 Bytes für den Stack zu reservieren.

RENDEZVOUS (Semaphor −−): Gibt eine Semaphor für die Zeit eines Taskwechsels frei und versucht, sie dann wieder für den aktuellen Task zu locken.

'S (Taddr −− T.Useraddr) ⟨***Uservariable***⟩ **immediate:** Berechnet die Adresse der Uservariablen im Task **Taddr**.

TASKS (−−): Listet alle Tasks auf. Der aktuelle Task wird mit "Main" bezeichnet. Zu den Tasks wird der Status "sleeping" ausgegeben, wenn sie nicht aktiviert sind.

Als Beispiel kann rechts oben eine Uhr mit Digitalziffern in einem eigenen Task gestartet werden. Da die Systemuhr nur im 2-Sekunden-Takt läuft, bietet es zudem noch eine Beispielanwendung für SYNC! und SYNC.

CLOCKTASK (−− Taddr): In diesem Task läuft die Uhr. Die Uhr ist als autostart Task angelegt, läuft also gleich nach dem Systemstart los.

CLOCK (−−): Startet die Uhr. CLOCK ist das eigentliche Uhrprogramm.

WAITC (−−): Hält den Uhrtask an.

STARTC (−−): Startet den Uhrtask wieder.

NOCLOCK (−−): Schaltet die Uhr aus. Auch nach einem erneuten Systemstart läuft die Uhr nicht wieder an.

SETCLOCK (−−): Mit diesem Befehl kann man die Uhrzeit stellen. Es wird folgende Zeile ausgegeben:

```
Geben Sie die aktuelle Uhrzeit ein: ..:..
```

Der Cursor steht auf dem ersten Punkt. Geben Sie die aktuelle Uhrzeit in Stunden und Minuten ein, vergessen Sie nicht, an der Stelle des Doppelpunktes auch ein beliebiges Zeichen einzugeben (wird nicht ausgewertet).

7. Druckertreiber

bigFORTH unterstützt Listings auf dem Drucker. Ebenso wird eine gleichzeitige Ausgabe auf dem Bildschirm und Drucker ermöglicht (Protokollfunktion).

PRINTER.STR (−−): Aus dieser Datei wird der Druckertreiber geladen.

ARGUMENTS (n1 .. nm m −− n1 .. nm): Stellt fest, ob überhaupt **m** Argumente auf dem Stack liegen. Ist dies nicht so, wird mit der Meldung "arguments ?!" abgebrochen.

PRINTER (−−) (VS voc −− PRINTER): Viele Befehle des Druckertreibers stehen in einem eigenen Vokabular. Nur die High-Level-Befehle sind im Vokabular FORTH definiert.

PRINT (−−): Leitet die Ausgabe auf den Drucker ein. Der Drucker wird initialsisiert (siehe NORMAL) und die Ausgabe (Uservariable OUTPUT) wird auf den Drucker umgeleitet.

(PROTOKOLL (−−): Ausgabeblock für die Protokollausgabe. Jede Ausgabe wird sowohl auf dem Bildschirm als auch auf dem Drucker ausgegeben.

PROTOKOLL (−−): Löscht den Bildschirm, initialisiert den Drucker und legt die Ausgabe auf (PROTOKOLL. Dieses Wort dient dazu, Interaktionen am Bildschirm auf dem Drucker zu protokollieren. Beendet werden kann die Protokollausgabe mit DISPLAY oder STANDARDI/O.

PTHRU (first last −−): Druckt die Screens **first** bis einschließlich **last** der aktuellen Datei aus. Dabei werden auf jeder Seite drei Screens ausgedruckt, die ersten drei links, die zweiten drei rechts. Werden weniger als 6 Screens ausgegeben, so werden rechts "Logo-Screens" (Screen 0) ausgegeben und unten wird der Platz frei gelassen. Damit zwei Screens mit jeweils 64 Zeichen nebeneinander Platz haben, wird in Schmalschrift gedruckt. Zur Veranschaulichung:

Screens	1-6	1-5	1-4	1-3	1-2	1-1
	1 4	1 4	1 3	1 3	1 2	1 0
	2 5	2 5	2 4	2 0		
	3 6	3 0				

PRINTALL (−−): Druckt alle Screens einer Datei mit PTHRU aus.

DOCUMENT (first last −−): Druckt die Screens **first** bis einschließlich **last** der aktuellen Datei aus. DOCUMENT eignet sich für Dateien mit Shadow-Screens. Links werden die Programm-Screens, rechts die dazugehörenden Shadow-Screens ausgedruckt.

LISTING (−−): Druckt eine ganze Datei mit DOCUMENT aus.

SPOOLER (−− Taddr): Task Area für den Spooler.

SPOOL' ([first last] −−) ⟨*Word*⟩: Das Wort ⟨***Word***⟩ wird im Hintergrund abgearbeitet. Primär ist der Spooler für das Drucken im Hintergrund gedacht, es können aber auch andere Wörter im Spooler-Task laufen. PTHRU und DOCUMENT müssen natürlich die Parameter **first** und **last** übergeben werden, SPOOL' nimmt deshalb bis zu zwei Werte vom Stack (wenn sie vorhanden sind). Nach dem Ende des Druckvorgangs wird mit dem Fehler "SPOOL Task's ready for next Job." abgebrochen.

Die eigentlichen Treiberbefehle sind im Vokabular PRINTER:

P! (char −−): Gibt **char** auf dem Drucker aus. Dabei wird gewartet, bis der Drucker bereit ist, es gibt kein Timeout. Deshalb muß der Drucker auch angeschaltet und Papier eingelegt sein.

BEL (−−): Sendet das Zeichen BEL ($07) an den Drucker. Der Drucker gibt einen Signalton von sich (wenn er eine Glocke hat).

LF (−−): Line Feed. Zeilenvorschub.

FF (−−): Form Feed. Seitenvorschub.

1/8" (−−): Setzt den Zeilenabstand auf 1/8 Zoll.

1/10" (−−): Setzt den Zeilenabstand auf 1/10 Zoll.

1/6" (−−): Setzt den Zeilenabstand auf 1/6 Zoll. Dies ist die Standardeinstellung des Druckers.

SUOFF (−−): Schaltet Sub- oder Superscript (Hoch- oder Tiefstellen) aus.

+JUMP (−−): Schaltet den Perforationssprung ein. Beim Erreichen des unteren Blattrandes wird automatisch die Perforation übersprungen.

−JUMP (−−): Schaltet den Perforationssprung aus. Endlospapier kann dann auch "endlos" bedruckt werden.

Die nun folgenden Attribute können kombiniert werden. Sie werden alle einzeln mit den entsprechenden Befehlen ein- und ausgeschaltet. Es wird von einem EPSON FX80-kompatiblen Drucker ausgegangen.

+DARK (−−): Schaltet den Doppeldruck ein (auf 9-Nadeldrucker und 24-Nadeldrucker mit altem Farbband wird der Ausdruck somit dunkler).

−DARK (−−): Schaltet den Doppeldruck aus.

+FAT (−−): Schaltet Fettschrift ein.

−FAT (−−): Schaltet Fettschrift aus.

+CURSIVE (−−): Kursivschrift (Schrägschrift) ein.

−CURSIVE (−−): Kursivschrift aus.

+WIDE (−−): Breitschrift ein (doppelte Breite).

−WIDE (−−): Breitschrift aus.

+UNDER (−−): Unterstreichen ein.

−UNDER (−−): Unterstreichen aus.

SUB (−−): Subscript (Tiefstellen) ein. Ausschalten mit SUOFF.
SUPER (−−): Superscript (Hochstellen) ein. Ausschalten mit SUOFF.
+SILENT (−−): Leisedruck ein. Das Gerät druckt halb so schnell.
−SILENT (−−): Leisedruck aus.

Die nun folgenden Befehle sind nur für den NEC P6 wirksam, nicht für den EPSON FX80 selbst. Sie können in der Datei PRINTER.SCR auskommentiert werden.

+SPEED (−−): Schnelldruck ein (nur für Draft 12 CPI=Highspeed Draft).
−SPEED (−−): Schnelldruck aus.
+HEIGHT (−−): Doppelte Höhe ein.
−HEIGHT (−−): Doppelte Höhe aus.
+JUMP (−−): Perforationssprung um 6 Zeilen. Im Gegensatz zum FX-80 muß beim P6 ein Parameter zum Perforationssprung übergeben werden, deshalb ist dieser Befehl zweimal vorhanden.
+NLQ (−−): Near Letter Quality ein.
−NLQ (−−): Near Letter Quality aus.
10CPI (−−): Schaltet auf 10 CPI (Characters per Inch).
PICA (−−): Schaltet auf 10 CPI (normale Schreibmaschinenschrift).
12CPI (−−): Schaltet auf 12 CPI.
ELITE (−−): Schaltet auf 12 CPI (schmalere Schreibmaschinenschrift).
15CPI (−−): Schaltet auf 15 CPI (nur NEC P6).
17CPI (−−): Schaltet auf 17 CPI.
SMALL (−−): Schaltet auf 17 CPI (Schmalschrift).
20CPI (−−): Schaltet auf 20 CPI (Nur NEC P6).
LINES (#lines −−): Setzt die Seitenlänge auf #lines Zeilen.
“LONG (zoll −−): Setzt die Seitenlänge auf zoll Zoll.
AMERICAN (−−): Schaltet in den amerikanischen Modus.
GERMAN (−−): Schaltet in den deutschen Modus. Es können die deutschen länderspeziefischen Sonderzeichen ausgegeben werden.
NORMAL (−−): Initialisiert den Drucker auf 10 CPI, 6 LPI und amerikanischen Modus.
FILTER (c −− c1 .. cn n): Deferred Word. Dient als Zeichenfilter. Übergeben wird das auszugebende Zeichen, zurückgegeben wird die Sequenz auszugebender Zeichen. **c1** wird zuerst ausgegeben, **cn** zuletzt.
NOFILTER (−−): Schaltet den Filter aus (setzt FILTER auf 1).
(FILTER (c −− c1 .. cn n): Filter für den ST. Paragraph und scharfes ß müssen gewandelt werden, da ersteres im IBM-Zeichensatz nicht vorhanden ist und letzteres eine andere Nummer hat.
>PRINTER (−−): Ausgabeblock für den Drucker. Lenkt die Ausgabe auf den Drucker um.

8. Die Notbremsen

Wie in den Kapiteln 2.19 und 2.20 beschrieben, werden sowohl auftretende Prozessor-Fehler als auch Resets abgefangen.

EXCEPT.SCR (−−): Aus dieser Datei werden die erweiterten Exception-Traps geladen. Eine genaue Beschreibung der Wirkungsweise finden Sie im Kapitel 2.20.
.REGS (−−): Gibt die gesicherten Register aus.

SAVEREGS (−−): Sichert die Register in dem Feld, in dem sie von .REGS ausgegeben werden können.

WR> (−− 16b) (RS 16b −−): Holt einen 16-Bit-Wert vom Returnstack.

NEWTRAPS (−−): Hängt in 'RESTART und sorgt dafür, daß die neuen Traps auch in die entsprechenden Vektoren eingehängt werden.

RESET.SCR (−−): Die Wörter in dieser Datei machen bigFORTH resetfest. Eine genaue Beschreibung finden Sie im Kapitel 2.19.

RESETFEST (−−): Macht bigFORTH resetfest. Es wird der Zeiger der Resetroutine in den Resetvektor (resvector=$42A) geschrieben. resvalid (=$426) wird auf den Wert $31415926 gesetzt und damit gültig gemacht. Die alten Werte werden gesichert. Ebenso werden die Register der Peripheriebausteine ausgelesen, um nach dem Reset die alten Werte wieder zurückzuschreiben. Probleme kann es mit den ST-aufwärtskompatiblen STE und TT geben, die zusätzliche Peripheriebausteine besitzen, welche nicht initialisiert werden können.

Nach einem Reset werden zunächst resvektor und resvalid zurückgesetzt, ebenso wie der BIOS/XBIOS-Registerstack in $4A2. Die alten Werte werden in die Peripheriebausteine geschrieben und der Bildschirm wird synchronisiert. Der weitere Ablauf entspricht dem in Kapitel 2.19 beschriebenen.

(BYE (−−): resvalid und resvektor müssen nach Verlassen des Systems natürlich auch wieder in den ursprünglichen Zustand zurückgesetzt werden — sonst ist der Inhalt der resetfesten RAM-Disk nach dem nächsten Reset ganz sicher unwiederbringlich verloren. Deshalb wird ein (BYE in 'BYE eingehängt, das diese Aufgabe übernimmt.

9. Hot Keys

Häufig benutzte Befehle können auf die Funktionstasten gelegt werden. Auch das ist eine Arbeitserleichterung, die die Arbeit angenehmer macht. Die Belegung der Tasten finden Sie in Kapitel 2.3.

FTAST.SCR (−−): Aus dieser Datei werden die Definitionen für die "Hot Keys" geladen.

STDECODE (addr pos0 key −− addr pos1): Ein neues STDECODE wird definiert, das die Funktionstasten auswertet. Es wird anstelle des STDECODEs im Kernel in den Outputblock geschrieben.

F' (n −−) ⟨***Word***⟩**:** ⟨***Word***⟩ wird beim Druck der Funktionstaste F**n** aufgerufen. Dabei werden geshiftete Funktionstasten als F11-F20 betrachtet. Beispiele:

```
9 F' V \ Der Editor kann mit F9 aufgerufen werden
11 F' DECIMAL \ Sh F1 schaltet auf Dezimal
12 F' HEX \ Sh F2 schaltet auf Hexadezimal
```

10. Tools für GEM

Eine Reihe von Befehlen erleichtert den Umgang mit GEM ganz beträchtlich, ist aber auf logischer Ebene eher in der Nähe der Kernel-Befehle anzusiedeln. Koordinatenwandlung und Speicherkommunikation werden beim Umgang mit GEM verstärkt benötigt - GEM liefert die Daten nicht gerade "FORTH-gerecht". Und der Umgang mit Punkten und Rechtecken kann durch ein paar Befehle sehr vereinfacht werden.

GEMLOAD.SCR (−−): Aus dieser Datei werden alle GEM-Libaries geladen.

EXTEND.SCR (−−): Aus dieser Datei werden die Erweiterungen geladen, die für GEM sehr brauchbar sind, aber nicht direkt zu GEM gehören — die Tools für GEM. Sie sind im Vokabular FORTH definiert.

2@ (addr −− d): Liest die doppelt genaue Zahl **d** aus **addr** aus. Der höherwertige Teil steht dabei an niedriger Adresse.

2! (d addr −−): Speichert die doppelt genaue Zahl **d** in **addr**.

2NIP (d1 d2 −− d2): Wie NIP, nur für doppelt genaue Zahlen.

2VARIABLE (−−) ⟨*Name*⟩:⟨*Name*⟩ (−− addr): Wie VARIABLE, legt aber einen zwei Zellen (acht Bytes) großen Bereich an.

2CONSTANT (D −−) ⟨*Name*⟩:⟨*Name*⟩ (−− D): Wie CONSTANT, für doppelt genaue Zahlen.

4DUP (n1 .. n4 −− n1 .. n4 n1 .. n4): Verdoppelt die vier obersten Werte auf dem Stack. Sie liegen dann nochmal in gleicher Reihenfolge auf dem Stack.

WSWAP (n1 −− n2): Vertauscht High- und Low-Word von **n1**.

PIN (n0 n1 .. nx n x −− n n1 .. nx): "Destruktiver" Gegenspieler von PICK. Schreibt den Wert n an die x-te Stelle im Stack, von oben aus gezählt.

WARRAY! (n1 .. nm addr m −−): Speichert die m 16-Bit-Werte **n1 .. nm** ab **addr**. Begonnen wird dabei mit **n1**.

WARRAY@ (addr m −− n1 .. nm): Liest **m** 16-Bit-Werte ab **addr** aus. Der erste Wert liegt im Stack unten.

ARRAY! (n1 .. nm addr m −−): Speichert **m** 32-Bit-Werte ab **addr**. Wie WARRAY!.

ARRAY@ (addr m −− n1 .. nm): Liest **m** 32-Bit-Werte ab **addr** aus. Wie WARRAY@.

4W! (n1 .. n4 addr −−): Speichert 4 16-Bit-Werte ab **addr**. Wie 4 WARRAY!.

2W! (n1 n2 addr −−): Speichert 2 16-Bit-Werte ab **addr**. Wie 2 WARRAY!.

4W@ (addr −− n1 .. n4): Liest 4 16-Bit-Werte ab **addr** aus. Wie 4 WARRAY@.

2W@ (addr −− n1 n2): Liest 2 16-Bit-Werte ab **addr** aus. Wie 2 WARRAY@.

4! (n1 .. n4 addr −−): Speichert 4 32-Bit-Werte ab **addr**. Wie 4 ARRAY!.

4@ (addr −− n1 .. n4): Liest 4 32-Bit-Werte ab **addr** aus. Wie 4 ARRAY@.

WARRAYCON (C0 .. Cn-1 n −−) ⟨*Name*⟩:⟨*Name*⟩ (i −− Ci): Speichert **n** 16-Bit-Konstanten in einem Feld mit dem Namen ⟨***Name***⟩. Zugegriffen werden kann per Index, bei Bereichsüberschreitung wird der letzte Wert **Cn-1** zurückgegeben.

ARRAYCON (C0 .. Cn-1 n −−) ⟨*Name*⟩:⟨*Name*⟩ (i −− Ci): Speichert **n** 32-Bit-Konstanten, sonst wie WARRAYCON.

AARRAYCON (A0 .. An-1 n −−) ⟨*Name*⟩:⟨*Name*⟩ (i −− Ai): Speichert **n** als Adressen markierte Werte. Sonst wie ARRAYCON.

>HL00 (n −− n_h n_l 0 0): Zerlegt eine Zahl in High-Word und Low-Word und legt noch zweimal 0 auf den Stack. wind_set müssen Adressen in dieser Form übergeben werden.

RCELL+ (−−) (RS n −− n+4): Erhöht den obersten Wert auf dem Returnstack um 4.

PAIR (x1 y1 x2 y2 −− x1Xx2 y1Xy2) ⟨*Name*⟩ immediate: Verknüpft **x1**, **x2** und **y1**, **y2** paarweise mit dem binären Operator ⟨***Name***⟩. Damit kann auf Punkte operiert werden. PAIR + beispielsweise addiert zu einen Punkt ein weiteres Wertepaar.

2DO (x y −− Xx Xy) ⟨*Name*⟩ immediate: Verknüpft **x** und **y** mit dem unärem Operator ⟨***Name***⟩.

P1− (x y −− x-1 y-1): Subtrahiert von **x** und **y** 1. Wirkt wie 2DO 1-.

>XYXY (x y w h −− x1 y1 x2 y2): Wandelt ein Rechteck vom AES-Format (Punkt, Breite und Höhe) in das VDI-Format (zwei Punkte). Entspricht 2OVER PAIR + P1-, es werden allerdings die Koordinaten sortiert (**x1 y1** liegen links oben).

>XYWH (x1 y1 x2 y2 −− x1 y1 w h): Wandelt ein Rechteck vom VDI-Format in das AES-Format. Entspricht 2OVER PAIR - 2DO 1+. Die Koordinaten werden nicht sortiert.

Da in GEM einige Strukturen bitweise aufgeteilt sind, ist es nützlich, wenn man dieselben Bitshift-Befehle wie in C zur Verfügung hat.

<< (n1 n2 −− n3): Bitshift von **n1** um **n2** nach links. Entspricht einer Multiplikation mit $2^{\mathbf{n2}}$.

>> (n1 n2 −− n3): Bitshift von **n1** um **n2** nach rechts. Entspricht einer Division durch $2^{\mathbf{n2}}$.

U>> (n1 n2 −− n3): Bitshift vorzeichenlos um **n2** nach rechts. Im Gegensatz zu >> wird nicht mit dem vordersten Bit, sondern mit 0 aufgefüllt.

R<< (n1 n2 −− n3): Bitrotation links um **n2**.

R>> (n1 n2 −− n3): Bitrotation rechts um **n2**.

GEM benutzt ausschließlich 0-terminated Strings. Damit die Benutzung leichterfällt, gibt es noch einige Wörter, die den Umgang mit solchen Strings erleichtern.

0PLACE (addr0 count addr1 −−): Speichert den String **addr0 count** als 0-terminated String in **addr1**.

,0“ (−−) ⟨*String*⟩”: Compiliert ⟨***String***⟩ als 0-terminated String.

0“ (−− addr) ⟨*String*⟩” immediate: Gibt die Adresse des 0-terminated Strings ⟨***String***⟩ zurück. Einsatz wie “ ⟨***String***⟩”.

BLANK (addr len −−): Füllt den Bereich **addr len** mit Leerzeichen.

Zur Zeitmessung gibt es noch eine Stoppuhr. Sie hat eine Genauigkeit von 1/200 Sekunde und benutzt den System-Timer. Sie eignet sich vor allem zur laufenden Zeitmessung in Benchmarks. Der Start wird mit !TIME markiert, die Zeit wird mit .TIME ausgegeben.

TIME (−− addr): In dieser Variable wird der Startwert für die Stoppuhr gespeichert.

!TIME (−−): Speichert den Startwert für die Stoppuhr.

.TIME (−−): Gibt die aktuelle Zeit der Stoppuhr aus.

GETTOS# (−− tos#): Holt die Versionsnummer des TOS. Die Nummer $100 bedeutet TOS 1.0 (“(Altes) ROM-TOS”), $102 TOS 1.2 (“Blitter-TOS”) und $104 TOS 1.4 (“Rainbow-TOS”).

9 Objektorientiertes FORTH

1. Was ist objektorientierte Programmierung?

Das Modewort des Endes der 80er Jahre und auch noch Anfang der 90er Jahre in der Software–Industrie ist zweifellos „objektorientiert“. Kein Betriebssystem, kein Anwenderprogramm und erst recht keine Programmiersprache, die nicht objektorientiert ist. Forth macht da selbstverständlich mit, wie verschiedene Veröffentlichungen, wie etwa Dick Pountain's “Object–Oriented FORTH” [1] deutlich zeigen.

Ewald Rieger hatte Pountain's OOF portiert und mir auf der FORTH–Tagung '91 zum Ansehen mitgegeben. Wegen verschiedener Unzulänglichkeiten wurde daraus eine völlige Neuimplementierung, die dieses interessante Programm–Paradigma für bigFORTH zur Verfügung stellt. Das System ist seit etwa drei Jahren bei Ewald Rieger im Einsatz und hat damit seine Praxistauglichkeit auch in der etwas rauheren Welt der Echtzeitprogrammierung bewiesen.

Der Verbund von Daten und Algorithmen zu einem Objekt hat sich gerade für wechselnde Hardwarekonfigurationen, wie sie für viele Aufgaben in der Praxis typisch ist, als sehr brauchbar, wenn nicht unverzichtbar erwiesen.

Damit man abschätzen kann, was mit objektorientierter Programmierung möglich ist, und wie das gemacht wird, folgt nun eine Einführung anhand eines Beispiels (das allerdings nicht auf Hardware zugreift — denn sonst könnte man — ohne Hardware — kaum damit spielen).

Die Sourcen finden sich in der Datei `OOFSAMPL.SCR`. Als Beispiel dient eine kleine Sammlung von Datentypen, wie sie aus der Informatik bekannt sind: Integer, Listen, Arrays und Pointer.

Objektorientierte Programmierung versteckt sich hinter einem Slang, der bei näherer Betrachtung durchaus Ähnlichkeiten mit bekannten Konzepten wie Modularität und der Definition sauberer Schnittstellen erkennen läßt.

1.1. Das Klassenkonzept

Der Kerngedanke an der objektorientierte Programmierung ist die Kapselung von Daten und den sie bearbeitenden Prozeduren in ein *Objekt*. Im Idealfall haben die Prozeduren („Methoden“) eines Objekts alleinigen Zugriff auf dessen Daten und sind somit die einzige Möglichkeit, die Daten zu bearbeiten. Schnittstelle zu einem Objekt sind die Namen der Methoden (Messages) und Parameter, die bei den Messages mitgeschickt werden. Da Objekte auf viele Methoden auch ein Ergebnis zurückliefern, verwendet man für das “message passing” ganz konventionell den Stack und ruft die Methode wie ein FORTH–Wort auf — mit dem Umweg über das Objekt, das die Kapselung handhabt.

Nun wäre es eine ziemliche Verschwendung, für jedes Objekt eigene Methoden neu zu programmieren; vor allem, da viele Objekte einen gleichen oder ähnlichen Aufbau haben und sich nur in den Daten unterscheiden. Solche gleichartige Objekte faßt man zu einer Klasse zusammen. Eine Klasse ist gewissermaßen eine Schablone (oder ein Formular) für ein Objekt; erst durch Instanzierung entsteht aus der Klasse ein reales Objekt mit Platz für Daten und den allen Objekten der Klasse gemeinsamen Methoden.

Da oft an einer Klasse nur kleine Modifikationen vorgenommen werden müssen, um eine neue Klasse zu erhalten, benutzt man die „Vererbung“ (“Inheritance”) um eine Unterklasse — ein Derivat — zu erhalten. Zusätzlich benötigte Variablen und veränderte oder neue Methoden werden zur Klasse nur dazugefügt.

Alle Objekte einer Klasse und auch die aller Unterklassen haben ein gemeinsames Message–Protokoll, sie verstehen dieselben Messages und reagieren gleichartig darauf. Die Unterschiede im Detail nennt man “Polymorphismus” — Vielgestaltigkeit. So mag z. B. jedes Graphik–Objekt eine Methode „zeichne Dich“ haben, das eine Objekt aber einen Kreis, das andere einen Punkt oder ein Rechteck darstellen.

Legt man viel Wert auf ein gleichartiges Protokoll zu allen Objekten einer Klassenhierarchie, so entwirft man dieses Protokoll getrennt von den einzelnen Implementierungen der Unterklassen und nennt die so entstandene Klasse, die nur Protokoll, nicht aber Implementierung enthält, „abstrakter Datentyp“. Ein solcher ist die im folgenden Listing vorgestellte Klasse `data`:

```
\ Data structures: data                                    30apr93py

Memory also Forth

object class data       \ abstract data class
        cell var ref    \ reference counter
public: method !        method @        method .
        method null     method atom?    method #
how:    : atom? ( -- flag )  true ;
        : #     ( -- n )     0 ;
        : null  ( -- addr )  new ;
class;
```

Hier muß noch dazu gesagt werden, daß in bigFORTH alle Klassen letztendlich von einer Vater–Klasse abstammen müssen, von der Klasse `object`. Auch sind Klassen und Objekte nicht nur für die Bearbeitung von Daten zuständig, sondern auch für die Erzeugung neuer Unterklassen und Instanzen der Objekte. Deshalb ist eine Klasse ein nur um seinen Datenbereich beschnittenes Objekt, das mit der Methode `class` eine neue Unterklasse erzeugen kann.

Die Beschreibung einer Klasse besteht aus zwei Teilen: Der Deklaration von Variablen und Methoden, sowie der Implementierung der Methoden. Alle Variablen, alle polymorphen und von außen zugänglichen Methoden müssen deklariert werden; Hilfsmethoden können deklariert werden, müssen aber nicht. Bei der Implementierung werden undeklarierte Methoden automatisch als EARLY (privat) deklariert.

Im Beispiel wird eine Variable namens `ref` von der Größe einer Zelle angelegt, und zwar im privaten Bereich der Klasse, die von außen nicht sichtbar ist, wohl aber vererbt werden kann (entspricht also dem `protected:` in C++). `public:`, also öffentlich zugänglich sind die 6 Methoden `!`, `@`, `.`, `null`, `atom?` und `#`, die für Speichern, Auslesen und Anzeigen des Wertes, zur Erzeugung eines „Null“–Objektes, der Feststellung, ob das Objekt unteilbar oder zusammengesetzt ist, sowie der Anzahl an Unterobjekten, falls letzteres der Fall ist.

Die letzten drei Methoden werden schon implementiert, da sie für alle einfachen Objekte gleich sind. Die nicht implementierten Methoden können nicht ausgeführt werden, genauer gesagt, sie führen auf `abort"`. Sie müssen in realen Datentypen implementiert werden, wie im folgenden Datentyp Integer:

```
\ Data structures: int                                  30apr93py

data class int
        cell var value
how:    : !     value F ! ;
        : @     value F @ ;
        : .     @ 0 .r ;
        : init  ( data -- )  ! ;
        : dispose  -1 ref +!
          ref F @ 0> 0= IF  super dispose  THEN ;
        : null  ( -- addr )  0 new ;
class;
```

Hier wird in bekannter Manier eine neue Klasse erzeugt und eine (private) Variable `value` angelegt. Die beiden Methoden store und fetch (`!` und `@`) greifen auf `value` zu — als Schnittstelle völlig ausreichend. Das `F` vor den Bezeichnern verhindert, daß die Methoden des Objekts selbst verwendet werden und schaltet statt dessen auf das normale Vokabular um — es werden also die normalen Wörter, die man unter dem Namen `!` und `@` kennt, verwendet. Auch die Methode `.` ist einfach zu verstehen. Hier steht `@` allerdings für die Zugriffsmethode.

Zu den Methoden `init` und `dispose` muß noch etwas gesagt werden: `init` wird beim Erzeugen eines Objekts aufgerufen und dient der Vorinitialisierung des Objekts. Hier im Beispiel wird `value` mit einer auf dem Stack liegenden Zahl initialisiert. `dispose` entfernt ein Objekt aus der dynamischen Speicherverwaltung. Modifiziert man diese Methode, muß man (anders als in C++) explizit auch die Entfernungsmethode der Vaterklasse aufrufen (mit `super dispose`). Entfernt wird nur, wenn der Reference Counter 0 oder negativ geworden ist, also kein Verweis mehr vorhanden ist. Andernfalls wird der Reference Counter eben nur erniedrigt.

`null` erhält hier die Bedeutung, die man schon vermutet: Es legt ein Objekt mit dem Wert 0 (dynamisch) an und hinterläßt dessen Adresse. `new` ist ebenso wie `dispose` eine Methode. Ohne zusätzliche Information (also ohne Angabe von Klasse oder Objekt) wird die Methode der aktuellen Klasse verwendet.

1.2. Binding: Late oder early?

An dieser Stelle empfiehlt es sich, etwas über die Vorgehensweise beim Aufruf einer Methode auszuholen. Wieviel kann hier schon zur Compile–Zeit bestimmt werden und was muß bis zur Laufzeit offenbleiben (und sollte dann möglichst keinen Fehler mehr produzieren)?

Sofern, wie bei `super dispose` klar ist, welche Methode welcher Klasse ausgeführt werden soll, wird man das schon zur Compile–Zeit auflösen und einen direkten Call zur angegebenen Methode compilieren. Das nennt man dann "early binding". Das ist sicher am schnellsten, und auch am einfachsten, erlaubt aber leider keinen Polymorphismus.

Oft ist ja nicht von vornherein klar, welcher Unterklasse das Objekt angehört, dessen Methode man ausführen will. Man muß also die genaue Adresse der Methode zur Laufzeit herauskriegen. Eine Suche im Dictionary kommt aus Effiziensgründen nicht in Frage, auch eine (womöglich gar sequentielle) Suche über einen numerischen Schlüssel ist nicht das, was man sich unter Laufzeiteffizienz vorstellt.

In bigFORTH steht daher in jedem Objekt an erster Stelle ein Zeiger auf eine Sprungtabelle, in der alle Methoden eingetragen sind. Das garantiert nicht nur die Reaktionszeit

unabhängig von der Menge der Methoden (schließlich soll FORTH ja eine Echtzeitsprache bleiben), es ist auch sehr schnell. Vor allem, weil das Verfahren so einfach codiert werden kann, daß es als Makro direkt in den Code des Aufrufers eingefügt wird.

Was man sich mit diesem Trick verbaut, ist die Mehrfachvererbung, auch "Multiple Inheritance" genannt. Aus mehreren Vaterklassen eine Tochterklasse zu kreuzen ist ohnehin problematisch: Gleichnamige Methoden und Variablen müssen umbenannt werden, wenn sie nicht von derselben „Opaklasse" vererbt worden sind, die Offsets für die Variablen im Objekt ändern sich (müssen also auch erst zur Laufzeit bestimmt werden) oder der Compiler muß schon vorher Sorge tragen, daß der in der Mischklasse nötig gewordene Platz in seinen Vorfahren reserviert ist (ein Space–Time–Tradeoff, der aber mit einem One–Pass–Compiler nicht zu machen ist und selbst in komplexeren Systemen zu fast unüberwindbaren Schwierigkeiten führt).

Ein wichtiger Aspekt ist auch die Echtzeitfähigkeit des erzeugten Codes. Dazu müssen die Ablaufzeiten vom Programmierer bestimmt werden können; es bietet sich also nur ein völlig deterministisches Verfahren zum Erzeugen der Bindings an. Nur dadurch verliert der Programmierer nicht die Kontrolle über das, was er da schreibt. C++ hat diese Eigenschaft nicht und ist deshalb für Echtzeitaufgaben nur begrenzt einsatzfähig.

1.3. Objekte als Instanzvariablen

Wie dem auch sei, oft kommt man mit klaren Klassenhierarchien gut aus. Geeignete abstrakte Datentypen erlauben es oft, eine echte Mehrfachvererbung zu umgehen. Notfalls kann man Mehrfachvererbung auch von Hand und mit dem Editor durch Zusammenkopieren der Sourcen erreichen. Was allerdings unbedingt nötig ist, sind Objekte als Instanzvariablen in einem anderen Objekt, und zwar sowohl als Zeiger als auch direkt. Das wird beim Beispiel der Listen, die ja nur mit Pointern implementiert werden können, deutlich:

```
\ Data structures: list                                  17nov93py

forward nil
data class lists
public: data ptr first  data ptr next
        method empty?   method ?
how:    : null  nil ;
        : atom? false ;
class;

| lists class nil-class
how:    : empty? true ;
        : dispose ;
        : . ." ()" ;
class;

| nil-class : (nil
(nil self Aconstant nil
nil (nil bind first
nil (nil bind next
```

Hier wird erst einmal die abstrakte Datenklasse der Listen angelegt; diese brauchen an Daten sowohl einen Zeiger auf das erste als auch auf den Rest der Liste. Dieses kann auch

ein normales Datum sein, es sind also "dot pairs" wie in Lisp erlaubt. Müßte der Rest wieder eine Liste sein, wird der Typ nicht angegeben; das erzeugt dann einen Pointer auf ein Objekt der aktuell deklarierten Klasse. `lists ptr next` ginge nicht, da die Klasse `lists` zu diesem Zeitpunkt noch nicht fertig definiert ist und deshalb nicht ausgeführt werden kann.

Neben diesen Pointern braucht man natürlich auch noch ein paar Methoden: Eine Liste kann natürlich auch leer sein, also muß man das abfragen können. Ebenso ist oft ganz brauchbar, das erste Element anzuzeigen (mit `?`).

Eine Null–Liste ist die leere Liste, auch "nil" genannt. Da das eine Liste ist, kann sie erst später deklariert werden, also wird eine Vorwärts–Referenz angelegt, die sich bei der der späteren Definition von `nil` auflöst.

Leere Listen unterscheiden sich in ihrem Verhalten von anderen deutlich. Sie geben auf `empty?` immer true zurück, es gibt nur eine von ihnen und die darf natürlich nicht gelöscht werden. Ausgegeben wird sie als ein Klammernpaar.

Von der Klasse der leeren Listen wird nun ein Element angelegt, und die Adresse dieses Elements (die die Methode `self` auf den Stack legt) heißt dann endlich `nil`. Sowohl erstes als auch nächstes Element der leeren Liste ist wieder die leere Liste. Damit fällt man nicht auf die Nase, wenn man über das Listenende hinausgeht.

Die Methode `bind` erlaubt es, Objekt–Referenzen an echte Objekte zu binden (wie der Name sagt). Der Objekt–Pointer `first` des Objekts `(nil` verhält sich, nachdem er gebunden ist, also genauso wie das Objekt, an das er gebunden ist, also `(nil` selbst. Interessanter wird das aber erst bei echten Listen:

```
\ Data structures: list                                    17nov93py

lists class linked
how:    : empty? false ;
        : init ( first next -- )
                dup >o 1 ref +! o> bind next
                dup >o 1 ref +! o> bind first ;
        : ?     first . ;
        : @     first @ ;
        : !     first ! ;
        : .     self >o  '(
                BEGIN  emit ?  next atom?  next self o> >o
                       IF ."  . " data . o> ." )"  EXIT  THEN  bl
                       empty?  UNTIL  o> drop ." )" ;
        : #     next # 1+ ;
        : dispose  -1 ref +!  ref F @ 0> 0=
          IF first dispose  next dispose  super dispose THEN ;
class;
```

Eine linked list ist natürlich nicht leer. Beim Erzeugen werden die Referenzen `first` und `next` gleich gebunden; entsprechende Objekt–Addressen müssen also schon auf dem Stack liegen. Beim Binden müssen auch die Reference Counter der Objekte um eins erhöht werden — es zeigt ja jetzt ein Pointer mehr auf sie. Damit sie aktuelles Objekt werden, werden sie auf den Objekt–Stack geschoben, dadurch wird mit `ref` *ihr* Reference Counter angesprochen, und nicht der der Liste. Der Objekt–Stack ist übrigens kein richtiger; nur das oberste Element wird in einem Register gehalten, der Rest auf dem Returnstack.

Die Methoden `@`, `!` und `?` beziehen sich, auf das jeweils erste Objekt der Liste; sie werden einfach weitergereicht. Keine komplexe Zeigerverwaltung ist notwendig, der Name der Referenz reicht aus.

Zur Ausgabe der Liste muß man sich durch die Liste durchhangeln. Vor dem ersten Element muß eine Klammer geöffnet werden, ansonsten werden die Elemente durch einen Blank getrennt. Das jeweils erste Element der Liste wird ausgegeben. Ist das nächste Element ein Atom, muß es als dot pair ausgegeben werden, die Liste ist damit zu Ende. Auch zu Ende ist sie, wenn das nächste Element die leere Liste ist. Dann muß nur noch die Klammer geschlossen werden und der Blank vom Stack gelöscht werden.

Verblüffend ist die Rekursion in `#`, das die Länge der Liste zurückgibt. Es wird einfach die Länge des Rests der Liste bestimmt, das um 1 erhöht und die Sache hat sich. Solange die Liste mit `nil` oder einem Atom abschließt, das definitiv die Länge 0 hat, terminiert die Rekursion. Hier zeigt sich erstmals klar der Vorteil objektorientierter Programmierung, der viele `IF..ELSE..THEN` für Fallunterscheidungen überflüssig macht und so sehr einfache Rekursionen erlaubt.

Beim Löschen einer Liste müssen natürlich beide Teile der Liste und der Knoten selbst gelöscht werden. Auch hier wieder sind keinerlei Fallunterscheidungen nötig und die Frage nach der Abbruchbedingung, die bei Rekursionen gerne vergessen wird, stellt sich gar nicht.

Nun brauchen wir noch Element–Objekte für die Liste. Zahlen haben wir ja schon, Strings wären auch noch schön. Hier sind sie:

```
\ Data structures: string                                30apr93py

int class string
how:    : !     ( addr count -- )
                value over 1+ SetHandleSize
                value F @ place ;
        : @     ( -- addr count )  value F @ count ;
        : .     @ type ;
        : init  ( addr count -- )
          dup 1+ value Handle! ! ;
        : null  S" " new ;
        : dispose  ref F @ 1- 0> 0=
          IF  value HandleOff  THEN  super dispose  ;
class;
```

Wir leiten die Klasse `string` von `int` ab. Deren Instanzvariable `value` verwenden wir hier als Handle, als Zeiger auf einen verschiebbaren Speicherbereich. Dort ist der String dann als counted String gespeichert. Beim Speichern eines neuen Strings muß natürlich die Größe des Speicherblocks angepaßt werden; beim erstmaligen Anlegen muß er überhaupt erst angefordert und beim Löschen natürlich wieder freigegeben werden. Alles andere erklärt sich jetzt hoffentlich ziemlich von selbst.

Sehr brauchbar ist auch eine Pointer–Klasse. Zwar kann man Pointervariablen direkt erzeugen, aber nicht z. B. in eine Liste einhängen.

```
\ Data structures: pointer                               30apr93py

data class pointer
public: data ptr container
```

```
        method ptr!
how:    : !     container ! ;
        : @     container @ ;
        : .     container . ;
        : #     container # ;
        : init  ( data -- )  dup >o 1 ref +! o> bind container ;
        : ptr!  ( data -- )  container dispose  init ;
        : dispose  -1 ref +!  ref F @ 0> 0=
          IF  container dispose super dispose  THEN ;
        : null  nil new ;
class;
```

Analog zur Liste wird eine Pointer–Instanzvariable angelegt (`container`), dazu gibt's noch eine Methode `ptr!`, mit der man ein Objekt zuweisen kann. Die Methoden `@`, `!`, `.` und `#` werden an den Container durchgereicht. Die Init–Methode bindet ein übergebenes Objekt an den Zeiger. `ptr!` gibt zuerst das vorherige Objekt frei und speichert anschließend das neue Objekt. Dabei wird natürlich auch das Reference Counting berücksichtigt.

Löschen eines Zeigerobjekts bedeutet ebenfalls, daß auf das Objekt eine Referenz weniger weist (es also ggf. gelöscht werden muß); anschließend wird der Pointer gelöscht.

Analog zu einem Pointer kann man gleich ein ganzes Array von Pointern einbinden:

```
\ Data structures: array                                    30apr93py

data class array
public: data [] container
        cell var range
how:    : !     ( <value> n -- )  container ! ;
        : @     ( n -- <value> )  container @ ;
        : .     '[
                # 0 ?DO emit I container . ', LOOP drop ." ]" ;
        : init  ( data n -- )  range F !  bind container ;
        : dispose  -1 ref +!  ref F @ 0> 0=
          IF  # 0 ?DO  I container dispose  LOOP
              super dispose  THEN ;
        : null  ( -- addr )  nil 0 new ;
        : #     ( -- n )     range F @ ;
        : atom? ( -- flag )  false ;
class;
```

Analog zur Methode `new` legt man mit `new[]` ein ganzes Array von Objekten an — die dann auch der gleichen Klasse angehören; also voneinander einen konstanten Abstand haben. Über diesen Abstand wird dann die n-te Objektadresse berechnet (die erste liegt bei 0). Die Array–Variable erwartet also eine Indexnummer auf dem Stack.

1.4. Tools und Anwendungsbeispiele

Insgesamt ist das Listenpaket jetzt noch nicht sehr einfach zu bedienen. Ich habe deshalb ein paar kleine Tools geschrieben, die den Umgang erleichtern — aber keineswegs ein vollwertiges Lisp oder sowas daraus machen:

```
\ Data structure utilities                                  17nov93py

: cons  linked new ;
: list  nil cons ;
: car   >o lists first self o> ;
: cdr   >o lists next  self o> ;
: print >o data . o> ;
: ddrop >o data dispose o> ;
: make-string string new ;
: $"    state @ IF  compile S" compile make-string exit  THEN
        '" parse make-string ;       immediate
```

`cons` und `list` helfen einem beim Anlegen einer Liste. `cons` verknüpft zwei auf dem Stack liegende Objekte zu einer Liste (TOS als next, sollte also eine Liste sein; NOS als first der Liste). `list` faßt ein Objekt zusammen mit nil zu einer Liste zusammen.

`car` und `cdr` sollten aus Lisp bekannt sein, sie liefern das erste Element bzw. den Rest der Liste.

`print` ruft die Ausgabemethode eines Objekts auf.

`ddrop` schließlich entfernt und löscht ein Objekt.

`make-string` ist der String-Konstruktor, analog zu `list`. Einen festen String konstruiert `$"`. Als Beispiel, wie man mit diesen Tools eine Liste aufbaut:

```
$" Dies" $" ist" $" ein" list cons $" Test" list cons cons  ok
dup print (Dies (ist ein) Test) ok
pointer : test  ok
test . (Dies (ist ein) Test) ok
test # . 3 ok
```

2. Die vollständige Sprachbeschreibung

2.1. Semantik der Objektschnittstelle

Die Schnittstellen zur Objektorientierten Programmierung in bigFORTH teilen sich in drei Bereiche auf:

- Tools zum Verwalten der Objekte, die selbst nicht objektbezogen sind und die Klassen, von denen alle anderen selbstprogrammierten Klassen und Objekte abstammen müssen
- Tools zum Anlegen vom Instanz–Variablen und Methoden
- und die Methoden der Wurzelklasse, die Erzeugen von neuen Klassen, Instanzen, Handling von Objektpointern und ähnliches erlauben.

Nur die Wörter des ersten Punkts sind von FORTH aus direkt zugänglich. Die Wörter zum zweiten Punkt sind nur während der Deklaration einer Objektklasse benutzbar, und die Wörter des dritten Punkts sind gar keine Wörter, sondern Methoden von Objekten.

bigFORTH benutzt einen besonders konsequenten Weg, Klassen zu verwalten: Auch Klassen sind Objekte, allerdings mit klassenglobalen Instanzvariablen, die lediglich dazu dienen, neue Klassen und Objekte der Klasse zu erzeugen und zu bearbeiten. Klassen

dienen auch dazu, Methoden an Objekte zu schicken, deren Adresse im Objektpointer steht und deren Kontext explizit angegeben werden muß (weil es sich nicht um das gerade definierte Objekt handelt), dienen also einer Art Casting.

ˆ (−− o): Liefert den Zeiger auf das gerade aktive Objekt (User–Variable OP)

>O (o −−) (OS −− o): Schiebt den Zeiger auf das Objekt **o** auf den Objektstack. Das Objekt wird dadurch zum aktuellen Objekt. Achtung: Das vorher benutzte Objekt wird auf den Returnstack gepusht, Objektzugriffe müssen also mit anderen Returnstackzugriffen wie DO LOOPs und >R und R> ausbalanciert werden.

O> (−−) (OS o −−): Nimmt den Zeiger auf das aktuelle Objekt vom Objektstack. Der davor benutzte Objekt wird vom Returnstack zurück in den Objektpointer geladen.

O@ (−− addr): Liefert die Adresse der Methodentabelle des aktuellen Objekts.

BIND (o −−) ⟨*name*⟩: Bindet das Objekt **o** an den Objektpointer ⟨*name*⟩. **o** muß der Klasse von ⟨*name*⟩ oder einer davon abgeleiteten Unterklasse angehören.

DYNAMIC (−−): Objekte werden mit NEW dynamisch im Heap angelegt. Das ist das Default–Verhalten von NEW.

STATIC (−−): Objekte werden mit NEW statisch im Dictionary angelegt. Damit kann man sich Objektstrukturen zusammencompilieren, mit Savesystem speichern und nach dem Laden wieder benutzen, vorausgesetzt, die Objekte selbst verwenden keine weiteren Funktionen um dynamischen Speicher anzufordern.

Jedes Objekt besteht aus Variablen und Methoden, die deklariert werden müssen. Die Methoden müssen anschließend noch implementiert werden. Die Sichtbarkeit dieser Variablen und Methoden nach außen kann festgelegt werden: Private Methoden und Variablen sind nur für die Klasse selbst und ihre Unterklassen sichtbar, von außen sichtbare Methoden oder Variablen müssen als “public” deklariert werden.

bigFORTH trennt Deklarationsteil und Implementierungsteil der Klassen voneinander. Beide zusammen bilden die Definition einer Klasse.

TYPES (−−) (VS voc −− TYPES): Alle Wörter, die zur Deklaration von Klassen und Implementierung von Methoden dienen, sind im Vokabular TYPES. TYPES muß in der Suchordnung immer ganz oben liegen, da sonst Konflikte auftreten würden (: z. B. ist sowohl in TYPES, im aktuellen Public–Thread als auch in FORTH vorhanden).

PUBLIC: (−−): Schaltet auf öffentliche Deklaration um. Alle weiteren Methoden und Variablen sind sichtbare Schnittstellen zum Objekt.

PRIVATE: (−−): Schaltet auf private Deklaration um (Default–Zustand). Alle weiteren Methoden und Variablen können nur von der deklarierten Objekt–Klasse und ihren Unterklassen benutzt werden.

VAR (size −−) ⟨*name*⟩: Legt eine Instanz–Variablen der Größe **size** an.

STATIC (−−) ⟨*name*⟩: Legt eine Variable an, die allen Objekten einer Klasse gemeinsam ist. Diese Variable ist eine Zelle groß und wird uninitialisiert als Pointer angelegt.

METHOD (−−) ⟨*name*⟩: Deklariert eine Methode. Solche Methoden werden spät gebunden, falls nicht klar ist, von welcher Klasse sie ausgeführt werden.

EARLY (−−) ⟨*name*⟩: Deklariert eine immer früh gebundene Methode. Solche Methoden können von Unterklassen nicht verändert werden. Soll der Name nochmal benutzt werden, muß die Methode erneut deklariert werden.

DEFER (−−) ⟨*name*⟩: Deklariert eine objektspezifische Methode, die für jedes Objekt eine eigene Aktion ausführen kann und auch mit IS verändert werden kann. Damit können z. B. in einer graphischen Benutzeroberfläche die Callbacks einzelner Objekte gebunden werden.

PTR (−−) ⟨*name*⟩: Deklariert einen Objektpointer, der auf ein Objekt der gerade deklarierten Klasse oder einer ihrer Unterklassen zeigen kann und mit BIND initialisiert werden muß.

ASPTR (class −−) ⟨*name*⟩: „Castet“ einen mit PTR angelegten Pointer auf die deklarierte Klasse und legt den gecasteten Pointer unter ⟨*name*⟩ an

F (−−) ⟨*name*⟩: Compiliert ⟨*name*⟩ mit FORTH als erstes Vokabular im Suchpfad.

HOW: (−−): Schaltet vom Deklarationsteil zum Implementierungsteil um. Hier werden statische Variablen initialisiert und Methoden implementiert.

: (−−) ⟨*name*⟩: Implementiert die Methode ⟨*name*⟩. Die Implementierung wird mit ; abgeschlossen.

CLASS; (−−): Beendet die Definition einer Klasse.

Die Verwaltung von Klassen und Objekten wird von den Klassen selbst vorgenommen. Dazu stellt die Wurzelklasse OBJECT einige Methoden und klassenglobale Variablen zur Verfügung. Diese lassen sich in verschiedene Gruppen aufteilen:

- Klassen–Browser
- Unterklassen–Erzeugung
- Speicher–Verwaltung, Instanzenerzeugung
- Binding

Class **OBJECT" (... −− ...) ⟨*method*⟩:**Ist die übervaterklasse aller Objektklassen. Führt ⟨*method*⟩ aus bzw. compiliert sie dynamisch gebunden im Kontext des aktuellen Objekts.

Public:

Static Variable **PARENTO (−− addr):** Zeigt auf die Elter–Klasse

Static Variable **CHILDO (−− addr):** Zeigt auf die zuletzt abgeleitete Kind–Klasse

Static Variable **NEXTO (−− addr):** Zeigt auf das nächst–„ältere“ Geschwister derselben Elter–Klasse

Static Variable **NEWLINK (−− addr):** Zeigt auf eine Liste aller Objekte, die Speicher im Objekt belegen und dient der internen Speicherverwaltung

Early Method **CLASS (−−) ⟨*class*⟩:** Leitet die Deklaration der Unterklasse ⟨*class*⟩ ein

Early Method **CLASS? (object −− flag):** Prüft die Klassenbeziehungen nach. **flag** ist nur true, wenn **object** einer Vaterklasse des Objekts angehört, das CLASS? ausführt.

Early Method **SEAL (−−):** Macht die private Methoden– und Variablenkette unsichtbar und verhindert eine weitere Benutzung bei erneuter Vererbung

Method **INIT (... −−):** Initialisiert ein Objekt mit den Parametern .... INIT wird beim Neuanlegen eines Objekts auch für alle Objekte aufgerufen, die als Instanzvariablen verwendet werden; und zwar der Reihenfolge nach, in der sie deklariert wurden, zuerst aber für das neu anzulegende Objekt selbst. INIT ist eine polymorphe Methode.

Early Method **NEW (−− object) immediate:** Legt ein (namenloses) Objekt der angegebenen Klasse neu an

Early Method **NEW[] (n −− object) immediate:** Legt ein Array (namenloser) Objekte der angegebenen Klasse mit **n** Elementen an

Method **DISPOSE (−−):** Gibt den Speicherplatz eines Objekts wieder frei. DISPOSE ist ein polymorphe Methode.

Early Method **: (−−)** ⟨***name***⟩**:** Legt ein Objekt unter dem Namen ⟨***name***⟩ an

Early Method **PTR (−−)** ⟨***name***⟩**:** Legt einen Pointer auf ein Objekt unter dem Namen ⟨***name***⟩ an. Ein Objekt muß noch mit BIND an den Pointer gebunden werden.

Early Method **ASPTR (class −−)** ⟨***name***⟩**:** "Castet" einen mit PTR angelegten Pointer auf die aufgerufene Klasse und legt den gecasteten Pointer unter ⟨***name***⟩ an

Early Method **[] (n −−)** ⟨***name***⟩**:** Legt ein Array von Objekten mit **n** Elementen unter dem Namen ⟨***name***⟩ an

Early Method **:: (−−)** ⟨***method***⟩ **immediate:** Bindet ⟨***method***⟩ early.

Im Implementierungsteil einer Klasse direkt aufgerufen, vererbt es Methoden „quer" von anderen Klassen, erlaubt also eingeschränkte Mehrfachvererbung. Die Methode muß dazu bereits in einer gemeinsamen Vaterklasse definiert sein, ererbt wird lediglich die Codeadresse für die Methode.

Early Method **SUPER (−−)** ⟨***method***⟩ **immediate restrict:** Bindet ⟨***method***⟩ der Superklasse early. SUPER wird benutzt, um vererbtes Verhalten zwar zu modifizieren, aber auf das ursprüngliche Verhalten zugreifen zu können.

Early Method **GOTO (−−)** ⟨***method***⟩ **immediate restrict:** Dient zum Entrekursivieren. Die Methode ⟨***method***⟩ wird direkt angesprungen, ohne die Rückkehradresse (und ggf. die alte Objektadresse) auf den Returnstack zu legen.

Early Method **SELF (−− addr):** Liefert die Adresse des Objekts

Early Method **BIND (object −−)** ⟨***pointer***⟩ **immediate:** Speichert die Adresse **objekt** in der Pointer–Variable ⟨***pointer***⟩. Dabei wird eine Typprüfung vorgenommen. **objekt** muß der Klasse von ⟨***pointer***⟩ oder einer deren Unterklassen angehören.

Early Method **LINK (−− class addr)** ⟨***name***⟩**:** Berechnet die Referenz des Objektpointers ⟨***name***⟩, also seine Adresse und Klasse

Early Method **BOUND (object class addr −−):** Speichert das Objekt **object** in der Referenz **class addr** ab und nimmt dabei eine Typprüfung vor

Early Method **IS (xt −−)** ⟨***deferred***⟩ **immediate:** Weist der ⟨***deferred***⟩ method die Adresse **xt** eines Wortes zu, das beim Aufruf von ⟨***deferred***⟩ dieses Objekts ausgeführt werden muß.

Einige Variablen sind nicht von außen zugänglich, können aber zu Debugzwecken in Unterklassen verwendet werden:

Private:

Static Variable **PUBLIC (−− addr):** Zeigt auf den Wörter–Thread aller öffentlichen Variablen und Methoden

Static Variable **PRIVATE (−− addr):** Zeigt auf den Wörter–Thread aller privaten Variablen und Methoden

Static Variable **METHOD# (−− addr):** Enthält die Anzahl Methoden/statischer Variablen in Bytes

Static Variable **SIZE (−− addr):** Enthält die Größe eines Objekts in Bytes

Variable **OBLINK (−− addr):** Erste Instanzvariable: Zeigt auf die Methodentabelle des Objekts

Für Debugging–Zwecke gibt es das Objekt DEBUGGING. Es enthält weitere Methoden, die beim Debugging hilfreich sind.

CLASS **DEBUGGING″ (... −− ...)** ⟨***method***⟩**:**Ist eine Hilfsklasse, die nötige Werkzeuge zum Debuggen von Objekten bereitstellt. Ansonsten wie OBJECT.

PUBLIC:

EARLY METHOD **words (−−):** Listet die Wörter im Public– und im Private–Vokabular auf

EARLY METHOD **' (−− cfa)** ⟨***Name***⟩**:** Findet die **cfa** einer Methode oder Objektvariable ⟨***Name***⟩

EARLY METHOD **see (−−)** ⟨***Name***⟩**:** Decompiliert ⟨***Name***⟩

EARLY METHOD **view (−−)** ⟨***Name***⟩**:** Ruft den Editor an der Deklaration von ⟨***Name***⟩ auf

EARLY METHOD **trace' (.. −− ..)** ⟨***Name***⟩**:** Trace der Methode ⟨***Name***⟩

EARLY METHOD **debug (−−)** ⟨***Name***⟩**:** Die Methode ⟨***Name***⟩ wird bei den nächsten Aufrufen getracet

2.2. Formale Syntax

⟨*declaration*⟩ ::=
  ⟨*parent*⟩ `CLASS` ⟨*object*⟩
  {[`private` : |`public` :] ⟨*creator*⟩ ⟨*selector*⟩ }
  [`HOW` : {: ⟨*method*⟩ ⟨*coding*⟩ ; }]
  `CLASS`;
⟨*creator*⟩ ⟨*selector*⟩ ::=
  `STATIC` ⟨*static*⟩ |`METHOD` ⟨*method*⟩ |`EARLY` ⟨*method*⟩ |
  ⟨*number*⟩ `VAR` ⟨*var*⟩ | ⟨*object*⟩ (:|[]|`PTR`) ⟨*instance*⟩ |
⟨*parent*⟩ ::=
  `OBJECT`| ⟨*object*⟩
⟨*creation*⟩ ::=
  ⟨*object*⟩ (:|`PTR`) ⟨*instance*⟩ | ⟨*number*⟩ ⟨*object*⟩ [] ⟨*instance*⟩
⟨*coding*⟩ ::=
⟨*word*⟩ ⟨*coding*⟩ |{⟨*instance*⟩ } ⟨*selector*⟩ ⟨*coding*⟩ |

10 MINOΣ Dokumentation

1. Was ist MINOΣ?

1.1. Was bedeutet Visual?

Auf der Forth-Tagung 1996 war der Wunsch nach einem "Visual FORTH" deutlich zu hören, besonders von Friedrich Prinz. Vorbild sind Programmiersysteme wie Visual BASIC (Microsoft) und Delphi (Borland). Auch für C++ und sogar für Java gibt es inzwischen Ähnliches.

Im Kern sind solche Programmiersysteme Formular–Malprogramme. Sie bestehen zum einen aus einer Bibliothek mit vielfältigen Elementen einer grafischen Benutzeroberfläche, also Fenster, Buttons, Edit–Controls, Zeichenfeldern, etc.. Zum anderen gibt es einen Editor, in dem man mit der Maus diese Elemente zu einem Formular zusammenstellt, und den noch fehlenden Code einträgt.

Typischer Anwendungsfall sind Datenbank–Applikationen. Deshalb sind in vielen solchen Systemen bereits Datenbanken enthalten oder über eine offene Datenbankschnittstelle angebunden.

Ein weiterer wichtiger Aspekt sind komplexe Komponenten. Damit kann man etwa einen Web–Browser mit einigen Mausklicks und ein paar Zeilen Code zusammenfügen. Allerdings verstecken solche komplexen Komponenten meist ihre Einzelteile — mit einem fertigen Web–Browser ist man dann auch nicht unbedingt schlechter bedient.

In der Regel ist die Interaktivität solcher Werkzeuge nicht gerade berauschend. Man erstellt sein Formular, schreibt seine Aktionen als Code und compiliert das dann mehr (Delphi) oder weniger (Visual Age for C++) schnell. Ausprobieren kann man also erst nach einem Compilerlauf.

1.2. Wozu Visual?

Grafische Benutzeroberflächen muß man ja nicht notwendigerweise zusammenpinseln, ebensowenig wie man Texte notwendigerweise mit WYSIWYG bearbeiten muß. Viele Textsatzfunktionen sind eher inhaltlicher als visueller Kathegorie, etwa ob ein Text eine Überschrift ist oder hervorgehoben sein soll. Eben das gilt grundsätzlich auch für Benutzeroberflächen, zum Teil sogar wesentlich stärker. Nicht der Programmierer soll entscheiden, in welchem Font und welcher Fontgröße die Benutzeroberfläche gestaltet ist — das ist Sache des Benutzers. Ebenso die Farbe der Buttons und Texte.

Auch für die räumlichen Beziehungen der einzelnen Schaltelemente zueinander ist mehr Abstraktion als die Angabe von Position, Höhe und Breite sinnvoll. Typischerweise werden Buttons neben- oder übereinander angeordnet, evtl. mit einem standardisierten Abstand zwischen ihnen. Die Größe der Buttons muß sich einerseits an den enthaltenen Strings ausrichten, andererseits an ästhetischen Gesichtspunkten (etwa alle Buttons einer Leiste gleich groß).

Ein solches abstraktes Modell, etwa an TEX's Boxes&Glues angelehnt, programmiert sich aber bereits ohne visuellen Editor ganz gut. Für den eigentlichen „Satz" der Buttons und Boxen ist nicht der Programmierer verantwortlich. Er gibt nur das Konzept vor. Diese Vorgehensweise ist unter Unix auch gebräuchlich. Motif und Tcl/Tk nutzen

Nachbarschaftsbeziehungen, Interviews nutzt das Boxes&Glues–Konzept. Ich habe mich für das Boxes&Glues–Konzept entschieden, weil das einen sehr schnelle Formatierung der Objekte erlaubt.

Diese Konzepte stehen einer grafischen Bearbeitung der Dialoge teilweise im Wege, da hier ja nicht abstrakte Konzepte (soll ,,neben“ ein Objekt), sondern Positionen angegeben werden.

1.3. Visual Forth?

Ein Punkt, der nachdenklich stimmt: In den Paketen, die tatsächlich visuelle Formularprogrammierung erlauben, steckt sehr viel Arbeit. Mikrosoft, Borland und IBM können es sich leisten, hunderte von Programmierern nur für so ein Projekt zu beschäftigen. Diese Man–Power steht für ein Forth–Projekt einfach nicht zur Verfügung. Aber halt:

- Forth behauptet, einen Programmierer viel effizienter arbeiten zu lassen.
- Ein Team von 300 Mann blockiert sich gegenseitig. Teilt man Arbeit auf, so ergibt sich für den Programmierer die Notwendigkeit, seine (auch die vorerst nur geplanten) Funktionen zu dokumentieren und die Dokumentation der Funktionen der anderen zu lesen und zu verstehen. Jeder weiß, daß Dokumentieren viel länger dauern als das eigentliche Programmieren. Ab einer gewissen Projektgröße bleibt dann keine Zeit mehr zum eigentlichen Programmieren übrig; alle Zeit wird verwendet, geplante Funktionen zu spezifizieren, und die Spezifikationen von anderen Programmierern zu lesen. Oder man unterhält sich einfach vor der Türöffnung des ohnehin viel zu engen und zu lauten Cubicles.
- Ein guter Programmierer arbeitet oft 20 mal schneller als ein schlechter, obwohl er pro Zeiteinheit nicht mehr Zeichen tippen kann. Das resultierende Programm ist einfach bis zu 20 mal kürzer oder hat 20 mal weniger Fehler (oder beides) — und das bei größerer Funktionalität.
- Auch bei großen Projekten wird aus diesen Gründen die eigentliche Arbeit von einem kleinen “Core Team” erledigt. Und da gilt dann die Dilbert–Regel: Was man mit zwei Leuten machen kann, kann man für höchstens die Hälfte der Kosten auch mit einem Mann machen.

Außerdem gibt's für bigFORTH–DOS schon ein ,,Text-GUI“, wenn auch ohne Editor, und mit einem abstrakten Boxes&Glues–Konzept, das ja, wie oben behauptet, einem grafischen Editor eher im Weg steht.

Schließlich wollte ich vom DOS loskommen und bigFORTH auf ein richtiges Betriebssystem (Linux) portieren. Anders als unter Windows oder unter OS/2 werden hier Oberfläche und Bildschirmzugriff getrennt. Der Bildschirmzugriff erfolgt über das X Windows System (kurz X), die eigentliche Oberfläche wird mit einer Library implementiert. Aus diesem Grund gibt es auch keine einheitliche Oberfläche, sondern verschiedene Libraries, wie die Athena Widgets, Motif, Tcl/Tk, xforms, etc.. Als ,,Look and Feel“ haben sich Motif–artige Buttons inzwischen durchgesetzt, auch unter Windows und MacOS.

All diese Libraries haben ihre Nachteile. Die Athena Widgets sind hoffnungslos veraltet. Motif ist kommerziell, kostet also, auch wenn mit Lesstif eine freie Variante im Entstehen ist. Außerdem ist es langsam und verbraucht Unmengen Speicher. Tcl/Tk ist zwar nicht so speicherintensiv, dafür aber *noch* langsamer. Wie macht man aber dem Benutzer klar,

daß das Zeichnen eines Fensters im Sekundenbereich liegt, während Quake 3D–Grafik flüssig animiert?

Deshalb habe ich mich entschlossen, die Widget–Klassen von bigFORTH–DOS auf X zu portieren, und dafür dann einen geeigneten Editor zu schreiben. Außerdem fügen sich solche in Forth geschriebene Widget–Klassen natürlich besser in die Entwicklungsumgebung ein und sind — vom Forth–Standpunkt — auch einfacher zu warten. Es gibt ja nicht deshalb so viele Widget–Libraries in C, weil das so einfach mal an einem Nachmittag geschrieben wäre, sondern weil die vorhandenen nie die Anforderungen erfüllt haben, und eine Modifikation aussichtslos erschien.

1.4. Der Name — warum MINOΣ?

„Visual XXX“ ist eigentlich ein Allerweltsname. „Forth“ darf man sowieso nicht mehr sagen, schließlich besteht der kommende Markt aus einer Milliarde Chinesen, und für die ist 4 nunmal eine Unglückszahl (weil „se“ (vier) ziemlich ähnlich klingt wie „se“ (tot)). Naja, Borland nennt sein Produkt ja auch nicht „Visual TurboPascal“, sondern Delphi. Was an „Oracle“ erinnern soll, und die Assoziationen zu Datenbankanwendungen wecken soll.

Griechisch oder ähnliches ist auf alle Fälle gut, schließlich lehnt sich die Library an das Boxes&Glues–Modell von TEX an, und das wird ja auch griechisch ausgesprochen. Dazu ist die Library im Vergleich zu Motif recht kompakt (MINimal), und da es für Linux ist, ist auch der phonetische Abstand nicht arg groß...

2. Widget–Klassen

Die grundsätzliche Klassenhierarchie ergab sich natürlich aus dem vorhandenen Material für DOS. Dort werden grob Widgets („Window Gadgets“) und Displays unterschieden. Displays sind Widgets, die auch zeichnen können, also Fenster, Viewports, Backing Stores und Doublebuffers. Sie sind für die Umsetzung des abstrakten Interfaces auf die konkrete Grafik–Bibliothek verantwortlich, für die Verwaltung der Ereignisse (Mausklicks, Tastendrücke, Redraws, etc.).

Die Widgets selbst teilen sich auf in Boxes (horizontal und vertikal orientiert), Buttons, Switches, Labels, Icons, Texteingabefelder, Slider, Scaler, Canvas... derzeit insgesamt 76 Klassen.

Ursprünglich waren die Aktionen, die beim Klick auf einen Button ausgeführt wurden, einfache Forth–Wörter. Es hat sich aber herausgestellt, daß das nicht zweckmäßig ist. Bei vielen Aktionen reicht es völlig aus, etwa eine Variable zu setzen. Deshalb werden die Aktionen jetzt von Objekten generiert. Auch hier gibt es neun verschiedene Klassen. Für anspruchsvollere Zwecke, etwa einer kontextsensitiven Hilfe oder einem Kontextmenü, könnten hier weitere Klassen erzeugt werden, die dann etwa ein Fenster erzeugen, wenn sich die Maus über dem Objekt befindet, und dieses Fenster wieder verschwinden lassen, wenn die Maus das Objekt verläßt. Damit wäre das Problem vielfältiger Reaktionen auf verschiedene Ereignisse mit einfachen Mitteln gelöst, ohne den Normalfall aufzublähen.

Zu den Displays gehört noch eine Klasse, die der Resourcen. Sie enthält die bildschirmspezifischen Daten, also Display, Screen, Fonts, Farben, Color–Map, Cursor und den Zeichenkontext.

Zu einer Klassenhierarchie gehört ein gemeinsames Protokoll, also Methoden und Variablen, die alle Klassen verstehen. Die wichtigsten Elemente der Protokolle für Widgets (Abbildung 10.1) und Displays (Abbildung 10.2) sollen nun vorgestellt werden.

Methode	Zweck
PARENT	Zeigt auf das Vater–Objekt
WIDGETS	Zeigt auf das nächste Objekt
DPY	Das Display des Objekts
INIT	Das Objekt wird initialisiert
DISPOSE	Das Objekt wird gelöscht
HGLUE	Horizontale Dehnung
VGLUE	Vertikale Dehnung
XINC	Horizontale Schrittweite
YINC	Vertikale Schrittweite
XYWH	Umrandendes Rechteck
RESIZE	Größenveränderung
REPOS	Positionsveränderung
RESIZED	Die Größe neu berechnen
!RESIZED	Alle Größenparameter neu
CLOSE	Schließt das Fenster
DRAW	Zeichnet das Objekt
ASSIGN	Weist einen neuen Inhalt zu
CLICKED	Das Objekt wurde angeklickt
KEYED	Eine Taste wurde gedrückt
INSIDE?	Ist der Punkt im Objekt?
HANDLE-KEY?	Versteht es Tastendrücke?
FOCUS	Objekt im Fokus
DEFOCUS	Objekt aus dem Fokus
SHOW	Das Objekt ist sichtbar
HIDE	Das Objekt ist unsichtbar
MOVED	Mauszeiger über dem Objekt
LEAVE	Mauszeiger nicht mehr darüber
DELETE	Objekt aus der Liste löschen
APPEND	Objekt in Liste einfügen
SHOW-YOU	Objekt soll sich zeigen
FIRST-ACTIVE	Das erste Objekt aktiv setzen
NEXT-ACTIVE	Das nächste Objekt wird aktiv
PREV-ACTIVE	Das vorherige Objekt wird aktiv

Abbildung 10.1: Widget–Messages

Methode	Zweck
XRC	X Resource
LINE	Linie zwischen zwei Punkten
TEXT	Text zeichnen
IMAGE	Pixmap zeichnen
BOX	Rechteck zeichnen
MASK	Icon zeichnen
FILL	Polygon füllen
STROKE	Polygon zeichnen
DRAWER	Zeichenroutine aufrufen
DRAWABLE	Resourcen zum Zeichnen
SYNC	Update beenden
MAP	Fenster anzeigen
UNMAP	Fenster iconifizieren
MOUSE	Mausposition
SCREENPOS	Bildschirmposition
TRANS	Koordinatenumrechnung
TRANS'	Koordinaten revers
TRANSBACK	Umrechnung zu GET–WIN
GET–DPY	Oberstes Fenster
GET–WIN	Übergeordnetes Fenster
SET–FONT	Zeichensatz setzen
SET–COLOR	Farbe setzen
SET–CURSOR	Mauszeiger setzen
TXY!	Tile–Offset setzen
CLIP–RECT	Zeichenausschnitt setzen
GET–EVENT	Ereignis holen
HANDLE–EVENT	Ereignisse bearbeiten
SCHEDULE–EVENT	Ereignisse verteilen
CHILD–MOVED	Verteilt Mausbewegungen
CLICK	Auf Mausklick warten
CLICK?	Mausklick abfragen
MOVED?	Hat sich die Maus bewegt?
MOVED!	Maus auf ,,bewegt“ setzen
SHOW–ME	Zeigt Objekt an (x,y)
SCROLL	Scrollt an (x,y)
CLIPX	Horizontale Begrenzung
CLIPY	Vertikale Begrenzung
GEOMETRY	Resize in Objektkoordinaten
>EXPOSED	Auf Sichtbarwerden warten

Abbildung 10.2: Display–Messages

Abgeleitete Objekte haben natürlich zusätzliche Variablen und Objekt–Pointer und evtl. zusätzliche Methoden.

Die Display–Klasse ist von der Widget–Klasse abgeleitet. Deshalb versteht sie alle Messages der Widget–Klasse. Einige Displays wie Viewports, Backing Store und Double–Buffer lassen sich auch wie normale Widgets als Teil eines Dialogs oder Fensters verwenden.

2.1. Vollständige Klassenhierarchie

Die Klassenhierarchie gibt zusätzlich die Bytes pro Objekt (für Variablen) und die Größe der Methodentabelle (pro Klasse) an. Einrückungen stellen die Klassenhierarchie dar.

Aktoren

Die vorhandenen Aktoren konzentrieren sich vor allem auf Toggle– und Radio–Buttons. Diese Knöpfe können zwei verschiedene Zustände einnehmen — gesetzt und nicht gesetzt. Dabei kann das Ziel der Aktion das Setzen einer Flag (`toggle-var`) oder einer Variable mit einer Zahl (`toggle-num`) sein, oder es können Aktionen beim Setzen und Löschen (`toggle`) bzw. zum Abfragen und Ändern (`toggle-state`) definiert werden. Slider und Scaler (mit Maximalwert und Schrittweite) werden ähnlich `toggle-state` behandelt.

Class **ACTOR (... –– ...)** ⟨***method***⟩**:** Abstract data type, provides the common interface of all actors.

Public:

Init–Parameter (o ––): Sets the called object

OBJECT Pointer **CALLED (... –– ...)** ⟨***method***⟩**:** This variable points to the object that is being called

GADGET Pointer **CALLER (... –– ...)** ⟨***method***⟩**:** This variable points to the object that is calling (the widget).

Method **SET (––):** sets the flag

Method **RESET (––):** resets the flag

Method **TOGGLE (––):** toggles the flag

Method **FETCH (–– x1 .. xn):** queries the value(s)

Method **STORE (x1 .. xn ––):** changes the value(s)

Method **CLICK (x y b n ––):** performs the action for a click

Method **KEY (key sh ––):** performs the action for a keystroke

Method **ENTER (––):** performs the action for entering the widget

Method **LEAVE (––):** performs the action for leaving the widget

Method **ASSIGN (x1 .. xn ––):** initially assigns the state

Method **SET–CALLED (o ––):** sets the called object

Class **TOGGLE (... –– ...)** ⟨***method***⟩**:** models a flag with initial state and two functions for each state

Public:

Init–Parameter (o state xtset xtreset ––):

Variable **DO–SET (–– addr):**

Variable **DO–RESET (–– addr):**

Variable **SET? (–– addr):**

Method **ASSIGN (flag ––):**

Method **FETCH (–– flag):**

Method **STORE (flag −−):**
Method **CLICK (x y b n −−):**

Class **TOGGLE–VAR (... −− ...)** ⟨***method***⟩**:** keeps the flag in addr, and executes xt on changes

Public:

Init–Parameter (o addr xt −−):
Variable **ADDR (−− addr):**
Variable **XT (−− addr):**
Method **FETCH (−− n):**
Method **STORE (n −−):**
Method **ASSIGN (addr −−):**

Class **TOGGLE–NUM (... −− ...)** ⟨***method***⟩**:** is responsible for state n in addr (sets addr to n when set), and executes xt on changes

Public:

Init–Parameter (o num addr xt −−):
Variable **NUM (−− addr):**
Method **ASSIGN (num addr −−):**
Method **FETCH (−− flag):**
Method **STORE (n −−):**

Class **TOGGLE–STATE (... −− ...)** ⟨***method***⟩**:** allows generic fetch and store functions

Public:

Init–Parameter (o xtstore xtfetch −−):
Variable **DO–STORE (−− addr):**
Variable **DO–FETCH (−− addr):**
Method **FETCH (−− x1 .. xn):**
Method **STORE (x1 .. xn −−):**

Class **SIMPLE (... −− ...)** ⟨***method***⟩**:** xt is executed at every store (no state maintained)

Public:

Init–Parameter (o xt −−):
Variable **DO–IT (−− addr):**
Method **FETCH (−− 0):**
Method **STORE (x −−):**

Class **CLICK (... −− ...)** ⟨***method***⟩**:** Action for handling clicks and drag&drop operations

Public:

Init–Parameter (o xt −−):
Method **CLICK (x y b n −−):**
Method **FETCH (−−):**
Method **STORE (x y b n −−):**

Class **DRAG (... −− ...)** ⟨***method***⟩**:** Calls toggle on each click event

Public:

Init–Parameter (o xt −−):

METHOD **CLICK (x y b n ––):**

CLASS **REP (... –– ...)** ⟨***method***⟩**:** Calls toggle repeatedly while the user holds down the mouse button

PUBLIC:

Init–Parameter (o xt ––):
METHOD **CLICK (x y b n ––):**

CLASS **DATA–ACT (... –– ...)** ⟨***method***⟩**:** Simple action which can preserve data

PUBLIC:

Init–Parameter (o data xt ––):
VARIABLE **DATA (–– addr):**
METHOD **STORE (––):**

CLASS **SCALE–ACT (... –– ...)** ⟨***method***⟩**:** Generic slider actor (maximum slider position provided)

PUBLIC:

Init–Parameter (o xtstore xtfetch max ––):
VARIABLE **MAX (–– addr):**
METHOD **ASSIGN (max ––):**
METHOD **FETCH (–– max):**

CLASS **SLIDER–ACT (... –– ...)** ⟨***method***⟩**:** Generic scaler actor (maximum scaler position provided)

PUBLIC:

Init–Parameter (o xtstore xtfetch max step ––):
METHOD **ASSIGN (max step ––):**
METHOD **FETCH (–– max step):**

CLASS **SCALE–VAR (... –– ...)** ⟨***method***⟩**:** Scaler actor, keeps position and maximum value in own variables.

PUBLIC:

Init–Parameter (o pos max ––):
VARIABLE **MAX (–– addr):**
VARIABLE **POS (–– addr):**
METHOD **ASSIGN (pos max ––):**
METHOD **FETCH (–– max pos):**
METHOD **STORE (pos ––):**

CLASS **SLIDER–VAR (... –– ...)** ⟨***method***⟩**:** Slider actor, keeps position, step, and maximum value in own variables

PUBLIC:

Init–Parameter (o pos max step ––):
VARIABLE **STEP (–– addr):**
METHOD **ASSIGN (pos max step ––):**
METHOD **FETCH (–– max step pos):**

CLASS **SCALE–DO (... –– ...)** ⟨***method***⟩**:** Same as SCALE-VAR, but executes xt (pos ––) on changes

PUBLIC:

Init–Parameter (o n max xt ––):

VARIABLE **ACTION (−− addr):**
METHOD **STORE (−−):**

CLASS **KEY–ACTOR (... −− ...)** ⟨***method***⟩**:** This is an actor for keyboard macros. It inserts keystrokes into the called widget.

PUBLIC:

Init–Parameter (o addr u −−):
VARIABLE **STRING (−− addr):**
METHOD **FETCH (−− 0):**
METHOD **STORE (x −−):**

CLASS **TOOLTIP (... −− ...)** ⟨***method***⟩**:** A tooltip is a nested actor; it shows the widget tip some time after entering with the mouse, and forwards the other messages to actor

PUBLIC:

Init–Parameter (actor tip −−):
WIDGET POINTER **TIP (... −− ...)** ⟨***method***⟩**:**
ACTOR POINTER **FEED (... −− ...)** ⟨***method***⟩**:**
FRAME–TIP POINTER **TIP–FRAME (... −− ...)** ⟨***method***⟩**:**
EARLY METHOD **SHOW–TIP (−−):**

CLASS **EDIT–ACTION (... −− ...)** ⟨***method***⟩**:** This actor handles text input field key events, and does all the editing stuff. After each keystroke and each click, it calls xt (−−).

PUBLIC:

Init–Parameter (o xt −−):
STATIC VARIABLE **KEY–METHODS (−− addr):**
VARIABLE **STROKE (−− addr):**
EARLY METHOD **BIND–KEY (key method −−):**
EARLY METHOD **FIND–KEY (key −− addr):**
METHOD **STORE (addr u −−):**
METHOD **FETCH (−− addr u):**

2.2. Widgets

There are a lot of widgets, but fortunately, one can classify them into a few sets. Widgets have a common protocol.

CLASS **GADGET (... −− ...)** ⟨***method***⟩**:** The parent class of all widgets and display is the gadget. It defines the basic protocol, and the common data handling.

PUBLIC:

VARIABLE **X (−− addr):** x-coordinate of gadget
VARIABLE **Y (−− addr):** y-coordinate of gadget
VARIABLE **W (−− addr):** width of gadget
VARIABLE **H (−− addr):** height of gadget
STATIC VARIABLE **/STEP (−− addr):** milliseconds for a repeated action (like scrolling)
STATIC VARIABLE **FOCUSCOL (−− addr):** color index for drawing focused widget
STATIC VARIABLE **DEFOCUSCOL (−− addr):** color index for drawing defocused widget

STATIC VARIABLE **SHADOWCOL (−− addr):** color index for drawing shadows
GADGET POINTER **WIDGET (... −− ...)** ⟨***method***⟩**:** pointer to the next widget in the same hierarchy
GADGET POINTER **PARENT (... −− ...)** ⟨***method***⟩**:** pointer to the parent widget
METHOD **DPY! (dpy −−):** sets the dpy of the widget (and child widgets if any)
METHOD **FONT! (font −−):** sets the font of this widget (and child widgets if any)
METHOD **HGLUE (−− min glue):** returns minimum and extendable width for horizontal dimension
METHOD **VGLUE (−− min glue):** returns minimum and extendable width for vertical dimension
METHOD **HGLUE@ (−− min glue):** returns cached HGLUE
METHOD **VGLUE@ (−− min glue):** returns cached VGLUE
METHOD **XINC (−− off delta):**
METHOD **YINC (−− off delta):**
METHOD **XYWH (−− x y w h):** returns the current coordinates of a widget
METHOD **RESIZE (x y w h −−):** changes the position and size of a widget
METHOD **REPOS (x y −−):** changes the position of a widget
METHOD **RESIZED (−−):**
METHOD **MOVED (x y −−):**
METHOD **!RESIZED (−−):**
METHOD **ASSIGN (−−):** assigns a widget-specific value to a widget
METHOD **GET (−−):** obtains the widget-specific value
METHOD **CLICKED (x y b n −−):** called on a click event. **x y** is the mouse coordinate, **b** is the button bit vector, and **n** is the number of edges
METHOD **>RELEASED (x y b n −−):** waits for mouse button to be released
METHOD **KEYED (key state −−):** called on a keyboard event. **key** is the key, **state** is the state bit vector of the modifier keys.
METHOD **FOCUS (−−):** changes the appearing of the widget for being in focus
METHOD **DEFOCUS (−−):** changes the appearing of the widget for being out of focus
METHOD **LEAVE (−−):**
METHOD **SHOW (−−):** makes the widget visible
METHOD **HIDE (−−):** makes the widget invisible
METHOD **SHOW–YOU (−−):** makes the widget visible within a scrollable window
METHOD **DRAW (−−):** draws the widget
METHOD **CLOSE (−−):** reacts on the close event of the surrounding window, and passes it to the responsible widget
METHOD **INSIDE? (x y −− flag):** returns true when the coordinage **x y** is inside the widget
METHOD **HANDLE–KEY? (−− flag):** returns true if the widget can handle keyboard events
METHOD **NEXT–ACTIVE (−− flag):** finds the next active child. Returns false when no further widget is found
METHOD **PREV–ACTIVE (−− flag):** finds the previous active child. Returns false when no further widget is found
METHOD **FIRST–ACTIVE (−−):** finds the first active child.
METHOD **APPEND (o before −−):** adds the widget into the widget chain
METHOD **DELETE (addr addr' −−):** removes the widget out of the widget chain

Class **WIDGET (... −− ...)** ⟨***method***⟩**:** This is the base class of all widgets, it defines the protocol, and a few actions that are nice to have.

Private:

Early Method **>callback (cb −−):**

Public:

DISPLAYS Pointer **DPY (... −− ...)** ⟨***method***⟩**:**
ACTOR Pointer **CALLBACK (... −− ...)** ⟨***method***⟩**:**
Early Method **DOPRESS (dx dy −− dx dy x y):**
Early Method **WHILEPRESS (x y b n −−):**
Early Method **SHADOW (−− lc sc):**
Early Method **DRAWSHADOW (lc sc n x y w h −−):**
Early Method **TEXTSIZE (addr u n −− w h):**
Early Method **XS (−− n):**
Early Method **XN (−− n):**
Early Method **XM (−− n):**
Early Method **HM (−− n):**
Method **+PUSH (−−):**
Method **−PUSH (−−):**

Buttons und Labels

Die aktiven Komponenten sind in großer Zahl vorhanden, mit und ohne Icon, als Button, als Toggle–Button, zum Öffnen von Menüs...

```
GADGET                    28 180
  WIDGET                  32 184
    BOXCHAR               52 188
      BUTTON              56 188
        ALERTBUTTON       64 188
        MENU-ENTRY        56 188
          EDIMENU-ENTRY   60 188
          MENU-TITLE      60 196
            INFO-MENU     68 200
            SUB-MENU      60 196
        ICON-BUTTON       60 188
          BIG-ICON        60 188
        LBUTTON           56 188
          FILE-WIDGET     80 188
          LABEL           56 188
            MENU-LABEL    56 188
      TOGGLECHAR          56 196
        FLIPICON          60 196
        TOGGLEICON        64 196
        TBUTTON           60 196
          TICONBUTTON     68 196
          TOPINDEX        60 196
          TOGGLEBUTTON    64 196
          FLIPBUTTON      60 196
          RBUTTON         60 196
    ICON                  36 184
    ICON-PIXMAP           40 188
```

2.3. Boxes

Diese Widgets nehmen andere Widgets auf und formatieren sie. Die Buchstaben stehen für:

H Horizontal

V Vertikal

A ein aktives Element, Navigation mit TAB

R Radiobuttons

T Tabulatorbox — alle Objekte sind gleich groß

Daneben gibt es noch Boxen für Viewports, die die entsprechenden Slider automatisch aufbauen, sowie Boxen die mit Hilfe eines `hsizer` bzw. `vsizer` vom Benutzer in der Größe verändert werden können, und Boxen, die die Schrittweite beim Resizing der übergeordneten Boxen und Viewports auf ihre Höhe bzw. Breite setzen.

Neben den Aufteilungen in Klassen besitzen Boxen noch Attribute. So kann ihre Ausdehnung in Höhe und Breite auf das Minimum beschränkt werden, zwischen jedes Element ein Abstand eingefügt werden, ein Rand (nach ,,oben“ und ,,unten“) erzeugt werden, oder die ganze Box zeitweilig unsichtbar gemacht werden. Mit dieser Technik werden die ,,Karteireiter“ implementiert. Außerdem kann sie verwendet werden, um zur Zeit nicht verfügbare Kommandos zu verstecken.

Textfelder

Textfelder erlauben die Eingabe von Texten und Zahlen (mit Syntaxprüfung).

```
TEXTFIELD                60 204
  INFOTEXTFIELD          64 204
    INFONUMBERFIELD      64 204
  NUMBERFIELD            60 204
```

Slider und Resizer

Slider und Scaler positionieren Viewports und geben Zahlen (im kleinen Bereich) ein. Resizer ändern die Größe einer `hasbox` bzw. `vasbox`, durch Ziehen in die entsprechende Richtung.

```
SLIDER                   52 212
  VSLIDER                52 212
    VSCALER              64 216
    VSLIDER0             52 212
  HSLIDER                52 212
    HSCALER              64 216
    HSLIDER0             52 212
HRTSIZER                 52 192
  HMRTSIZER              52 192
    HSIZER               52 192
VRTSIZER                 52 192
  VMRTSIZER              52 192
    VSIZER               52 192
```

Glues

Glues sind dehnbare Objekte, die an der richtigen Stelle eingefügt ein gefälliges Layout erlauben. Plaziert man zwei gleichgroße Glues links und rechts von einem Objekt, so wird es z.B. zentriert. Ein besonderer Glue ist die `canvas`, in die gezeichnet werden kann. Sie versteht eine Art Turtle–Grafik und Koordinatentransformationen.

```
(NIL                    32 184
GLUE                    48 184
  RULE                  52 184
    CANVAS              88 224
    MFILL               52 184
    MSKIP               52 184
      SSKIP             60 184
    VRULE               52 184
    HRULE               52 184
  VGLUE                 48 184
  HGLUE                 48 184
```

Terminal und Editor

Terminal und Screen-Editor stehen ebenfalls als elementare Komponente zur Verfügung.

```
TERMINAL                80 260
  SCREDIT              116 284
```

2.4. Displays

Hier gibt es Fenster, Viewports (zeigen einen Ausschnitt), Doublebuffer (für's flimmerfreie Zeichnen), Menü–Rahmen...

```
DISPLAYS                136 344
  WINDOW                148 356
    FILE-SELECTOR       176 360
    TERMWIN             148 356
    MENU-WINDOW         148 356
    MENU-FRAME          152 356
  (NILSCREEN            136 344
  BACKING               160 348
    VIEWPORT            208 360
      SCRVIEWPORT       208 360
      HVIEWPORT         216 360
      VVIEWPORT         216 360
    DOUBLEBUFFER        160 348
```

Resources

Fonts

3. Theseus — der GUI-Editor

Wie editiert man nun so eine Oberfläche? Das Formatieren der Buttons und Textfelder wird ja vom System selbst übernommen, ist also nicht Aufgabe des Benutzers. Dieser muß lediglich die logische Anordnung festlegen.

Das Projekt wird dazu hierarchisch gegliedert. Die oberste Hierarchie bilden die Fenster. Es gibt hierbei sowohl modale als auch nicht–modale Dialoge. In diese Dialoge bettet nun

der Benutzer ein Gerüst aus horizontalen und vertikalen Boxen ein. Diese Boxen füllt er mit entsprechenden Inhalten und Glues. Theseus kann derzeit allerdings nur einen Dialog auf einmal editieren.

3.1. Theseus Komponenten

Menü

Das Menü besteht aus File-Menü, Edit-Menü und Help-Menü.

Widget Gruppen

Hier sind die verschiedenen Widgets gruppiert. Ein Klick auf einen Button fügt das entsprechende Widget in den gerade bearbeiteten Dialog ein. Das Widget wird an der aktuellen Position eingefügt (entsprechend der Einstellungen).

Bearbeitungsmodi

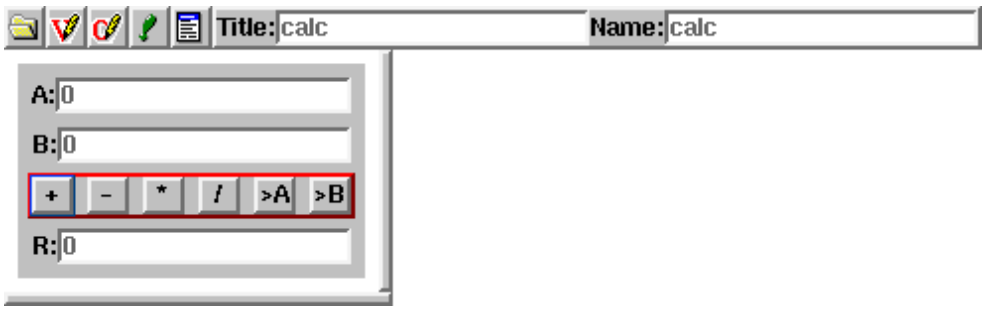
Die Bearbeitungsmodi teilen sich in drei Gruppen auf:

- Insert modi:
  - Objekt am Anfang der aktuellen Box einfügen
  - Objekt am Ende der aktuellen Box einfügen
  - Objekt vor aktuellem Objekt einfügen
  - Objekt nach aktuellem Objekt einfügen
- Navigation, wählt die aktive Box. Das aktuelle Widget kann nur mit der Maus ausgewählt werden.
  - Eine Box hoch in der Hierarchie
  - Eine Box nach links/oben
  - Eine Box nach rechts/unten

  - Zum ersten Kind

- Kurzwahl:
  - Dialog laden
  - Dialog speichern
  - Dialog ausprobieren
  - Als Modul speichern

Dialogeditor



Der Dialogeditor zeigt eine Navigationszeile für jeden bearbeiteten Dialog. Den Dialog kann man mit den Leisten rechts und unten vergrößern, um zu sehen, wie er sich dabei verhält.

Die Navigationszeile besteht aus einem Icon, das den Dialog zeigt/versteckt, einem, das das Deklarationsfeld zeigt/versteckt, einem, das das Codefeld zeigt/versteckt, einem, um zu wählen, ob der Dialog beim Start gezeigt wird, einem Dialogmen und ein Dialog-Titel und -Name. Man muß jedem Dialog einen Namen geben, denn ohne Namen kann der Dialog nicht gespeichert werden. Diese Name ist der Name der abgeleiteten Klasse; man kann auf ihn im eigenen Code zugreifen.

Das Deklarationsfeld enthält Variablen und Methodendeklarationen für diese Klasse.

Das Codefeld enthält Methodendefinitionen.

Das Dialogeditorfeld enthält den Dialog selbst.

Click modi:

Edit Ein Klick öffnet den Inspektor des Objekts, mit Fokus auf dem Text/Code/Namen (links/mitte/rechts).

Cut&Paste ⟨⇧⟩⟨mouse⟩ Klick links schneidet das Objekt auf den Stack, Klick mitte oder rechts fügt es aus dem Stack ein.

Try ⟨Ctrl⟩⟨mouse⟩ Klick läßt das Objekt reagieren wie im Dialog, nur ohne den Code auszuführen.

Box Creator

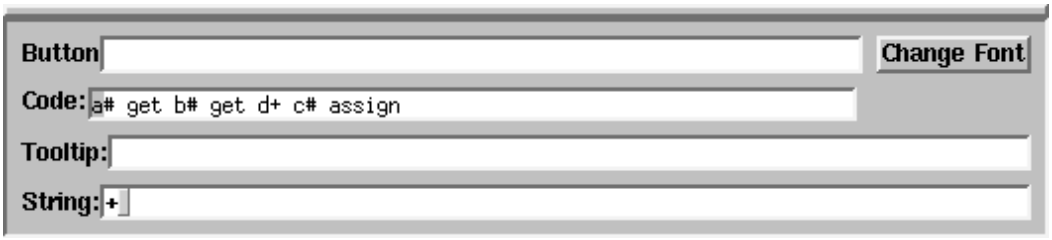
The box creator has two buttons to create horizontal and vertical boxes. Boxes are simple layout managers, that arrange containing objects one after the other. Boxes are created as normal objects, so they go to the same places where a normal object would go. They inherent the settings of the parent object, so these settings have to be changed using the box inspector.

Box inspector

The properties of the current box can be changed in the box inspector:

- The **horizontal** switch changes the direction of the box
- The **active** switch changes the selection behavior: active boxes contain one single active object, navigation with `tab` is possible.
- The **radio** switch activates deselection on click, thus only one switch inside such a box may be active at a time.
- The **tabbing** switch changes the layout: all objects except glues in tabbed boxes have the same size.
- The **hfixbox** shrinks the box to the minimal size, no growing is possible in horizontal direction
- The **vfixbox** shrinks the box to the minimal size, no growing is possible in vertical direction
- The **flipbox** hides the box when active
- The **hskip** box or slider (with **Details** activated) adds horizontal skips between objects
- The **vskip** box or slider (with **Details** activated) adds vertical skips between objects
- The **border** box or slider (with **Details** activated) adds a shadow to the box (raised or sunken).

Object inspector



The object inspector contains the informations of the current object. The fields depend on the class of the object, however, some fields are common between objects.

- The **name** field selects the name of the object, this name is used in code to refer to this object
- The **string** field is the string the object displays
- The **code** field is the code that is executed on clicks
- The **tooltip** field is the tooltip that is shown when the mouse is over the object (an empty string means no tooltip)

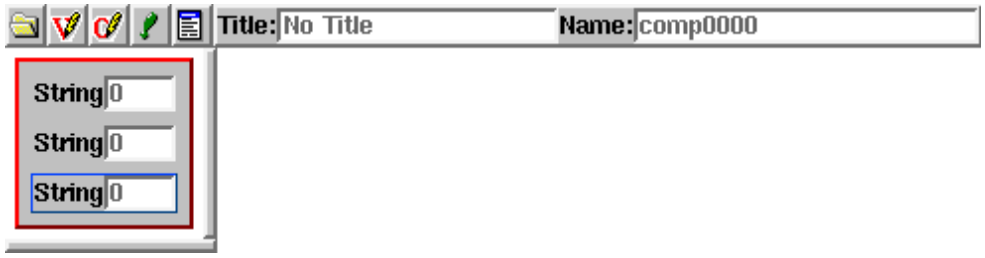
There are many other fields, for other properties of the widget.

3.2. Step by Step Example

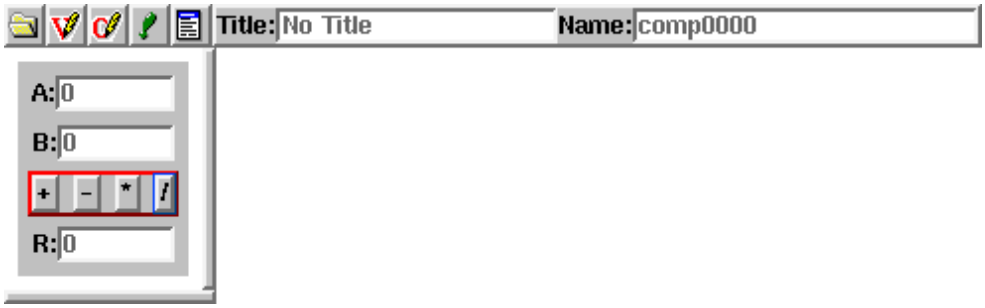
Anhand eines Beispiels soll das Vorgehen erklärt werden. Ziel ist ein kleiner Taschenrechner mit den vier Grundrechenarten, der auf Ganzzahlen operiert.

Die Eingabefelder und das Ergebnisfeld sollen untereinander, also wird eine `vbox` angelegt, und darin drei `infonumberfields`:

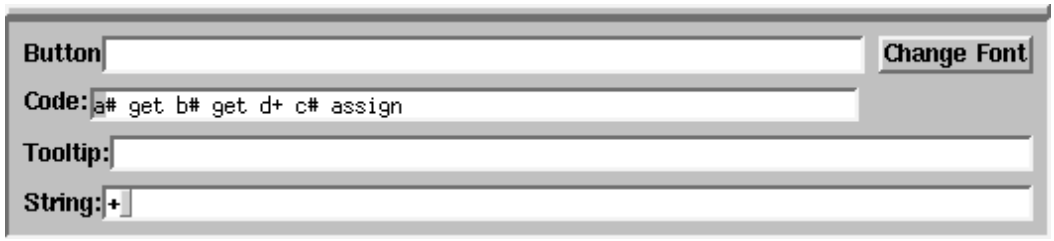
Unter die beiden Eingabefelder sollen die Buttons für die Operationen waagerecht nebeneinander stehen. Es muß also eine horizontale Box vor dem aktuellen Objekt angelegt werden, und darin vier Buttons. Ein wenig Abstand zwischen den Eingabeboxen und Buttons wäre auch nett:



Nun sollen die Objekte noch einen vernünftigen Text bekommen. Dazu klickt man jedes Objekt an (im Modus „Edit“), und tippt den Text ein:



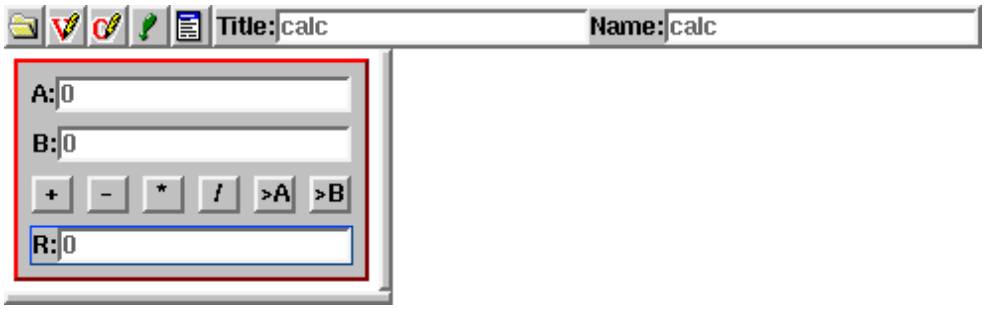
Damit man die Eingabefelder referenzieren kann, müssen sie einen Namen bekommen. Dazu wählt man den Modus ,,Name“, klickt die Felder an und tippt den Namen ein (`a#`, `b#` und `r#`). Nun kann man Code eingeben, etwa für die Operation `A+B`. Entsprechend dieses Beispiels gibt man auch den anderen Code ein:



Der einzugebende Code sieht also so aus:

```
a# get b# get d+ r# assign
a# get b# get d- r# assign
a# get b# get d* r# assign
a# get b# get drop ud/mod r# assign drop
```

Doch halt! Vielleicht sollte man noch eine Möglichkeit schaffen, das Ergebnis als neuen Eingabewert (A oder B) zu verwenden. Also werden noch zwei Knöpfe gebraucht, und damit das schön aussieht, macht man alle Buttons gleich groß:



Der zusätzliche Code sieht so aus:

```
r# get a# assign
r# get b# assign
```

Nun kann man das Ergebnis auch ausprobieren, indem man auf  drückt. Dabei wird Code generiert und von einem neu gestarteten bigFORTH compiliert und gestartet. Abbildung 10.3 zeigt das fertige Fenster.

3.3. Der automatisch generierte Code

Theseus generiert aus den Buttons ein von der Klasse `window` abgeleitetes Objekt, in das der Dialog eingepaßt wird. Alle Objekte außer den Boxen bekommen einen (ggf. automatisch generierten) Namen und einen Objektpointer, damit man sie ansprechen kann. Der für dieses Projet erzeugte Sourcecode sieht wie folgt aus:

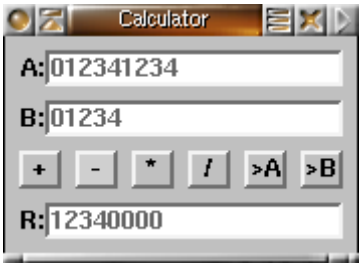


Abbildung 10.3: Der fertige Taschenrechner

```
#! xbigforth
\ automatic generated code
\ do not edit

also editor also minos also forth

component class calc
public:
  early widget
  early open
  early dialog
  early open-app
  tableinfotextfield ptr a#
  tableinfotextfield ptr b#
  tableinfotextfield ptr r#
 ( [varstart] )  ( [varend] )
how:
  : open     new DF[ 0 ]DF s" Calculator" open-component ;
  : dialog   new DF[ 0 ]DF s" Calculator" open-dialog ;
  : open-app new DF[ 0 ]DF s" Calculator" open-application ;
class;

calc implements
 ( [methodstart] )  ( [methodend] )
  : widget  ( [dumpstart] )
        ^^ SN[  ]SN ( MINOS ) &0. ]N ( MINOS ) S" A:"
           tableinfotextfield new  ^^bind a#
        ^^ SN[  ]SN ( MINOS ) &0. ]N ( MINOS ) S" B:"
           tableinfotextfield new  ^^bind b#
          ^^ S[ a# get b# get d+ r# assign ]S ( MINOS ) S" +" button new
          ^^ S[ a# get b# get d- r# assign ]S ( MINOS ) S" -" button new
          ^^ S[ a# get b# get d* r# assign ]S ( MINOS ) S" *" button new
          ^^ S[ a# get b# get drop ud/mod r# assign drop ]S ( MINOS )
             S" /" button new
          ^^ S[ r# get a# assign ]S ( MINOS ) S" >A" button new
          ^^ S[ r# get b# assign ]S ( MINOS ) S" >B" button new
        &6 hatbox new &1 hskips
        ^^ SN[  ]SN ( MINOS ) &0. ]N ( MINOS ) S" R:"
```

```
          tableinfotextfield new  ^^bind r#
      &4 vabox new panel
    ( [dumpend] ) ;
  : init  ^>^^  assign  widget 1 super init ;
class;

: main
  calc open-app
  $1 0 ?DO  stop  LOOP bye ;
script? [IF]  main  [THEN]
previous previous previous
```

11 Support Classes

NINOS uses a set of support classes to interface with other parts of the system. The multimedia classes provide a way to play sound and video, the database classes interface with SQL databases, and the 3D turtle graphics class provides an abstraction level for OpenGL support.

1. "Dragon Graphics" Forth, OpenGL und 3D-Turtle-Graphics

1.1. Einleitung

Auf der letzten Forth-Tagung habe ich eine direkte OpenGL-Anbindung in Forth vorgestellt. OpenGL ist eine 3D-Grafik-Library, die einem viel Arbeit abnimmt. Allerdings ist OpenGL relativ low-level, und bietet „nur" Koordinatentransformationen sowie das Zeichen von Strips, also Aneinanderreihungen von Dreiecken oder Rechtecken an. Zudem benötigt OpenGL Normalenvektoren und Texturkoordinaten, die man aber automatisch berechnen kann.

Mein Vorhaben war daher, OpenGL in eine einfacher zu benutzende Library zu kapseln, eine Art 3D-Turtle-Grafik. Um den Jahreswechsel gab es eine Diskussion in comp.lang.forth über eine solche 3D-Turtle-Grafik. Dave Taliaferro stellte eine in pForth geschriebene 3D-Turtle-Grafik vor. Marcel Hendrix implementierte kurz darauf etwas vergleichbares in iForth.

Beide Turtles können sich durch den Raum bewegen und hinterlassen dabei eine Spur aus OpenGL-Objekten, etwa Zylindern oder Kugeln. Man kann damit also keine komplexen Körper erzeugen.

Das hier vorgestellte System setzt auf dem Turtle-Prinzip auf, erlaubt es aber, Körper zu beschreiben. Da es diese nicht als Komposition aus starren Einzelteilen aufbaut, ist eine echte Skelettanimation möglich, etwas, was auch bei Hollywood-Tools noch mit viel Aufwand verbunden ist. Nicht zufällig haben die abendfüllenden Streifen Insekten, also Außenskelette, als Darsteller. Animationen mit Innenskeletten beschränken sich auf kurze Sequenzen. 3000 Punkte (der Drache) kann man auch nicht einfach von Hand eingeben.

1.2. Das Prinzip

Eine normale 2D-Turtle-Grafik kann vorwärts und rückwärts fahren, sowie sich nach rechts und links drehen. Dabei hinterläßt sie Spuren, also Striche. Das Prinzip läßt sich auch auf Flächen erweitern, indem man die von der Turtle gezeichneten Polygone auffüllt.

Im Raum ist die Turtle in ihrem richtigen Element (unter Wasser). Statt schwerfällig herumzukriechen, kann sie auch nach oben und unten schwimmen, sowie um ihre Achse rollen. Man muß sich nun überlegen, wie die „Spur" aussehen soll, und wie man von Strichen auf Flächen und noch wichtiger Körper kommt.

Statt einfach vorgefertigte Objekte fallenzulassen, erlaubt es diese 3D-Turtle-Grafik, Schnitte durch den Körper zu beschreiben. Diese Schnittebenen werden dann miteinander verbunden, um einen Körper zu formen. Um etwa einen Zylinder darzustellen, verbindet man zwei Kreise miteinander. Kreise werden durch Vielecke angenähert.

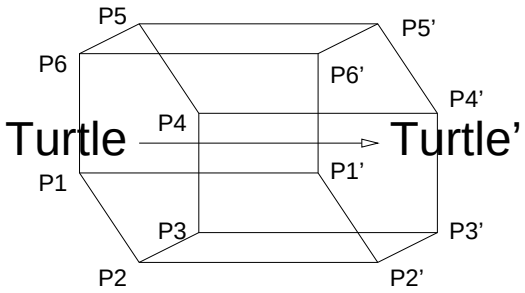


Abbildung 11.1: 3D-Turtle-Prinzip

Die 3D-Turtle-Grafik stellt für diese Schnitte keine 2D-Turtle-Grafik zur Verfügung (obwohl das ja irgendwie naheliegend wäre), sondern verschiedene Koordinatensysteme, etwa Zylinder-Koordinaten. Man kann natürlich auch die 3D-Turtle verwenden, Umrisse abzufahren. Der Ursprung ist durch die Turtle bestimmt, die Ausrichtung des Koordinatensystem entspricht der Blickrichtung der Turtle.

1.3. Ein einfaches Beispiel

Als einfaches Beispiel soll uns ein Baum dienen. Ein Baum besteht aus einem Stamm und Zweigen, die wir hier durch mit Sechsecken angenäherten Zylindern darstellen. Als Blatt soll eine einfache Kugel-Näherung dienen. Unser Baum hat ein paar Parameter: die

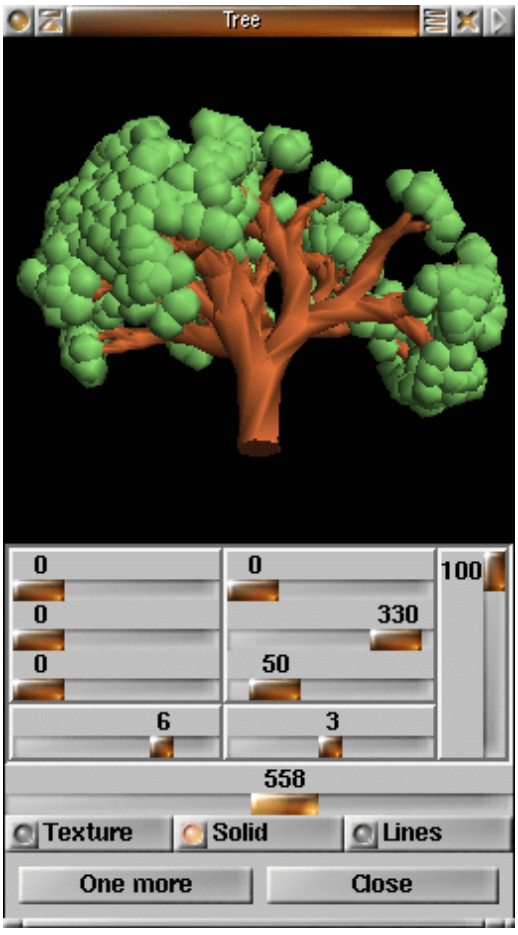


Abbildung 11.2: Baum

Verzweigungstiefe, und die Anzahl der Äste. Der oben dargestellte Baum hat auch noch

eine Wahrscheinlichkeit, mit der Äste ausfallen, die wollen wir hier aber nicht implementieren.

Fangen wir also mit dem Stumpf an. Zunächst brauchen wir eine untere Begrenzungsfläche, hier erst einmal ein Sechseck. Wir lassen die Turtle, wo sie gerade steht, und öffnen einen Pfad mit sechs Punkten pro Runde.

```
: baum ( m n -- )
  .brown .color  6 open-path
```

Die Sechsecke haben einen Winkel von $\pi/3$ pro Schritt, den können wir uns schon mal vormerken. Er bestimmt die Schrittweite für die Funktionen, die keinen Winkel als Parameter haben.

```
  pi 3 fm/ set-dphi
```

Nun fangen wir mit sechs Punkten in der Mitte an. Wir müssen zunächst die sechs Punkte hinzufügen (der Pfad ist am Anfang leer), und dann in der nächsten Runde nochmal setzen, um die Normalenvektoren richtig zu setzen (aller Anfang ist schwer — da die Normalenvektoren sich auf die letzte Runde beziehen, gibt es in der ersten Runde noch gar keine).

```
  6 0 DO  add  LOOP next-round
  6 0 DO  set  LOOP next-round
```

Um sie herum werden in der nächsten Runde die Dreiecke gezeichnet, die das Boden-Sechseck ergeben. Die Größe der Dreiecke ist hier aus der Verzweigungstiefe errechnet, indem die mit 0.03 multipliziert wird. Da OpenGL selbst mit Fließkommazahlen arbeitet, verwendet auch die Turtle-Grafik solche Zahlen.

```
  6 0 DO dup !.03 fm* set-r LOOP next-round
```

Nun kommt ein kleiner Trick, um eine scharfe Kante zu setzen - die 3D-Turtle-Grafik berechnet nämlich die Normalenvektoren an einem Punkt aus der Summe der Kreuzprodukte der Vektoren nach links/hinten und nach rechts/vorne. Ein weiterer Schnitt an derselben Stelle bewirkt, daß nur eine Richtung für den Normalenvektor berücksichtigt wird.

```
  6 0 DO  dup !.03 fm* set-r  LOOP
```

Nun können wir zum eigentlichen rekursiven Teil kommen, den Zweigen:

```
  zweige ;
: zweige ( m n -- )  recursive
```

Um eine doppelte Rekursion zu vermeiden, verwende ich eine Schleife für die Endrekursion.

```
  BEGIN  dup  WHILE
```

Auch hier müssen wir erstmal eine neue Runde anfangen. Damit der Baum nicht so plattgepreßt in der Ebene steht, drehen wir ihn pro Verzweigung um 54 Grad.

```
        next-round  pi !.3 f* roll-left
```

Als nächstes müssen wir entsprechend der Verzweigungstiefe vorwärts gehen, und einen neuen Ring zeichnen.

```
        dup !.1 fm* forward
        6 0 DO dup !.03 fm* set-r LOOP
```

Für die weiteren Verzweigungen brauchen wir eine Schleife — bis auf die letzte Verzweigung, die wird ja von der Endrekursion abgearbeitet.

```
        over 1 ?DO
```

Jeder Ast wird durch Rotieren um die Blickachse gedreht — `I'` ist hier das Ende der Schleife. Der Befehl `>turtle` sichert den aktuellen Status der Turtle auf einem Turtle-Stack, `turtle>` nimmt ihn wieder herunter. Ich verwende eine lokale Variable, da die Turtle etwas Returnstackplatz braucht, und damit `I` und `I'` nicht verfügbar sind. Den

Fließkommastack darf man auch nur für Zwischenberechnungen verwenden, da die C-Library von einem leeren Stack ausgeht.

Nach der Drehung müssen wir nach rechts (um 18 Grad hier), und danach die Turtle wieder zurückdrehen - damit die Punkte der jeweiligen Schnitte zusammenpassen. Die geänderte Blickrichtung der Turtle bleibt durch diese Operation erhalten, nur ihre Ausrichtung im Raum wird zurückgesetzt.

```
            2pi I I' fm*/ { f: di |
            >turtle
               di roll-left pi 5 fm/ right
               di roll-right
               2dup 1- zweige
            turtle> }
```

So, nun noch die Schleife fertigmachen

```
            LOOP
```

und für die Endrekursion nach rechts kippen (diesmal ist die Drehung 0 Grad).

```
          pi 5 fm/ right
  1- REPEAT
```

Am Schluß noch den Pfad zumachen, und ein Blatt zeichnen.

```
  close-path leaf 2drop ;
```

Das Blatt selbst ist eine einfache angenäherte Kugel:

```
  : leaf ( -- )
    .green .color
    6 open-path  6 0 DO  add  LOOP
    next-round !.1 forward
        6 0 DO  !.2 set-r  LOOP
    next-round !.2 forward
        6 0 DO  !.2 set-r  LOOP
    next-round !.1 forward
        6 0 DO  !.1 set-r  LOOP
    next-round
        6 0 DO  !0  set-r  LOOP
    close-path .brown .color ;
```

Das sind noch nicht die ganzen Sourcen, wir brauchen noch etwas Overhead, um die Ansicht auf den Baum zu verändern. Die ganzen Sourcen finden sich in der Datei `tree.str` (3D-Grafik) und `tree.m` (Benutzeroberfläche).

1.4. Ein komplexeres Beispiel: Der Drache

Da der Drache sehr komplex ist, beschreibe ich hier nur die wesentlichen Punkte. In typischer Forth-Tradition wird der Drache vom Schwanz her aufgezäumt.

Schwanz

Der Drache besteht aus einzelnen Segmenten, die im wesentlichen einen Kreis mit einem Zacken darstellen:

```
: dragon-segment ( ri ro n -- )
  { f: ri f: ro | next-round
    ro set-r  1 DO  ri set-r  LOOP
    ro !-0.0001 set-rp !0 phi df! } ;
```

Damit der Schwanz so schön wackelt, und auch um die anderen Bewegungen zu synchronisieren, gibt es einen Timer, der in einen Winkel $[0, 2\pi[$ umgesetzt wird.

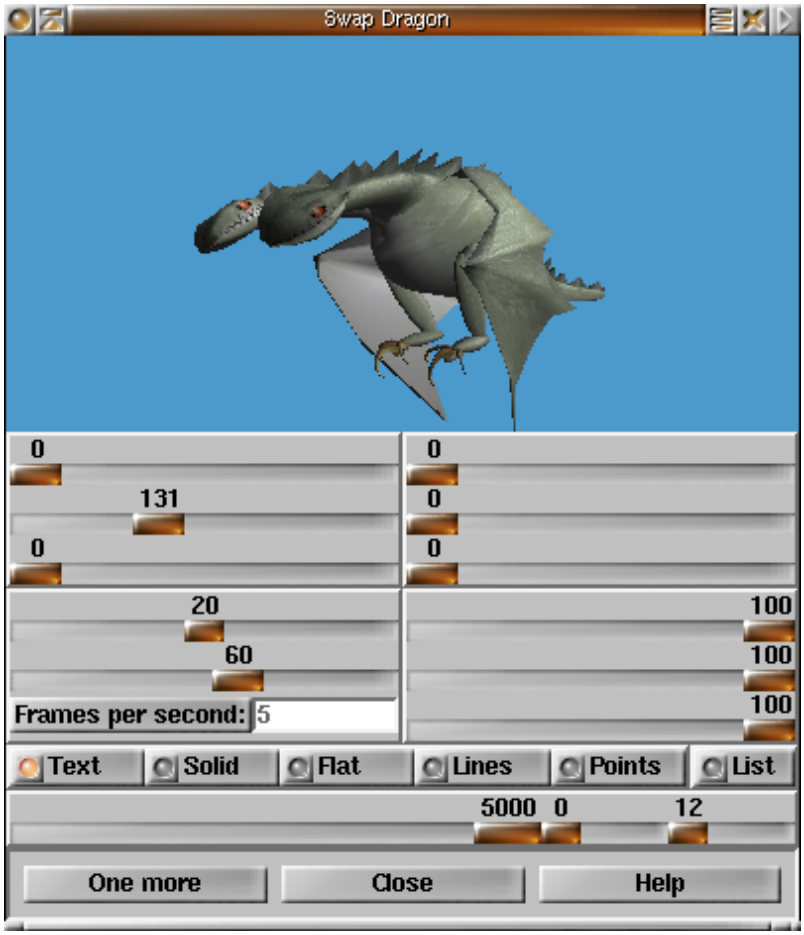


Abbildung 11.3: Swap-Drache

```
Variable tail-time
: time' ( -- 0..2pi )
  tail-time @ &24 &60 &30 * * um* drop
  0 d>f !$2'-8 pi f* f* ;
```

Das eigentliche Schwanzwackeln wird dann aus der Segmentnummer und der Zeit berechnet — das Ergebnis ist die Verschiebung nach links bzw. rechts.

```
: tail-wag ( n -- f )
  >r pi r@ 1 + fm* !.2 f* time' f+
  fsin r> 2+ dup * 1+ fm/ !30 f* ;
```

Der Ursprung des Drachens liegt im Bauch, nicht an der Schwanzspitze. Gezeichnet wird der Drache aber von der Schwanzspitze her — also muß zunächst eine Kompensation berechnet werden, sonst wackelt der Schwanz mit dem Drachen.[1]

```
: tail-compensate ( n -- f )  !0
  0 DO  I 2+ tail-wag f+ !1.1 f/  LOOP
  !1.1 !20 f** f* fnegate ;
```

Der eigentliche Schwanz ist damit recht einfach: erstmal zur Schwanzspitze zurück, und einen Punkt als Anfangspolygon setzen. Dann Schritt für Schritt den Schwanz wackeln lassen, ein Stück vorwärts gehen, und ein Drachensegment zeichnen. Jedes zweite Drachensegment hat einen Zacken nach oben, und die Skalierung macht den Schwanz auch immer dicker. Der Radius wird zusätzlich vergrößert. Diese Skalierung muß natürlich zuerst in die andere Richtung vorgenommen werden. Als Texture-Mapping-Funktion wird

[1] Sowas soll in der Politik schon mal vorkommen.

z, ϕ verwendet, also Bewegung der Turtle für die eine Texturkoordinate, und der Winkel gegen die Senkrechte für die andere.

```
: dragon-tail ( ri r+ h n -- ri h )
  zphi-texture
  { f: ri f: r+ f: h n |
  !1.05 !-20 f**
  !1.1  !-20 f** !1 scale-xyz
  h -&15 fm* &20 tail-compensate
  h -&25 fm* forward-xyz
  n 1+ 0 DO  add  LOOP
  20 0 DO !0  i 2+ tail-wag  h forward-xyz
      pi &90 fm/ up
      ri fdup I 1 and 0= IF  r+ f+ THEN
      n dragon-segment
      !1.05 !1.1 !1 scale-xyz
      !.025 ri f+ to ri
  LOOP  ri r+ h } ;
```

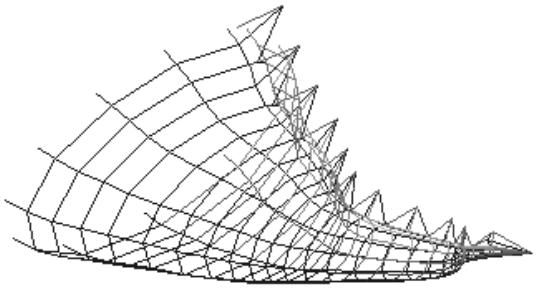


Abbildung 11.4: Schwanz

Körper

Der Körper des Drachens besteht aus genau denselben Segmenten wie der Schwanz, nur statt weiter zu wachsen, muß der Körper sich wieder schließen.

```
: dragon-wamp ( ri r+ h ri+ n -- ri' )
  { f: ri f: r+ f: h f: ri+ n |
  8 0 DO  h forward
      ri fdup I 1 and 0= IF  r+ f+ THEN
      n dragon-segment
      ri+ ri f+ to ri !-0.02 ri+ f+ to ri+
  LOOP  ri ri+ !.02 f+ f- } ;
```

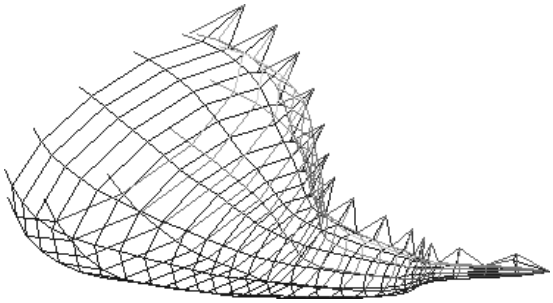


Abbildung 11.5: Körper

Hals

Auch der Hals besteht aus diesen Segmenten; allerdings gibt es hier zwei verschiedene Wachstumsfunktionen, eine für die Schulter (schnelle Größenabnahme), und eine für den eigentlichen Hals (langsame Abnahme). Die Schulter biegt sich nach links, der Hals wieder nach rechts. Entsprechend wird die Funktion `dragon-neck-part` zweimal aufgerufen.

```
: dragon-neck-part
  ( ri r+ h factor angle n m -- ri' )
  swap { f: ri f: r+ f: h f: factor f: angle n |
  0 ?DO  h forward  angle left
      pi &30 fm/
      time' fsin !.01 f* f+ down
      factor ri f* to ri
      ri fdup I 1 and 0= IF  r+ f+ THEN
      n dragon-segment
  LOOP ri } ;
: dragon-neck ( ri r+ h angle n -- )
  { f: r+ f: h f: angle n |
  r+ h !.82 angle
    n 4 dragon-neck-part
  r+ h !.92 angle f2/ fnegate
    n 6 dragon-neck-part
  fdrop close-path } ;
```

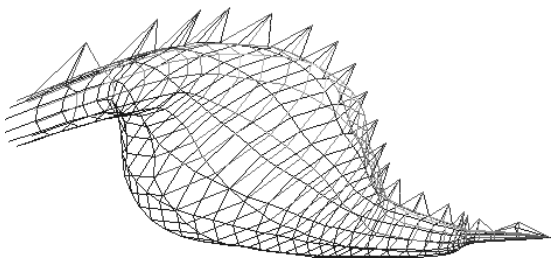


Abbildung 11.6: Hals

Kopf

Der Kopf besteht aus einem abgerundeten Rechteck mit einer Einritzung für die Zähne. Die Funktion ist nicht so einfach zu generieren, deshalb verwende ich ein Array für die Koordinaten, allerdings nur für die linke Hälfte des Kopfs; die rechte wird durch Spiegelung an der Y-Achse gewonnen. Die Größenverhältnisse der Schnitte zueinander entsprechen in etwa dem Bauch. Der Kopf hat eine andere Textur, eine mit Augen, Nasenlöchern und Zähnen.

```
Create head-xy
       !0.28 f>fs , !0.0 f>fs ,
       !0.30 f>fs , !0.5 f>fs ,
       !0.25 f>fs , !0.6 f>fs ,
       !0.05 f>fs , !0.6 f>fs ,
       !0.00 f>fs , !0.5 f>fs ,
       !-.05 f>fs , !0.6 f>fs ,
       !-.10 f>fs , !0.6 f>fs ,
       !-.15 f>fs , !0.5 f>fs ,
: dragon-head ( t1 shade -- )  !text
```

```
pi 6 fm/ down  !1.2 !.4 !.4 scale-xyz
!-.65 forward
!.5 x-text df!
16 open-path  16 0 DO  add  LOOP
6 0 DO
    I 5 = IF    !.25
          ELSE  I 0= IF !0 ELSE !.35 THEN
          THEN forward
    >matrix
    pi !0.1 f* I 2* 5 - fm* fcos
    fdup !.5 f+ !1 scale-xyz
    next-round
    head-xy 16 cells bounds DO
        I sf@ I cell+ sf@ set-xy
        2 cells +LOOP
    head-xy dup 14 cells + DO
        I sf@ I cell+ sf@
        !1'-6 f+ fnegate set-xy
        -2 cells +LOOP
    matrix>
LOOP
!1 x-text df!
close-path ;
```

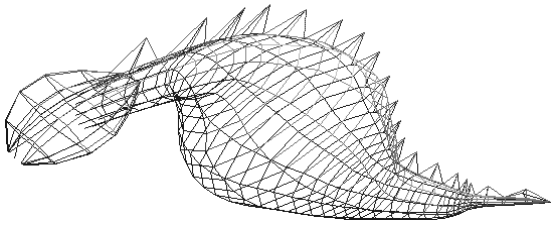


Abbildung 11.7: Kopf

Der zweite Hals und Kopf werden mit entsprechend negierten Winkeln gezeichnet. Ähnlich dem vorherigen Beispiel wird dazu der Status der Turtle gesichert, und vom selben Status erneut ausgegangen.

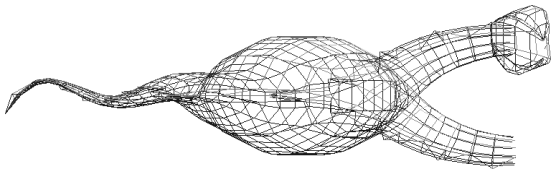


Abbildung 11.8: Zweiter Hals

Flügel

Der Flügel hat ein einfaches, flachgestrecktes Sechseck als Schnitt. Dieses Sechseck sorgt für die Krümmung des Flügels, und wird zur Modellierung der „Finger“ verwendet.

```
: wing-step { f: f2 f: f3 |
  next-round
  !0 f2 fnegate                    set-xy
```

```
  f3 f2/ f2 fnegate                set-xy
  f3 f3 !.125 f*                   set-xy
  f3 !.001 f- f3 !.125 f* !.001 f+ set-xy
  f3 f2/ f2                        set-xy
  !0.001 f2 fmin f2                set-xy } ;
```

Die Falte-Funktion des Flügels sorgt für eine Bewegung von Arm/Unterarm und der Finger abhängig von der Zeit für eine Auf/Abwärtsbewegung des Flügels. f_2 ist ein additioneller Term zum Cosinus, f_1 ein multiplikativer.

```
: wing-fold ( f1 f2 -- )
  time pi 5 fm/ f- fcos f+ f* down ;
```

Die Bewegung und der Aufbau des Flügels sind kompliziert; deshalb erkläre ich nicht alle Einzelheiten. Auch hier wird zunächst ein Pfad geöffnet. Danach werden schrittweise Flügelansatz, Ober- und Unterarm, und zuletzt die drei Finger gezeichnet.

```
: wing ( -- )
  8 open-path !.9 scale
  6 0 DO  add  LOOP
  !.02 !1.2 wing-step !.3 forward                              Ansatz
  pi &10 fm/ down  pi &8 fm/ roll-left
  time' fsin !1.3 f* !.2 f+ right
  !.02 !1 wing-step                                            Oberarm
  pi 5 fm/ up  pi &10 fm/ right  !1 forward
  pi 5 fm/ down  pi &20 fm/ left
  time' fcos !-.25 f* !.5 f- roll-left
  time' fcos pi 6 fm/ f* down
  !.02 !1 wing-step                                            Unterarm
  time' !1 f- fcos !1 f+ pi 8 fm/ f* right
  pi -3 fm/ !-1.0 wing-fold
  pi &10 fm/ left !1 forward
  pi 4 fm/ !-1.5 wing-fold
  !.02 !2 wing-step
  2 0 DO  !.025 forward
          pi &12 fm/ !1.2 wing-fold
          pi &10 fm/ right !.05 forward
          !.02 !2 wing-step                                    Finger
  LOOP
  !0 !2 wing-step                                              Abschluß
  close-path ;
```

Der eigentliche Flügel wird für rechts und links grundsätzlich gleich gezeichnet. Die Symmetrie wird durch eine Spiegelung an der Y-Achse erreicht. Hier muß noch ein Wort zu OpenGL gesagt werden: Nur die Vorderseiten der Dreiecke werden tatsächlich gezeichnet. Durch so eine Spiegelung werden aber aus allen Vorderseiten „Rückseiten“, weil sich die Umlaufrichtung ändert. Also muß man das OpenGL mitteilen, und das macht `flip-clock`.

```
: right-wing ( h -- )
  pi/4 roll-right pi/2 right
  !2 f*  forward pi !.3 f* roll-left
  zp-texture !.13 y-text df! wing ;
: left-wing ( h -- ) !1 !-1 !1 scale-xyz
  flip-clock right-wing flip-clock ;
```

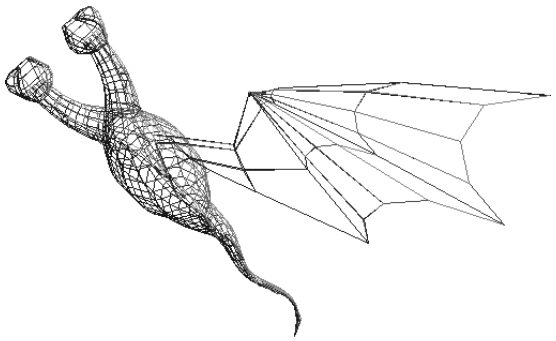


Abbildung 11.9: Flügel

Der ganze Drache

Die Beine lasse ich hier mal weg, sie sind nicht so interessant, da sie aus statischen Teilen bestehen (im Wesentlichen langgestreckte Ellipsoide und bogenförmige Klauen). Kommen wir zum Hauptprogramm:

Zunächst wackelt der Drache bei jedem Flügelschlag etwas auf und ab. Dann muß für die Drachensegmente noch der Winkel gesetzt werden.

```
: dragon-body
  ( t0 s t3 s t1 s t3 s t2 s n -- ) >r
  time' fsin !.1 f* !0 !0 forward-xyz
  pi f2* r@ fm/ set-dphi
  r@ 1+ open-path
```

Zuerst wird wie gesagt der Schwanz gezeichnet.

```
  !.1 !.3 !.2 r@ dragon-tail
```

Die Rückgabeparameter des Schwanz werden auch für den Bauch weiterverwendet.

```
  r> { f: ri f: r+ f: h n |
  ri r+ h !.06 n dragon-wamp fdrop
```

Hals und Kopf werden jeweils für rechts und links aufsetzend von derselben Stelle gezeichnet; jeweils mit negierten Winkelparameter

```
  >turtle
     ri r+ h  !10 grad>rad n dragon-neck
     2dup dragon-head  2swap !text
  turtle>  >matrix
     ri r+ h !-10 grad>rad n dragon-neck
     dragon-head  2drop
  matrix>
```

Danach muß die Textur geändert werden, und die beiden Flügel werden gezeichnet.

```
  2dup !text
  h !2 f* forward
  >turtle  h right-wing   turtle>
  >turtle  h left-wing   turtle>
```

Analog werden auch die Füße gezeichnet.

```
  h !-6 f* forward
  >turtle  right-leg   turtle>
  >turtle  left-leg    turtle>
  2drop 2drop } ;
```

1.5. Ausblick

Was kann man damit machen, was fehlt noch? Als seriöse Anwendung kann natürlich die Darstellung von dreidimensionalen Daten gelten. „Unseriösere“ Anwendungen wären etwa Computerspiele. Dafür braucht man dann Kollisionserkennung, und wahrscheinlich ein hierarchisches Modell, um Räume und bewegte/bewegbare Objekte einzuordnen. Auch unterschiedliche Levels of Detail abhängig von der Größe des Objekts am Bildschirm müssen jetzt noch mühsam von Hand programmiert werden. Ob es hier für animierte Objekte überhaupt eine andere Möglichkeit gibt, ist mir nicht klar.

Die Behandlung von verschiedenen Texturen ist im Moment noch zu aufwendig; sie müssen auf dem Stack herumgeschleppt werden. Hier muß auch das 3D-Turtle-Objekt selbst noch mehr Tools zur Verfügung stellen.

Und wie immer macht Windows Schwierigkeiten. Obwohl man nicht behaupten kann, daß die MESA-Bibliothek unter Linux fehlerfrei ist, so implementiert sie zumindest alle Features von OpenGL 1.2. Die Windows 95-OpenGL-Library von Microsoft läßt Texturen gleich ganz weg, und funktioniert auch sonst recht wenig zuverlässig. Der Drache wird jedenfalls schwarz auf schwarz dargestellt. Da Silicon Graphics ihre GLX-Sourcen freigegeben haben, sind die verbleibenden Linux-Probleme und die fehlende Hardwareunterstützung (nur 3Dfx wird unterstützt) wahrscheinlich in Kürze ausgeräumt.

1.6. Anhang: Befehle der 3D-Turtle-Grafik

Class **3D–TURTLE (... –– ...)** ⟨***method***⟩**:** The 3D turtle.

Navigation

Early Method **left (f ––):** turns the turtle's head left
Early Method **right (f ––):** turns the turtle's head right
Early Method **up (f ––):** turns the turtle's head up
Early Method **down (f ––):** turns the turtle's head down
Early Method **roll–left (f ––):** rolls the turtle's head left
Early Method **roll–right (f ––):** rolls the turtle's head right
Early Method **x–left (f ––):** rotate the turtle left around the x axis
Early Method **x–right (f ––):** rotate the turtle right around the x axis
Early Method **y–left (f ––):** rotate the turtle left around the y axis
Early Method **y–right (f ––):** rotate the turtle right around the y axis
Early Method **z–left (f ––):** rotate the turtle left around the z axis
Early Method **z–right (f ––):** rotate the turtle right around the z axis
Early Method **forward (f ––):** move the turtle in z direction
Early Method **forward–xyz (fx fy fz ––):** move the turtle
Early Method **degrees (f ––):** steps per circle. Common cases: 2π for radians (default), 360 for deg, 64 for asian degrees, or whatever you find suits your application best.
Early Method **scale (f ––):** scales the turtle's step width by the factor f
Early Method **scale–xyz (fx fy fz ––):** scale the turtle's step width in x, y, and z direction
Early Method **flip–clock (––):** change default coordinate from left hand to right or the other way round. Use that after `scale-xyz` with an odd number of negative scale factors.

Turtle state

EARLY METHOD **>matrix (−−):** push turtle matrix on the matrix stack
EARLY METHOD **matrix> (−−):** pop turtle matrix from the matrix stack
EARLY METHOD **matrix@ (−−):** copy turtle matrix from the stack
EARLY METHOD **1matrix (−−):** initialize turtle state with the identity matrix
EARLY METHOD **matrix∗ (−−):** multiply current transformation matrix with the one on the top of the matrix stack (and pop that one)
EARLY METHOD **clone (−− o):** create a clone of the turtle
EARLY METHOD **>turtle (−−):** clone the turtle and use it as current object
EARLY METHOD **turtle> (−−):** destroy current turtle and pop previos incarnation

Pathes

EARLY METHOD **open–path (n −−):** opens a path with n points in the first round
EARLY METHOD **close–path (−−):** closes a path and performs the final rendering action
EARLY METHOD **next–round (−−):** closes a round and opens the next one
EARLY METHOD **open–round (n −−):** opens a round with n points (obsolete)
EARLY METHOD **close–round (−−):** closes a round (by copying the first point as last point) and performs the per-round rendering action (obsolete)
EARLY METHOD **finish–round (−−):** performs the per-round rendering action without closing the round first (this is for open objects) (obsolete)
EARLY METHOD **add–xyz (fx fy fz −−):** adds the point at the x, y, z-coordinates relative to the turtle. x is up from the turtle, y right, z before. The point is connected to the same point of the previous round as the point before.
EARLY METHOD **set–xyz (fx fy fz −−):** sets a point with x, y, z-coordinates. The point is connected to the next point of the previous round as the point before.
EARLY METHOD **drop–point (−−):** skips one point, `set-xyz` is equal to `add-xyz drop-point`
EARLY METHOD **set–rpz (fr fphi fz −−):** set with cylinder coordinates
EARLY METHOD **set–xy (fx fy −−):** set-xyz with $z = 0$
EARLY METHOD **set–rp (fr fphi −−):** set with cylinder coordinates, $z = 0$
EARLY METHOD **set–r (fr −−):** set with cylinder coordinates, $z = 0$, $\phi = \phi_{cur}$, $\phi_{cur} = \phi_{cur} + \Delta\phi$
EARLY METHOD **set (−−):** set at current turtle location
EARLY METHOD **add–rpz (fr fphi fz −−):** add with cylinder coordinates
EARLY METHOD **add–xy (fx fy −−):** add-xyz with $z = 0$
EARLY METHOD **add–rp (fr fphi −−):** add with cylinder coordinates, $z = 0$
EARLY METHOD **add–r (fr −−):** add with cylinder coordinates, $z = 0$, $\phi = \phi_{cur}$, $\phi_{cur} = \phi_{cur} + \Delta\phi$
EARLY METHOD **add (−−):** add at current turtle location
EARLY METHOD **set–dphi (fdphi −−):** sets $\Delta\phi$

Drawing Modes

EARLY METHOD **points (−−):** draw only vertex points
EARLY METHOD **lines (−−):** draw a wire frame
EARLY METHOD **triangles (−−):** draw solid triangles
EARLY METHOD **textured (−−):** draw textured triangles

Early Method **smooth (−−):** variable: set on for smooth normals when rendering textured, set off for non-smooth rendering
Early Method **xy–texture (−−):** texture mapping based on x and y coordinates
Early Method **zphi–texture (−−):** texture mapping based on z and ϕ coordinates
Early Method **rphi–texture (−−):** texture mapping based on r and ϕ coordinates
Early Method **zp–texture (−−):** texture mapping based on z and the point number coordinates
Early Method **load–texture (addr u −− t):** loads a ppm file with the name *addr* u and returns the texture index t
Early Method **set–light (par1..4 par n −−):** Set light source n

2. SQL Interface

The SQL interface allows to interface with a database using the structured query language SQL. It now has only an interface to PostgreSQL, because noone wrote one to other databases.

The database interface has a simple foundation. You can send an SQL query string and get the result back as a table.

Class **DATABASE″ (... −− ...)** ⟨***method***⟩**:**The database class
Init–Parameter (addr u −−): opens the database specified by name
Method **exec (addr u −−):** the query string
Method **fields (−− n):** the number of fields per result
Method **tuples (−− n):** the number of tuples (results)
Method **field@ (i −− addr u):** obtains the field name
Method **tuple@ (i j −− addr u):** obtains a tuple entry
Method **clear (−−):** clears the result buffer

There are also some output functions to display the result of a query or to create a table containing the entries.

Method **.heads (−−):** displays the field names
Method **.entry (i −−):** displays an entry line
Method **.entries (−−):** displays all results including names
Method **entry–box (−− o):** creates a MINOS object with the query results

A set of methods facilitates the creations of new tables.

Method **create((addr u −−):** starts creation of a named table
Method **:string (addr u n −−):** a `varchar` array with n chars max
Method **:int (addr u −−):** an integer
Method **:float (addr u −−):** a floating point number
Method **:date (addr u −−):** a date
Method **:time (addr u −−):** a time
Method **inherits (addr u −−):** inherit mechanism of PostgreSQL, must be last (there may be multiple inherits statements)
Method **) (−−):** ends the creation o a table
Method **drop (addr u −−):** drops a table

There are also ways to construct a query string.

Method **select (addr u −−):** starts a select query, the argument is the selection; there can be several selections per query

METHOD **select–distinct (addr u −−):** starts a select distinct query
METHOD **select–as (addr1 u1 addr2 u2 −−):** argument 1 is the selection, argument 2 is the name it is assigned to
METHOD **from (addr u −−):** specifies the table(s) to select from
METHOD **where (addr u −−):** specify where clauses (several combined with AND)
METHOD **group (addr u −−):** grouped by argument
METHOD **order (addr u −−):** specifies the ordering argument
METHOD **order–using (addr u −−):** specifies order and ordering operation

12 American National Standard FORTH

1. Historie

Nachdem 1983 der Forth–83–Standard verabschiedet wurde, bemühte man sich um einen offizielleren Standard. Seit August 1987 arbeitet ein Team, das X3J14 Technical Comittee, an einem Standard für das American National Standard Institute, kurz ANSI. Dieser Standard ist inzwischen (am 24. März 1994) verabschiedet worden, also offiziell gültig.

bigFORTH erfüllt diesen Standard. Alle dort definierten Wörter sind in bigFORTH vorhanden oder können nachgeladen werden. Aus Gründen der Kompatibilität zu älteren Versionen von bigFORTH verhalten sich einige Wörter normalerweise nicht so, wie im Standard beschrieben, erst durch das Laden der Datei ANS.STR wird ihr Verhalten angepaßt.

2. Wortgruppen

Der ANSI–Standard gliedert Wörter in sogenannte “wordsets” (Wortgruppen). Diese Gruppen erlauben es dem Entwickler, die Systemgröße dem Bedarf anzupassen. Jede Wortgruppe besteht aus zwei Teilen, wovon der mit “–EXT” gekennzeichnete Extensions beinhaltet, die einzeln hinzugefügt werden können, also nicht als Gesamtheit vorhanden sein müssen.

bigFORTH implementiert dennoch alle Wortgruppen des Standards, sowie alle Extensions, denn nur ein voll erfüllter Standard stellt sicher, daß alle Standard–konformen Programme auch laufen. Im folgenden werden alle Wortgruppen des Standards aufgelistet, sowie alle Wörter, deren Verhalten sich von bigFORTH 1.10 unterscheidet.

2.1. Die CORE–Wortgruppe

Nur eine Wortgruppe ist verpflichtend vorgeschrieben: Die CORE–Wortgruppe. Alle anderen Wortgruppen sind optional, eine fehlende Implementierung beeinträchtigt nicht die Konformität zum Standard. Dennoch sind in bigFORTH alle Wortgruppen implementiert oder können nachgeladen werden.

! # #> #S ’ (∗ ∗/ ∗/MOD + +! +LOOP , − . .“ / /MOD 0< 0= 1+ 1− 2! 2∗ 2/ 2@ 2DROP 2DUP 2OVER 2SWAP : ; < <# = > >BODY >IN >NUMBER >R ?DUP @ ABORT ABORT“ ABS ACCEPT ALIGN ALIGNED ALLOT AND BASE BEGIN BL C! C, C@ CELL+ CELLS CHAR CHAR+ CHARS CONSTANT COUNT C“ CREATE DECIMAL DEPTH DO DOES> DROP DUP ELSE EMIT ENVIRONMENT? EVALUATE EXECUTE EXIT FILL FIND FM/MOD HERE HOLD I IF IMMEDIATE INVERT J KEY LEAVE LITERAL LOOP LSHIFT M∗ MAX MIN MOD MOVE NEGATE OR OVER POSTPONE QUIT R> R@ RECURSE REPEAT ROT RSHIFT S“ S>D SIGN SM/REM SOURCE SPACE SPACES STATE SWAP THEN TYPE U. U< UM∗ UM/MOD UNLOOP UNTIL VARIABLE WHILE WORD XOR [[’] [CHAR]]

! (x addr −−): Store **x** at address **addr**.

 # (ud1 −− ud2): Divide **ud1** by the number in BASE giving the quotient **ud2**. Convert the remainder (the least-significant digit of **ud1**) to a digit and insert it at the beginning of the pictured numeric output string.

#> (xd −− addr u): Drop **xd**. **addr** and **u** specify the resulting pictured numeric character string.

#S (ud −− 0.): Repeat # until the quotient **ud2** is zero.

' (−− xt) ⟨***name***⟩**:** Parse ⟨***name***⟩ up to space. Put its code field address on the stack. On failure the system returns "don't know ⟨***name***⟩".

((−−) ⟨***String***⟩**) immediate:** Disregard the input-stream up to). Exclude comments, which are not interpreted.

∗ (n1—u1 n2—u2 −− n3—u3): Multiply **n1|u1** by **n2|u2** giving the product **n3|u3**. Overflow is disregarded.

∗/ (n1 n2 n3 −− n4): Multiply **n1** by **n2** giving an intermediate double-cell result, which is divided by **n3** giving single-cell **n4**.

∗/mod (n1 n2 n3 −− m q): Multiply **n1** by **n2** giving an intermediate double-cell result, which is divided by **n3**, giving single-cell remainder **n4** and single-cell quotient **n5**.

+ (n1—u1 n2—u2 −− n3—u3): Add **n2|u2** to **n1|u1**, giving the sum **n3|u3**. All arguments must be of the same type.

+! (n addr −−): Add **n|u** to the single-cell number at address **addr**. Both arguments must be of the same type.

+LOOP (n −−) immediate restrict: Add **n** to the loop index. If the loop limit has not yet been reached, return to DO and renew execution of the loop. Otherwise, continue after the end of the loop.

, (x −−): Store **x** in the cell at HERE, increase HERE by the address units of one cell (4).

− (n1—u1 n2—u2 −− n3—u3): Subtract **n2|u2** from **n1—u1**, giving the difference **n3—u3**. All arguments must be of the same type.

. (n −−): Display **n** followed by space, if negative with leading minus-sign. (Picturing same as #.)

." (−−) ⟨***String***⟩**" immediate:** Compile ⟨***String***⟩ up to ". At run-time display ⟨***String***⟩.

/ (n1 n2 −− n3): Divide **n1** by **n2** giving the single-cell quotient **n3**. Attempting to divide by zero is reported as "Bus Error! /".

/mod (n1 n2 −− m q): Divide n1 by n2 giving the single-cell remainder n3 and the single-cel l quotient n4. Attempting to divide by zero is reported as "Bus Error ! /MOD".

0< (n −− flag): Flag is true, if n is less than zero.

0= (n −− flag): Flag is true, if x is equal to zero.

1+ (n1 −− n2): Add 1 to n1|u1 giving the sum n2|u2.

1− (n1 −− n2): Subtract 1 from n1|u1 giving the difference n2|u2.

2! (d addr −−): Store x2 at address addr and x1 in the next following cell.

2∗ (n1 −− n2): Shift x1 one bit toward the most significant bit. Put zero into the leas t significant bit. (Multiply x1 by 2 giving x2.)

2/ (n1 −− n2): Shift x1 one bit toward the least significant bit. Leave the most signif icant bit unchanged. (Divide x1 by 2 giving x2.)

2@ (addr −− d): Put cell pair x1x2 from address addr and the next following cell on the stack.

2DROP (d −−): Drop cell pair x1x2 form the stack.
2DUP (d −− d d): Duplicate cell pair x1x2 on the stack.
2OVER (d1 d2 −− d1 d2 d1): Copy cell pair x1x2 to the top of the stack.
2SWAP (d1 d2 −− d2 d1): Exchange the top two cell pairs.
: (−− csys) ⟨*name*⟩:⟨*name*⟩ (... −− ...): Colon definition. Parse ⟨***name***⟩ up to a space. Create a new forth word of the forth words immediately following until the concluding ; (semicolon), also ;CODE or [. At run-time these words are executed, when ⟨***name***⟩ is executed.

glos ; ; (semicolon) concludes a colon definition of a forth word. Also definitions, which started with :NONAME.

; (csys −−) immediate:
< (n1 n2 −− flag): Flag is true, if n1 is less than n2.
<# (d −− d): Initialize the pictured numeric output conversion process.
= (n1 n2 −− flag): Flag is true, if x1 is bit for bit the same as x2.
> (n1 n2 −− flag): Flag is true, if n1 is greater than n2.
>BODY (xt −− addr): Calculate the PFA (parameterfield-address) from the CFA (codefield-ad dress).
>IN (−− addr): User variable. addr points to a cell containing number of characters al ready interpreted. (The offset in characters from the start of the input buffer to the start of the parse area.)
>NUMBER (d1 addr1 u1 −− d2 addr2 u2): Convert a string of number-signs (unsigned) into an absolute double cell number. The characters in the string must correspond to the number stored in BASE. For a single transformation ud1 is zero, addr1 and u1 are the address and length of the string. ud2 is the result. addr2 points to the first character after the string or at error to the first character not converted. At success u 2 is zero, at error it contains the number of characters not converted. Overflow of ud2 is undecided. ¿NUMBER can be used cumulatively, e.g. when transforming a time string ss.mm.hh to a time counter reading by submitting each partial strin g with the applicable BASE. Then ud1 is the previous intermediate result.
>R (x −−) (RS −− x) restrict: Move x to the return stack, drop it from the stack.
?DUP (n −− n n / 0):
@ (addr −− x): x is the value stored in addr.
ABORT (−−): Empty the data stack. Perform QUIT, which includes emptying the retur n stack. No message. The system is partially reinitialized: warm start.
ABORT“ (flag −−) ⟨*String*⟩” immediate restrict: ¡text» If flag is not zero, display ¡text¿ and then execute ABORT.
ABS (n −− u): u is the absolute value of n.
ACCEPT (addr len1 −− len2): Receive a string of at most u1 characters. Display each character im mediately. The characters are stored from addr. u2 is the number of characters r eceived. Input is terminated by a line-terminator (Return) which is not stored w ith the string (similar to EXPECT). ACCEPT owns a legacy specialty coming down f rom EXPECT of Forth83. You pre-load the buffer with a counted string and supply addr and count. Before calling ACCEPT this count, which is u1, is negated. By th is that string is displayed as editable input. The standard foresees a maximum o f 32k bytes, bigFORTH works with a width of 32 bits.
ALIGN (−−): If HERE is uneven, increase it to the next even address. (NOOP for 38 6, compiles a blank character for 68k.)
ALIGNED (addr −− addr'): addr2 is the first aligned address greater than or equal to addr1.

ALLOT (n −−): If n is greater than zero reserve n bytes of data space. If n is less than zero, release absolute n bytes of data space. If n is zero do nothing.

AND (x1 x2 −− x3): x3 is the bit-by-bit logical ÄNDöf x1 with x2. Only if both input bi ts are 1, the resulting bit at the same place is 1, otherwise it is zero.

BASE (−− addr): User variable. Cell addr holds the current number base (2..36).

BEGIN (−−) immediate restrict: Marker for the beginning of a loop, which ends with REPEAT or UNTIL.

BL (−− bl): Constant: The ASCII-value of a space (blank).

C! (c addr −−): Store the character c at address addr.

C, (c −−): (C-comma) Store the character c at the address of HERE. HERE is increase d by 1.

C@ (addr −− c): Fetch from address addr one byte onto the stack.

CELL+ (addr −− addr'): Add the size of a cell (4 bytes) to addr1.

CELLS (n1 −− n2): n2 is the size in bytes on n1 cells.

CHAR (−− c) ⟨*Char*⟩: ¡text¿ Parse ¡text¿ up to a space. c is the value of the first charact er of ¡text¿.

CHAR+ (c-addr −− c-addr'): Add the storage length of one character to addr1 (1, as long as one c haracter occupies one byte).

CHARS (n1 −− n2): Multiply n1 by the storage length of one character (NOOP - as long as one character occupies one byte).

CONSTANT (n −−) ⟨*name*⟩:⟨*name*⟩ (−− n): ⟨*name*⟩ Compile-time: Define the constant ⟨*name*⟩ with the value x. At run-time put value x on the stack.

COUNT (addr −− addr' u): addr1 is the address of a counted string. addr2 points to the first c haracter, u is the number of characters starting at addr2.

C" (−− addr) ⟨*String*⟩" immediate: ¡text¿ Parse ¡text¿ up to "(double-quote). At run-time return addr, the address of the counted string ¡text¿.

CREATE (−−) ⟨*name*⟩:⟨*name*⟩ (−− addr): ⟨*name*⟩ Create a word header for ⟨*name*⟩. The related data space must yet be assigned (e.g. by ALLOT). At run-time ⟨*name*⟩ puts the address of this da ta space on the stack. The function of ⟨*name*⟩ can be changed by using DOES¿.

DECIMAL (−−): Set the number base to 10.

DEPTH (−− n): +n is the number of single-cell values contained in the data stack be fore +n was placed on the stack.

DO (end start −−) immediate restrict: All arguments must be of the same type. Start of a counted loop. n1|u1 a re the end, n2|u2 the start value of the loop index. The loop ends at LOOP or +L OOP. At run-time it is terminated when the next cycle would reach or exceed the end value. Using LEAVE, which sets the index to limit, execution can be continue d prematurely after LOOP or +LOOP. Using UNLOOP EXIT, the forth word in which th e loop is being executed is left prematurely. See also I and J.

DOES> (−− addr) immediate restrict: Extend a definition, which started with CREATE ⟨*name*⟩, with the forth words following. They decide the behaviour of the word at run-time and are exec uted each time ⟨*name*⟩ is called.

DROP (x −−): Remove x from the stack.

DUP (x −− x x): Duplicate x.

ELSE (−−): Start of the alternate part of an IF.

EMIT (c −−): Display x-th character of the character set.

ENVIRONMENT? (addr u −− values t / f): Search the vocabulary ENVIRONMENT for the string addr u. At su ccess execute the word which puts i*x and true onto the stack. At failure put fa lse onto the stack.

EVALUATE (addr u −−): Save the current input source specification. Store minus-one (-1) in SOURCE-ID. Make the string described by addr and u both the input source and input buffer, set ¿IN to zero, and interpret. When the parse area is empty, re store the prior input source specification. Other stack effects are due to the words EVALUATEd.

EXECUTE (xt −−): Execute the forth word with address addr.

EXIT (−−) restrict: Return control to the calling definition. The return address must be o n top of the return stack. When within a do-loop UNLOOP must be executed prior t o leaving.

FILL (addr u c −−): If u is greater than zero, store c in each of u consecutive characters of memory beginning at addr.

FIND (addr −− addr f / xt t): Find the definition named in the counted string at addr. All vocabular ies listed in the vocabulary stack are searched from top to bottom. If the defin ition is not found, return addr and zero. If found return the execution token cf a and n, which tells whether the word is immediate and/or restrict: -1: none. -2: restrict. 1: immediate. 2: immediate restrict.

FM/MOD (d n1 −− n2 n3): Divide d by n1 giving the floored quotient n3. n2 is the remainder, its sign being always the same as n1. n1 * n3 + n2 = d is always valid. Equival ent to M/MOD.

HERE (−− addr): addr is the pointer to the first free space in data-space.

HOLD (c −−): Add character c to the beginning of the pictured numeric output string .

I (−− n): n|u is a copy of the current (innermost) loop index.

IF (flag −−) immediate restrict: If all bits of x are zero, continue execution after the applicable ELSE or (if not present) THEN of the same nesting level.

IMMEDIATE (−−): Make the most recent definition an immediate word, which thereaft er is also executed during compilation.

INVERT (x1 −− x2): Invert all bits of x1, giving its logical inverse x2.

J (−− n): n|u is a copy of the next-outer loop index.

KEY (−− c): Needs ANS.STR. Receive one character, which is not displayed. c is the numerical value of the character received (1 - 255).

LEAVE (−−): Leave the loop prematurely. Continue execution after the closing LOOP or +LOOP.

LITERAL (n −−) immediate: Compile x into the current definition. At run-time place x on the s tack.

LOOP (−−) immediate restrict: Add 1 to the loop index. If the loop limit has not yet been reached, e xecute the words within the loop. Otherwise, continue after the end of the loop. See also DO.

LSHIFT (u1 n −− u2): Shift u1 left logically by u bits. Put zeroes into vacated least sig nificant bits.

M∗ (n1 n2 −− d): d is the double-cell signed product of n1 times n2.

MAX (n1 n2 −− n3): n3 is the greater of n1 and n2.

MIN (n1 n2 −− n3): n3 is the lesser of n1 and n2.

MOD (n1 n2 −− m q): Divide n1 by n2, giving the single-cell remainder n3.

MOVE (addr1 addr2 u −−): If u is greater than zero, copy u characters from addr1 to addr2.

NEGATE (n1 −− n2): Negate n1, giving its arithmetic inverse n2.

OR (x1 x2 −− x3): x3 is the bit-by-bit inclusive-or of x1 with x2. If any of the input bit s is one, the resulting bit is also one. Only if both input bits are zero, the r esulting bit is also zero.

OVER (x1 x2 −− x1 x2 x1): Place a copy of x1 on top of the stack.

POSTPONE (−−) ⟨*name*⟩ immediate restrict: ⟨***name***⟩ Parse ⟨***name***⟩ up to a space. Find ⟨***name***⟩ and compile it in to the current definition.

QUIT (−−): Empty the return stack, set SOURCE-ID to zero, switch to the user inpu t device and interpretation state. Accept a line of input into the input buffer, set ¿IN to zero, and interpret. On completion display the system prompt.

R> (−− x) (RS x −−) restrict: Move x from the return stack to the data stack.

R@ (−− x) (RS x −− x) restrict: Copy x from the return stack to the data stack.

RECURSE (−−) immediate restrict: Compile the definition, which is being gene- rated, and thereby defin e a recursion independent of the name of that definition, even when it was origi nally generated by :NONAME.

REPEAT (−−) immediate restrict: End marker of a BEGIN loop.

ROT (x1 x2 x3 −− x2 x3 x1): Rotate the top three stack entries.

RSHIFT (u1 n −− u2): Shift u1 right logically by u bits (unsigned, contrary to ¿!¿). Put zeroes into vacated most significant bits.

S“ (−− addr u) ⟨*String*⟩” immediate: ¡text¿ In direct mode and when compiling, parse ¡text¿ up to ”(double q uote). In direct mode and at run-time return address and count of the stored ¡te xt¿.

S>D (n −− d): Convert the number n to the double-cell number d with the same numerica l value. Alias for EXTEND.

SIGN (n −−): If n is negative, add a minus sign to the beginning of the pictured nu meric output string.

SM/REM (d n1 −− n2 n3): Divide d by n1 giving the symmetric quotient n3 . n2 is the remainde r, its sign being always the same as d. All stack arguments are signed. n1 * n3 + n2 = d is always valid.

SOURCE (−− addr u): addr is the address of, and u is the number of characters in, the in put buffer.

SPACE (−−): Display one space.

SPACES (n −−): If n is greater than zero, display n spaces.

STATE (−− addr): The value in addr is true when in compilation state, false otherwise. Must not be altered directly.

SWAP (x1 x2 −− x2 x1): Exchange the top two stack items.

THEN (−−): Continue execution, end of an IF. See also ELSE and IF.

TYPE (addr u −−): If u is greater than zero, display the character string specified by c -addr and u.

U. (u −−): Display u followed by a space.

U< (u1 u2 −− flag): Flag is true if u2 is greater than u1. Unsigned numbers.

UM∗ (u1 u2 −− ud): Multiply u1 by u2, giving the unsigned double-cell product ud. Values a nd arithmetic are unsigned.

UM/MOD (ud u −− um uq): Divide ud by u1, giving the quotient u3 and the remainder u2. Values and arithmetic are unsigned.

UNLOOP (−−) (RS limit index −−) restrict: Discard the loop-control parameters for the current nesting level. An UNLOOP is required for each nesting level before the definition may be EXITed .

UNTIL (flag −−) immediate restrict: If all bits of x are zero, continue execution at the applicable BEGIN .

VARIABLE (––) ⟨*name*⟩:⟨*name*⟩ (–– addr): ⟨*name*⟩ Parse ⟨*name*⟩ up to a space. Create a definition and reserve one data cell at an aligned address. At run-time put this address on the stack.

WHILE (flag ––) immediate restrict: Leave a BEGIN loop, if all bits of x are zero and then continue exec ution after the applicable UNTIL or REPEAT. If there are more than one WHILE in a loop, each additional WHILE requires an additional THEN at the end of the loop .

WORD (c –– addr): Parse the input buffer until character c is reached. addr is the addre ss of the counted string containing the input. If the word is to serve as header of a new definition, maximum 32 bytes including count byte are permitted.

XOR (x1 x2 –– x3): x3 is the bit-by-bit exclusive-or of x1 with x2. The resulting bit is z ero, when both input bits are equal (both zero or both one), otherwise always on e.

[(––) immediate: Enter interpretation state. (Turn off the compiler.)

[’] (–– xt) ⟨*name*⟩ immediate restrict: ⟨*name*⟩ Parse ⟨*name*⟩ and store its codefield address in the definition a s a constant. At run-time put the CFA on the stack.

[CHAR] (–– c) ⟨*Char*⟩ immediate restrict: ¡text¿ Parse ¡text¿ up to a space. At run-time place c, the value of the first character of ¡text¿, on the stack.

] (––): Enter compilation state. (Turn off the interpreter.)

ENVIRONMENT (––) (VS –– ENVIRONMENT):

ENVIRONMENT? (addr u –– values t / f): Search the vocabulary ENVIRONMENT for the string addr u. At su ccess execute the word which puts i*x and true onto the stack. At failure put fa lse onto the stack.

ENVIRONMENTAL (–– values t / f) ⟨*name*⟩:

S>D (n –– d): ist ein Alias für EXTEND und erweitert **n** vorzeichenbehaftet auf **d**

Die CORE–Extension–Wortgruppe

**#TIB .(.R 0<> 0> 2>R 2R> 2R@ :NONAME <> ?DO AGAIN C“ CASE COMPILE, CONVERT ENDCASE ENDOF ERASE EXPECT FALSE HEX MARKER NIP OF PAD PARSE PICK QUERY REFILL RESTORE–INPUT ROLL SAVE–INPUT SOURCE–ID SPAN TIB TO TRUE TUCK U.R U> UNUSED VALUE WITHIN [COMPILE] **

2>R (d ––) (RS –– d): Transfer cell pair x1 x2 from the data stack to the return stack.

2R> (–– d) (RS d ––): Transfer cell pair x1 x2 from the return stack to the data stack.

2R@ (–– d) (RS d –– d): Needs ANS.STR. Copy cell pair x1 x2 from the return stack.

:NONAME (–– cfa) ⟨*Word*⟩* ;: Generate a word without name. To make it useable put its code field address onto the stack.

MARKER (––) ⟨*name*⟩:⟨*name*⟩ (––): ⟨*name*⟩ Generate a definition with the Name ⟨*name*⟩ registering the cu rrent state of the dictionary. At execution of ⟨*name*⟩ that state of the dictiona ry is restored. All words defined since the definition of ⟨*name*⟩ including ¡name ¿ are forgotten.

REFILL (–– flag): Attempt to fill the input buffer. Keyboard: receive into terminal in put buffer, make the result the input buffer; an empty line is successful. Block : make the next block the input source and current input buffer by adding 1 to B LK; the value

of BLK must be a valid block number. Text file: read the next line , make the result the current input buffer. If successful set ¿IN to zero and re turn true, else return false. String from EVALUATE: return false and perform no other action.

SAVE–INPUT (−− x1 .. xn n): Save the position in an input-stream for RESTORE-INPUT. n is the number of parameters describing the input-stream, excluding n.

RESTORE–INPUT (x1 .. xn n −−): Restore the position in an input-stream saved by RESTORE-INPU T, acting on the same input-stream used by that word. Flag is true if the input source cannot be so restored.

SOURCE–ID (−− 0 / −1 / file): Identify the input-stream. Zero is direct mode. -1 is a string fr om EVALUATE, u is the file-handle from LOADFILE.

TUCK (x1 x2 −− x2 x1 x2): Entspricht UNDER

UNUSED (−− n): Liefert die Anzahl freier Bytes im Dictionary

WITHIN (u1 u2 u3 −− flag): Entspricht in etwa UWITHIN und ersetzt UWITHIN. Gibt true, wenn $\mathbf{u_3 - u_2 > u_1 - u_2}$. Da hier im vorzeichenlosen Zahlenraum gerechnet wird, ergibt eine Vertauschung der Bereichsgrenzen **u2** und **u3** eine Invertierung von **flag**.

2.2. Die BLOCK–Wortgruppe

BLK BLOCK BUFFER EVALUATE FLUSH LOAD SAVE–BUFFERS UPDATE

Die BLOCK–Extension–Wortgruppe

**EMPTY–BUFFERS LIST REFILL SCR THRU **

2.3. Die DOUBLE–Wortgruppe

2CONSTANT 2LITERAL 2VARIABLE D+ D− D. D.R D0< D0= D2∗ D2/ D< D= D>S DABS DMAX DMIN DNEGATE M∗/ M+ 2ROT DU<

2LITERAL (d −−) immeditate: Compiliert eine doppelt genaue Zahl

D2∗ (d1 −− d2): Verdoppelt **d1**

D2/ (d1 −− d2): Halbiert **d1**

D>S (d −− n): Entspricht DROP. Löscht die höherwertige Hälfte von **d** und wandelt damit eine doppelt genaue Zahl (im Rahmen der Genauigkeit) in eine Integerzahl.

DMAX (d1 d2 −− d1 / d2): Wählt die größere doppelt genaue Zahl aus

DMIN (d1 d2 −− d1 / d2): Wählt die kleinere doppelt genaue Zahl aus

M∗/ (d1 n1 u2 −− d2): Dividiert das Produkt aus **d1** und **n1** durch **u2** und gibt den Quotient als **d2** zurück. Das Zwischenergebnis ist 3 Zellen groß

M+ (d1 n −− d2): Erhöht **d1** um **n**

2ROT (d1 d2 d3 −− d2 d3 d1): Rotiert drei doppelt genaue Zahlen (wie ROT)

DU< (ud1 ud2 −− flag): flag is true, wenn $\mathbf{ud_1 < ud_2}$

2.4. Die EXCEPTION–Wortgruppe

ANS–FORTH hat eine neue Ausnahmebehandlung: Das CATCH/THROW–Konzept. Im Fehlerfall wird mit THROW an die Stelle gesprungen, an der das letzte CATCH stand. Stack und Returnstack werden bereinigt. Der Fehlercode, der THROW übergeben wurde, liegt nun an oberster Stelle auf dem Stack. Bei korrekter Ausführung der an CATCH

übergebenen CFA wird eine 0 zurückgegeben. Dieses Verfahren öffnet zwei Türen zur komfortablen Fehlerbehandlung:

- Wörter, die möglicherweise zu Fehlern führende Unterprozeduren haben, können sich gegen einen Absturz wappnen und eine eigene Fehlerbehandlung übernehmen. DUMP gibt bei ungültigen Adressen z. B. zwei Striche aus, anstatt abzustürzen. Auch eine standardisierte Fehlerbehandlung in eigenen Applikationen ist nun möglich.

- Geben Wörter einen Fehlercode zurück, kann mit einem nachfolgenden THROW die Fehlerbehandlung nach weiter außen durchgereicht werden.

CATCH THROW

HANDLER (−− addr): In dieser User–Variable steht die Adresse des zuletzt angelegten Error–Frames

CATCH (x1 .. xn cfa −− y1 .. ym 0 / z1 .. zn error): Legt einen Error–Frame auf den Returnstack und ruft **cfa** auf (wie EXECUTE). Wird während der Ausführung ein THROW ausgeführt, werden die Stacks bereinigt und die dem THROW übergebene Fehlernummer **error** wird auf den Stack gelegt, es sei denn, das THROW wird von einem später (innerhalb von **cfa**) aufgerufenen CATCH aufgefangen. Wird **cfa** korrekt ausgeführt, wird eine 0 zurückgegeben, die Stacks werden nicht bereinigt.

THROW (.. error −− ..): Ist **error** 0, so geschieht nichts. Ansonsten wird der Error–Frame vom Returnstack geholt, die Stacks werden zurückgesetzt und **error** wird auf dem Stack gelegt.

Die EXCEPTION–Extension–Wortgruppe

Auch die bisher in FORTH vorhandenen Ausnahmebehandlungen ABORT und ABORT“ werden mit THROW weitergeleitet. ABORT entspricht −1 THROW, ABORT“ speichert den String in ”ERROR und führt −2 THROW aus, wenn die Error–Flag true war.

ABORT ABORT“

”ERROR (−− addr): In dieser User–Variable wird der Error–String von ABORT“ gespeichert. Dieser String ist nur gültig, wenn CATCH −2 oder −$100 (ERROR“) zurückgibt.

2.5. Die FACILITY–Wortgruppe

AT–XY KEY? PAGE

AT–XY (x y −−): Entspricht SWAP AT, nur ist hier die Zeile der TOS, die Spalte der NOS

KEY? (−− flag): Ist im Gegensatz zum bigFORTH–KEY? nur true, wenn wirklich ein Zeichen ($\neq 0$) und keine Steuertaste gedruckt wurde

Die FACILITY–Extension Wortgruppe

EKEY EKEY>CHAR EKEY? EMIT? MS TIME&DATE

EKEY (−− n): Liefert einen erweiterten Tastencode. Entspricht bigFORTH's KEY, das ohnehin schon Scancode und Zeichencode zurückliefert. Mausklicks werden in bigFORTH als $\$8000 + x * \$100 + y$ codiert.

EKEY>CHAR (n −− c t / f): Wandelt einen Tastencode in das zugehörige Zeichen **c** um und liefert true, wenn erfolgreich, false sonst

EKEY? (−− flag): Liefert true, wenn eine Taste gedrückt wurde und noch nicht aus dem Tastaturpuffer abgeholt ist, false sonst.

EMIT? (−− flag): Liefert true, wenn ein Zeichen ohne Verzögerung ausgegeben werden kann. Achtung: Das gilt nicht für kompliziertere Steuerzeichen wie CR oder Seitenvorschub! Diese können trotzdem eine Verzögerung bewirken.

MS (n −−): Wartet (mindestens) **n** Millisekunden

TIME&DATE (−− sec min hour day month year): Liefert Datum und Uhrzeit. Die Wertebereiche entsprechen dabei dem einer Uhr oder eines Kalenders.

2.6. Die FILE–Wortgruppe

ANS–FORTH definiert ein standardisiertes Interface auf Dateien, das in etwa mit den üblichen Betriebssystemanbindungen vergleichbar ist. Dateizugriffe erfolgen entweder über den Dateinamen oder über eine File–ID (fid), die in bigFORTH dem File Control Block entspricht. Die File–ID ist ein Handle auf einen Speicherbereich, in dem Länge, Handle des Betriebssystem, wie oft die Datei geöffnet wurde und der Dateiname steht.

Die von ANS–FORTH definierten Zugriffsrechte auf eine Datei werden von bigFORTH ignoriert. Jede Datei wird immer zum Lesen und Schreiben geöffnet. Auch gibt es keinen Unterschied zwischen Text– und Binärdateien.

Die zurückgegebenen Fehlermeldungen (ior) entsprechen denen, die bigFORTH im Fehlerfall mit THROW zurückgibt. Eine 0 bedeutet dabei „kein Fehler".

(BIN CLOSE–FILE CREATE–FILE DELETE–FILE FILE–POSITION FILE–SIZE INCLUDE–FILE INCLUDED OPEN–FILE R/O R/W READ–FILE READ–LINE REPOSITION–FILE RESIZE–FILE S" SOURCE–ID W/O WRITE–FILE WRITE–LINE

R/O (−− 0): Datei nur zum Lesen öffnen

W/O (−− 1): Datei nur zum Schreiben öffnen

R/W (−− 2): Datei zum Lesen und Schreiben öffnen

BIN (x1 −− x2): Modifiziert ein Zugriffsrecht so, daß die Datei als Binärdatei geöffnet wird. In bigFORTH ist das ein NOOP.

OPEN–FILE (addr u x −− fid ior): Öffnet die Datei mit dem Namen **addr u** und dem Zugriffsrecht **x**. Zurückgegeben wird die File–ID bzw. 0, wenn der Versuch erfolglos war und die Fehlernummer **ior**.

CREATE–FILE (addr u x −− fid ior): Erzeugt die neue (leere) Datei mit dem Namen **addr u**, öffnet diese neue Datei mit den Zugriffsrechten **x** und gibt dieselben Werte zurück wie OPEN–FILE.

CLOSE–FILE (fid −− ior): Schließt die Datei **fid** und gibt eine Fehlernummer **ior** zurück

DELETE–FILE (addr u –– ior): Löscht die Datei mit dem Namen **addr u** und gibt eine Fehlernummer **ior** zurück

FILE–POSITION (fid –– ud ior): Gibt die Position **ud** des Schreib/Lese–Zeigers innerhalb der Datei **fid** relativ zum Dateianfang zurück

REPOSITION–FILE (ud fid –– ior): Setzt die Position des Schreib/Lese–Zeigers der Datei **fid** auf **ud** relativ zum Dateianfang

FILE–SIZE (fid –– ud ior): Gibt die Größe **ud** der Datei **fid** zurück

RESIZE–FILE (ud fid –– ior): Setzt die Größe der Datei **fid** auf **ud**. Falls die Datei vergrößert wird, ist nicht definiert, was sich in dem neuen Teil der Datei befindet.

READ–FILE (addr u1 fid –– u2 ior): Versucht, **u1** Bytes von der aktuellen Position des Schreib/Lese–Zeigers aus der Datei **fid** an die Adressen ab **addr** zu lesen. Zurückgegeben wird die Anzahl tatsächlich gelesener Bytes **u2** und eine Fehlernummer **ior**. Der Schreib/Lese–Zeiger steht nun hinter dem gelesenen Bereich.

WRITE–FILE (addr u fid –– ior): Schreibt **u** Bytes von **addr** an von der aktuellen Position des Schreib/Lese–Zeigers in die Datei **fid**. Zurückgegeben wird die Fehlernummer **ior**. Der Schreib/Lese–Zeiger steht nun hinter dem geschriebenen Bereich. Die Datei wird ggf. verlängert.

READ–LINE (addr u1 fid –– u2 flag ior): Liest eine Zeile aus der Textdatei **fid** der maximalen Länge **u1** von der aktuellen Position des Schreib/Lese–Zeigers an den Speicherbereich ab **addr**. Zurückgegeben wird die Länge der Zeile **u2**, true, solange das Dateiende noch nicht erreicht wurde und die Fehlernummer **ior**. Der Schreib/Lese–Zeiger zeigt auf die nächste Zeile; bei überlangen Zeilen ($u_2 = u_1$) auf den Rest der Zeile, der noch nicht eingelesen wurde.

WRITE–LINE (addr u fid –– ior): Schreibt die Zeile **addr u** ab der aktuellen Position des Schreib/Lese–Zeigers in die Datei **fid** inclusive der Zeilenendezeichen. Der Schreib/Lese–Zeiger steht nun hinter der Zeile, die Datei wurde ggf. verlängert. Zurückgegeben wird auch eine Fehlernummer **ior**.

INCLUDE–FILE (fid ––): Lädt die Datei **fid** als Textdatei Zeile für Zeile. Die Datei wird nach dem erfolgreichen Laden geschlossen.

INCLUDED (addr u ––): Lädt die Datei mit dem Namen **addr u** als Textdatei Zeile für Zeile

Die FILE–Extension–Wortgruppe

FILE–STATUS FLUSH–FILE REFILL RENAME–FILE

FILE–STATUS (addr u –– dta ior): Liefert den Status der Datei mit dem Namen **addr u**. Zurückgegeben wird die Adresse der Disk Transfer Area, die alle nötigen Informationen über die Datei enthält und eine Fehlernummer, falls die Suche erfolglos war.

FLUSH–FILE (fid –– ior): Schreibt alle Blöcke der Datei **fid** zurück

RENAME–FILE (addr1 u1 addr2 u2 –– ior): Benennt Dateien um. Die Datei mit dem Namen **addr1 u1** bekommt den Namen **addr2 u2**. Die Fehlernummer **ior** wird zurückgegeben.

2.7. Die FLOAT–Wortgruppe

Dank ANS–FORTH gibt es jetzt endlich einen Standard für Fließkommaerweiterungen für FORTH. Die meisten Probleme unterschiedlicher Implementierungen wurden damit

gelöst, allerdings hat sich die Kontroverse über getrennte oder einen gemeinsamen Stack auch im Standard nicht lösen lassen. Es ist daher beides erlaubt, getrennte Stacks werden aber bevorzugt. Da sich bigFORTH an den Vorgänger des ANSI–Standards, den der FORTH Vendor Group gehalten hat, haben sich auch bei den Namen keine gravierenden Änderungen ergeben. Lediglich komplexere Funktionen, wie Exponent oder transzendente Funktionen haben konsequenterweise ein F als Prefix.

Die Fließkommawörter befinden sich nach wie vor im Vokabular FLOAT, es muß also `FLOAT ALSO` eingegeben werden (nachdem FLOAT.SCR geladen wurde), bevor Fließkommabefehle verwendet werden können.

Fließkommazahlen in der Eingabe werden im ANS–FORTH mit einem „E“ oder „D“ (groß oder klein ist egal) als Trennzeichen zwischen Mantisse und Exponent markiert (ein leerer Exponent bedeutet dabei „0“). Daher kann man keine Hex–Zahlen eingeben, das Ergebnis wäre nicht eindeutig. bigFORTH verwendet die eigene, eindeutige Notation mit Prefix „!“ und Trennzeichen „’ “, wenn ANS.STR nicht geladen wurde, die vom Standard, wenn ANS.STR vorher geladen wurde. `' NOOP Alias ANS` vor dem Laden hat denselben Effekt

>FLOAT D>F F! F* F+ F– F/ F0< F0= F< F>D F@ FALIGN FALIGNED FCONSTANT FDEPTH FDROP FDUP FLITERAL FLOAT+ FLOATS FLOOR FMAX FMIN FNEGATE FOVER FROT FROUND FSWAP FVARIABLE REPRESENT

>FLOAT (addr u –– flag) (FS –– f /): Wandelt den String **addr u** in eine Fließkommazahl um. Dabei werden führende Leerzeichen und solche am Ende des Strings ignoriert. Bei erfolgreicher Wandlung wird true zurückgegeben, bei Mißerfolg false; es wird dann auch keine FP–Zahl zurückgegeben.

F>D (–– d) (FS f ––): Konvertiert eine Fließkommazahl in eine Integerzahl, rundet aber anders als in bigFORTH nicht nach $-\infty$, sondern nach 0. Analog F>S, welches zwar nicht im ANSI–Standard vorgesehen ist, aber eine Abkürzung für F>D D>S sein sollte.

FALIGN (––): Erhöht HERE bis zum nächsten Fließkomma–Alignment

FALIGNED (addr –– fp–addr): Erhöht **addr** bis zum nächsten Fließkomma–Alignment

FLOAT+ (addr –– addr’): Erhöht **addr** um die Länge einer Fließkommazahl

FLOATS (n1 –– n2): Multipliziert **n** mit der Länge einer Fließkommazahl

FLOOR (––) (FS f –– [f]): Vormals INT. Rundet **f** zur nächstkleineren ganzen Zahl ab.

REPRESENT (addr u –– n flag1 flag2) (FS f ––): Basisroutine zur Wandlung von Fließkommazahlen in Strings. Aus der Fließkommazahl wird der Exponent **n** bezogen auf die aktuelle Zahlenbasis gewonnen. **flag1** ist true, wenn **f** negativ ist, false sonst. **flag2** ist nur false, wenn sich die Zahl nicht wandeln lies, z. B. ∞ oder NaN (Not a Number). An die **u** Adressen ab **addr** wird die Mantisse als String abgelegt. Das Komma steht dabei implizit vor der ersten Ziffer des Strings.

Die FLOAT–Extension–Wortgruppe

Neben den absolut notwendige Wörtern im FLOAT–Wortgruppe gibt es noch Wörter für den Zugriff auf einfach bzw. doppelt genaue Zahlen nach dem IEEE–754–Standard, brauchbare Ausgabewörter und eine Reihe nützlicher mathematischer Funktionen.

DF! DF@ DFALIGN DFALIGNED DFLOAT+ DFLOATS F F. FABS FACOS FACOSH FALOG FASIN FASINH FATAN FATAN2 FATANH FCOS FCOSH FE. FEXP FEXPM1 FLN FLNP1 FLOG FS. FSIN FSINCOS FSINH FSQRT FTAN FTANH F~ PRECISION SET–PRECISION SF! SF@ SFALIGN SFALIGNED SFLOAT+ SFLOATS**

SF! (addr ––) (FS f ––): Speichert **f** als einfach genaue Fließkommazahl an **addr** gemäß IEEE–754 ab

SF@ (addr ––) (FS –– f): Liest von **addr** die einfach genaue Fließkommazahl **f** und legt sie auf den Fließkommastack

SFALIGN (––): Erhöht HERE zum nächstgrößeren single float Alignment

SFALIGNED (addr –– sf–addr): Erhöht **addr** zum nächstgrößeren single float Alignment **sf–addr**

SFLOAT+ (addr –– addr'): Erhöht **addr** um den Speicherbedarf einer single float–Zahl (4 Bytes)

SFLOATS (n1 –– n2): Multipliziert **n1** mit dem Speicherbedarf einer single float–Zahl (4 Bytes)

DF! (addr ––) (FS f ––): Speichert **f** als doppelt genaue Fließkommazahl an **addr** gemäß IEEE–754 ab

DF@ (addr ––) (FS –– f): Liest von **addr** die doppelt genaue Fließkommazahl **f** und legt sie auf den Fließkommastack

DFALIGN (––): Erhöht HERE zum nächstgrößeren double float Alignment

DFALIGNED (addr –– sf–addr): Erhöht **addr** zum nächstgrößeren double float Alignment **sf–addr**

DFLOAT+ (addr –– addr'): Erhöht **addr** um den Speicherbedarf einer double float–Zahl (8 Bytes)

DFLOATS (n1 –– n2): Multipliziert **n1** mit dem Speicherbedarf einer double float–Zahl (8 Bytes)

F (––) (FS f1 –– f2):** **f2** ist das Quadrat von **f1**

FSQRT (––) (FS f1 –– f2): **f2** ist die Quadratwurzel von **f1**. Wenn **f1** negativ ist, wird mit einer Fehlermeldung abgebrochen.

FABS (––) (FS f –– |f|): |**f**| ist der Absolutbetrag von **f**

FEXP (––) (FS f1 –– f2): Vormals EXP. **f2** ist die **f1**–te Potenz von e (Eulersche Zahl) e^{f_1}.

FEXPM1 (––) (FS f1 –– f2): $f_2 = e^{f_1} - 1$. **f2** ist die um eins verminderte **f1**–te Potenz von e. Diese Funktion ist im Bereich von 0 wesentlich genauer und erlaubt damit eine genaue Berechnung z. B. vom Sinus Hyperbolis.

FALOG (––) (FS f1 –– f2): **f2** ist die **f1**–te Potenz von 10

FLN (––) (FS f1 –– f2): **f2** ist der natürliche Logarithmus $\log_e(f_1)$ von **f1**

FLNP1 (––) (FS f1 –– f2): $f_2 = \log_e(f_1+1)$ ist die Umkehrfunktion von FEXPM1, der natürliche Logarithmus der um eins erhöhten Zahl **f1**

FLOG (––) (FS f1 –– f2): Vormals FLG. **f2** ist der Logarithmus von **f1** zur Basis 10 $f_2 = \log_{10}(f_1)$.

FSIN (––) (FS f1 –– f2): **f2** ist der Sinus von **f1**. $f_2 = \sin f_1$.

FCOS (––) (FS f1 –– f2): **f2** ist der Cosinus von **f1**. $f_2 = \cos f_1$.

FTAN (––) (FS f1 –– f2): **f2** ist der Tangens von **f1**. $f_2 = \tan f_1$.

FSINCOS (−−) (FS f1 −− f2 f3): f2 ist der Sinus, **f3** der Cosinus von **f1**. $f_2 = \sin f_1$, $f_3 = \cos f_1$.
FASIN (−−) (FS f1 −− f2): f1 ist der Sinus von **f2**. $f_2 = \sin^{-1} f_1$. Wenn **f1** betragsmäßig größer 1 ist, gibt es einen Abbruch mit Fehlermeldung.
FACOS (−−) (FS f1 −− f2): f1 ist der Cosinus von **f2**. $f_2 = \cos^{-1} f_1$. Wenn **f1** betragsmäßig größer 1 ist, gibt es einen Abbruch mit Fehlermeldung.
FATAN (−−) (FS f1 −− f2): f1 ist der Tangens von **f2**. $f_2 = \tan^{-1} f_1$.
FATAN2 (−−) (FS f1 f2 −− f3): f3 ist der Winkel, den der Vektor (**f1**, **f2**) gegenüber der **f1**–Achse gegen den Uhrzeigersinn hat. Sind beide Eingabewerte 0, wird $\pi/2$ zurückgegeben.

FSINH (−−) (FS f1 −− f2): f1 ist der Sinus Hyperbolis von **f2**
FCOSH (−−) (FS f1 −− f2): f1 ist der Cosinus Hyperbolis von **f2**
FTANH (−−) (FS f1 −− f2): f1 ist der Tangens Hyperbolis von **f2**
FASINH (−−) (FS f1 −− f2): f1 ist der Area Sinus von **f2**
FACOSH (−−) (FS f1 −− f2): f1 ist der Area Cosinus von **f2**
FATANH (−−) (FS f1 −− f2): f1 ist der Area Tangens von **f2**

F∼ (−− flag) (FS f1 f2 f3 −−): "F–proximate". Abhängig von **f3** wird auf folgende Bedingung geprüft:
$f_3 > 0$ $|f_1 - f_2| < f_3$
$f_3 = 0$ $f_1 = f_2$ (Darstellung von **f1** und **f2** identisch)
$f_3 < 0$ $|f_1 - f_2| < |f_3| * (|f_1| + |f_2|)$
Das alte F∼ ist nicht mehr vorhanden.

PRECISION (−− n): n ist die Anzahl Stellen in der Mantisse, die ausgegeben wird
SET–PRECISION (n −−): n ist die Anzahl Stellen, die zukünftig in der Mantisse ausgegeben werden und mit PRECISION zurückgegeben wird. SET–PRECISION wirkt per–Task, hinter PRECISION verbirgt sich also eine User–Variable.
F. (−−) (FS f −−): Gibt **f** in Fixpunkt–Notation mit einem abschließenden Space aus (FLOAT.SCR ohne ANS.STR geladen: wie früher in einem optimierten Format). Die optimale Notation ist über FX. weiter zugänglich, die Fixpunktnotation als FF..
FE. (−−) (FS f −−): Gibt **f** in Ingenieurs–Notation aus, also ⟨*Sign*⟩0.⟨*Mantisse*⟩E⟨*Exponent*⟩
FS. (−−) (FS f −−): Gibt **f** in wissenschaftlicher Notation aus, also 1–3 Stellen vor dem Komma, der Exponent als durch drei teilbare Zahl

2.8. Die LOCAL–Wortgruppe

ANS–FORTH sieht lokale Variablen vor. Allerdings gab es das Problem, daß keine zwei Implementierungen von lokalen Variablen gemeinsame Regeln hatten, oder etwa mit einer gemeinsamen Basis implementiert werden könnten[1]. Deshalb hat man sich auf eine sehr restriktive Basis geeinigt, die von den meisten etwas ausgefeilteren Local–Paketen emuliert werden kann.

Allerdings ist das Ergebnis wenig befriedigend: Locals können nur einmal in einem Wort definiert werden, und sind dann bis zum Ende des Wortes, bzw. bis zum Verlassen mit DOES> sichtbar. bigFORTH's lokale Variablen haben einen expliziten Scope; diese Scopes können ineinander geschachtelt werden, allerdings nicht über die Grenzen von

[1] Nicht einmal zwei FORTH–Systeme, bei denen derselbe Autor beteiligt ist: Vergleiche bigFORTH und gforth

Kontrollstrukturen hinaus (ähnlich wie in C oder Pascal), das Ende des Scopes muß explizit angezeigt werden.

Die Locals von ANS–FORTH werden über die Locals in bigFORTH emuliert, es sind also immer die mächtigeren Locals von bigFORTH vorhanden (siehe Seite 215).

(LOCAL) TO

Die LOCAL–Extension–Wortgruppe

LOCALS|

2.9. Die MEMORY–Wortgruppe

Was in bigFORTH mit einer sehr ausgefeilten Speicherverwaltung geleistet wird, versucht ANS–FORTH auf einfache Weise wenigstens teilweise zu ermöglichen: Das Allokieren von Speicher außerhalb des Dictionaries.

ALLOCATE FREE RESIZE

ALLOCATE (u −− addr ior): Belegt einen **u** Bytes großen Speicherblock, der an **addr** anfängt. Kann die Speicheranforderung nicht erfüllt werden, ist **ior** eine Fehlernummer $\neq 0$ und **addr** keine gültige Adresse. ALLOCATE wird über NEWPTR realisiert.

FREE (addr −− ior): Gibt den Speicherblock ab **addr** zurück. Es wird die Fehlerkennung **ior** zurückgegeben (0 bedeutet keinen Fehler). FREE wird über FREEPTR realisiert

RESIZE (addr1 u −− addr2 ior): Verändert die Größe des Speicherblocks ab **addr1** auf **u** Bytes. Anders als SETPTRSIZE wird ggf. ein neuer Block an einer anderen Adresse belegt, der überlappende Inhalt hineinkopiert und die neue Adresse **addr2** zurückgegeben. Schlägt die Vergrößerung fehl, ist **ior** nicht 0 und **addr2** entspricht **addr1**, dessen Inhalt nicht wieder freigegeben wurde.

2.10. Die TOOLKIT–Wortgruppe

Die Toolkit–Wörter sind für bigFORTH–Anwender sicher alte Bekannte:

.S ? DUMP SEE WORDS

Die TOOLKIT–Extension–Wortgruppe

Auch die Toolkit–Extensions–Wörter sind weitgehend bekannt. Die ausführbaren Konditionals hießen in bigFORTH 1.10 noch #IF ELSE# und THEN# und konnten nur innerhalb einer Zeile verwendet werden, und auch dort nur, wenn kein # vorkam.

;CODE AHEAD ASSEMBLER BYE CODE CS–PICK CS–ROLL EDITOR FORGET STATE [ELSE] [IF] [THEN]

AHEAD (−−) immediate restrict: Compiliert einen BRANCH hinter das nächste THEN. Aus AHEADs kann man Kontrollstrukturen wie ELSE aufbauen; AHEAD wurde in ANS–FORTH eingeführt, damit man auf Wörter wie BRANCH und ?BRANCH verzichten kann, und trotzdem Kontrollstrukturen aufbauen kann. Da das gelungen ist,

verzichtet auch bigFORTH auf BRANCH und ?BRANCH (sie sind nicht mehr sichtbar).

CS–PICK (c1 .. cn n – c1 .. cn c1) immediate restrict: Kopiert das **n**–te Kontrollwort aus dem Kontroll–Stack (der Stack während der Compilezeit) an den Top of Stack. In bigFORTH ist das die Vorwärts– oder Rückwärtsreferenz, die von IF, AHEAD, BEGIN, DO, ?DO oder FOR auf den Stack gelegt wurde.

CS–ROLL (c1 c2 .. cn n –– c2 .. cn c1) immediate restrict: Holt das **n**–te Kontrollwort aus dem Kontroll–Stack nach oben. Beispiel für die Verwendung:

```
: ELSE  POSTPONE AHEAD  1 cs-roll  POSTPONE THEN ;  immediate restrict
```

EDITOR (––) (VS voc –– EDITOR): Editor–Vokabular. In bigFORTH enthält das Editor–Vokabular kaum mehr als die Aufruf–Kommandos des Editors, die auch im FORTH–Vokabular schon vorhanden sind.

[IF] (flag ––) immediate: Wenn **flag** false ist, wird nach dem nächsten [ELSE] oder [THEN] im Input–Stream gesucht, das in derselben Schachtelungstiefe von [IF]..[ELSE]..[THEN]s ist. Achtung! [IF] und [ELSE] verändern lediglich den Parser, verlangen also, daß der Inputstrom weiterhin von INTERPRET oder etwas analogem geparst wird.

[ELSE] (––) immediate: Sucht nach dem nächsten [THEN] derselben Schachtelungstiefe im Inputstrom

[THEN] (––) immedate: Tut nichts, stellt nur das Ziel der Suche von [IF] und [ELSE] dar

2.11. Die SEARCH–Wortgruppe

Die Suchordnung mittels Vokabularen war im FORTH83–Standard noch nicht festgelegt, allerdings war in einem Anhang das von Perry & Laxen verwendete ONLY–ALSO–Konzept bereits erwähnt. ANS–FORTH greift darauf zurück, allderdings werden andere Primitives eingeführt, die den Vokabularstack direkt manipulieren und deshalb ggf. besser geeignet sind, ungewöhnliche Konzepte zu implementieren.

DEFINITIONS FIND FORTH–WORDLIST GET–CURRENT GET–ORDER SEARCH–WORDLIST SET–CURRENT SET–ORDER WORDLIST

FORTH–WORDLIST (–– wid): Gibt die Adresse des Vokabulars **wid** zurück, indem sich die vom System bereitgestellten Wörter befinden (Vokabular FORTH)

GET–CURRENT (–– wid): Entspricht CURRENT @. Gibt die Adresse des Vokabulars **wid** zurück, in das gerade definiert wird.

SET–CURRENT (wid ––): Entspricht CURRENT !. Setzt die Adresse des Vokabulars **wid** zurück, in dem danach neue Definitionen eingetragen werden.

GET–ORDER (–– wid1 .. widn n) (VS wid1 .. widn –– wid1 .. widn): Liest den Vokabularstack aus. **widn** ist dabei das Vokabular, das zuerst durchsucht wird, **wid1** das, das zuletzt durchsucht wird. Der Vokabularstack selbst wird nicht verändert.

SET–ORDER (wid1 .. widn n ––) (VS ⟨*any*⟩ –– wid1 .. widn): Setzt den Vokabularstack neu, die Parameter entsprechen GET–ORDER. Die vorherige Suchreihenfolge wird verworfen. −1 SET–ORDER entspricht ONLY.

WORDLIST (–– wid): Erzeugt ein neues (allerdings namenloses) Vokabular **wid**. Dieses Vokabular kann nur mit SET–ORDER oder SET–CURRENT in die Suchordnung aufgenommen werden.

SEARCH–WORDLIST (addr u wid –– cfa state / f): Durchsucht das Vokabular **wid** nach dem Wort **addr u**. Es gibt bei Erfolg die **cfa** und den Immediate–Status **state** zurück, sonst false. Achtung! Wörter, die mit RESTRICT markiert werden, geben 2 oder −2 zurück; diese Zahlen sind vom Standard nicht definiert.

Die SEARCH–Extension–Wortgruppe

ALSO FORTH ONLY ORDER PREVIOUS

PREVIOUS (––) (VS wid ––): Alias für TOSS. Nimmt das oberste Vokabular aus der Suchordnung.

2.12. Die STRING–Wortgruppe

Die String–Befehle entsprechen nicht immer den gleichnamigen Befehlen in älteren Versionen von bigFORTH: COMPARE und SEARCH nehmen andere Argumente. Beide sind jedoch CAPS–sensitiv geblieben. Wenn also CAPS ON ist, wird auf Groß– und Kleinschreibung keine Rücksicht genommen. Die Defaulteinstellung (CAPS OFF) ist daher die, in der sich das System ANSI–konform verhält.

–TRAILING /STRING BLANK CMOVE CMOVE> COMPARE SEARCH SLITERAL

COMPARE (addr1 u1 addr2 u2 –– n): Vergleicht die Strings **addr1 u1** und **addr2 u2**. **n** ist 1, wenn **addr1 u1** entsprechend der lexikographischen Ordnung größer als **addr2 u2** ist, −1, wenn es kleiner ist, und 0, wenn beide Strings gleich sind. Ein längerer String, dessen Prefix der kürzere ist, ist per Definition größer, ansonsten bestimmt die Differenz der ersten nicht gleichen Zeichen den Unterschied.

SEARCH (addr0 u0 addr1 u1 –– addr0' u0' flag): Sucht im String **addr0 u0** nach dem Substring **addr1 u1**. Wenn er gefunden wurde, ist **flag** true, und **addr0' u0'** geben die Position der ersten Fundstelle bis zum Ende des Strings an. Wird er nicht gefunden, ist **flag** false, und **addr0' u0'** ist **addr0 u0**.

SLITERAL (addr u ––) immediate: Compiliert einen String als Stringliteral. Bei der Ausführung wird der String wieder in der Form **addr' u** ausgegeben, wobei **addr'** nun die entsprechende Stelle im Code bezeichnet.

3. Documentation Requirements

Der ANSI–Standard legt einige Anforderungen an die Dokumentation fest, also "Documentation requirements"; das sind Systemeigenschaften, die ein Standard–System dokumentieren muß. Alle zu dokumentierenden Eigenschaften werden hier dokumentiert, unabhängig davon, ob sie an anderer Stelle ebenfalls dokumentiert sind, um den Anforderungen des Standards optimal zu genügen. Da der Standard ein amerikanischer ist, sind die Überschriften entsprechend den Document requirements auch in Englisch gehalten.

Nach besten Wissen und Gewissen ist bigFORTH ein ANS Forth System, welches

- die Core–Extensions–Wortgruppe
- die Block–Wortgruppe
- die Block–Extensions–Wortgruppe
- die Double–Number–Wortgruppe
- die Double–Number–Extensions–Wortgruppe
- die Exception–Wortgruppe
- die Facility–Wortgruppe
- die Facility–Extensions–Wortgruppe
- die File–Access–Wortgruppe
- die File–Access–Extensions–Wortgruppe
- die Floating–Point–Wortgruppe
- die Floating–Point–Extensions–Wortgruppe
- die Locals–Wortgruppe
- die Locals–Extensions–Wortgruppe
- die Memory–Allocation–Wortgruppe
- die Memory–Allocation–Extensions–Wortgruppe
- die Programming–Tools–Wortgruppe
- die Programming–Tools–Extensions–Wortgruppe
- die Search–Order–Wortgruppe
- die Search–Order–Extensions–Wortgruppe
- die String–Wortgruppe
- die String–Extensions–Wortgruppe

bereitstellt.

Der Standard verlangt die Dokumentation weiterer implementationsabhängige Fakten. Dieser Anforderung sollen die weiteren Abschnitte genügen. An vielen Stellen wird nicht die Information selbst gegeben, sondern eine Methode, sie direkt am System zu gewinnen, insbesondere, wenn die Information vom Prozessor oder Betriebssystem abhängt, oder wenn sie sich während der Entwicklung von bigFORTH verändern.

3.1. Die Core–Wörter

Implementation–defined options

Adressen–Alignment Prozessorabhängig. bigFORTH's Alignment–Wörter sorgen lediglich für Erfüllung der schwächsten Alignment–Bedingung, also 2 Bytes auf 68k, keine spezielle Ausrichtung auf i386.

Verhalten von EMIT für nicht–druckbare Zeichen Abhängig vom Ausgabevektor. Auf dem Bildschirm wird versucht, alle Zeichen als grafische Zeichen zu behandeln, soweit das vom Betriebssystem zugelassen wird. Verwendet bigFORTH ein Fenstersystem, werden alle Zeichen als grafisches Zeichen ausgegeben.

Kommandozeileneditor für ACCEPT und EXPECT Der Kommandozeileneditor lehnt sich an den GNU–Readline an mit Emacs–artigen Key–Bindings. Die History weicht insofern ab, als daß ein Ringpuffer verwendet wird, der mit jedem ⟨RET⟩ um eine Position fortgeschaltet wird, und ⟨⇧⟩ sowie ⟨⇩⟩ zurück- bzw. vorblättern. ⟨Tab⟩ weicht insofern ab, als daß bei jedem Tastendruck ein neues Wort erzeugt wird (der Länge und alphabetisch sortiert).

Zeichensatz Es wird der Zeichensatz des jeweiligen Displays verwendet. bigFORTH selbst ist 8–bit–clean, aber manche Geräte/Zeichensätze können hier Probleme machen. In manchen Fenstersystemen kann bigFORTH die Auswahl des Zeichensatzes selbst möglich machen.

Zeichenadressen–Alignment bigFORTH verwendet Bytes als Zeichen und verlangt deshalb auf den bisher unterstützen Prozessoren keine besonderen Alignment–Bedingungen. Das kann sich ggf. unter einem Unicode–Betriebssystem ändern.

Zeichensatzerweiterungen Jedes Zeichen kann im Namen eines Wortes verwendet werden (Zeichen $\leq$ bl werden allerdings beim Parsen ausgeschlossen). Der Vergleich ist Case–insensitiv. Die Umwandlung erfolgt durch bigFORTH's Wörter CAPITAL und TOLOWER, die eine Tabelle verwenden. Diese Tabelle ist den Standardzeichensätzen des jeweiligen Betriebssystems angepaßt.

Bedingungen, unter denen ein Kontrollzeichen als Space–Delimiter erkannt wird Wird WORD mit BL als Delimiter aufgerufen, werden alle Zeichen $\leq$ BL als Space interpretiert. PARSE behandelt BL wie jeden anderen Delimiter. Feste Spaces, wie $FF im IBM–Zeichensatz werden nicht als Space–Delimiter erkannt.

Format des Control–Flow–Stack Der Datenstack wird als Control–Flow–Stack verwendet. Alle Elemente sind eine Zelle groß. Diese Zelle ist eine Adresse im Code; bei unaufgelösten Vorwärtssprüngen die Adresse des Sprungoffsets.

Umwandlung von Ziffern größer als 35 Bei der Ausgabe werden solche Ziffern entsprechend den Zeichen nach Z ausgegeben. Bei der Eingabe werden nur die ersten Zeichen richtig erkannt, also [, \,], ^, _, ‘, da die Kleinbuchstaben in Großbuchstaben gewandelt werden. Es gibt deshalb keine Möglichkeit, viele der größeren Ziffern einzugeben.

Display nach Eingabeende von ACCEPT und EXPECT Der Cursor wird hinter das Ende des Eingabestrings bewegt, es wird ein Space ausgegeben.

Exception–Abort–Sequenz von ABORT" Der Fehlerstring wird in "ERROR gespeichert und ein `-2 throw` wird ausgeführt.

Eingabezeilenende Bei interaktiver Eingabe wird ⟨Ctrl⟩⟨M⟩ und ⟨Ctrl⟩⟨J⟩ als Zeilenende interpretiert. Eines von diesen Zeichen wird üblicherweise von ⟨RET⟩ oder ⟨Enter⟩ erzeugt.

Maximale Größe eines counted Strings `s /counted-string environment? drop .`, normalerweise 255 Zeichen.

Maximale Länge eines geparsten Strings Abhängig von der Eingabe. Maximale Länge einer Eingabezeile ist `max#tib @ .`; aus Streamfiles werden maximal 255 Zeichen pro Zeile gelesen. Ist der Input ein Block, so wird ein Kilobyte geparst. EVALUATE kann beliebig lange Strings (im Rahmen der Hauptspeichergröße) verdauen, kopiert diese Strings allerdings (es muß also mindestens soviel freier Speicher vorhanden sein, wie der String belegt).

Maximale Größe eines Definitionsnamen in Zeichen 31

Maximale Stringlänge für ENVIRONMENT? 31

Methode zur Auswahl des Usereingabegeräts Das Eingabegerät wird über den Eingabevektor INPUT ausgewählt. Jede mit INPUT: definierte Tabelle verändert beim Aufruf den Eingabevektor.

Methode zur Auswahl des Userausgabegeräts Das Ausgabegerät wird über den Ausgabevektor OUTPUT ausgewählt. Jede mit OUTPUT: definierte Tabelle verändert beim Aufruf den Ausgabevektor.

Wörterbuchcompilations–Methoden Jedes Wort besteht aus einem Header (der ggf. nach einem SAVE verschwinden kann, wenn das Wort headerless compiliert wurde) und einem Body, der bei ALIAS–Wörtern leer ist. Das Dictionary besteht aus zwei Teilen, dem aktuellen Modul, in die Bodies und Header von nichtheaderlosen Wörtern compiliert werden, sowie dem Headersegment, das zwischen Stack und Userarea des compilierenden Tasks liegt; hier werden Header von headerlosen Wörtern und die Bodies von mit HMACRO markierten Wörtern compiliert.

Ein Header sieht folgendermaßen aus:

- 2 Bytes View–Feld, die untersten 10 Bit beschreiben die Zeilen- oder Blocknummer, die obersten 6 Bits die Dateinummer.
- Eine Zelle Link–Feld, welches auf das Link–Feld des vorherigen Wortes im Vokabular oder auf 0 (erstes Wort im Vokabular) zeigt.
- Countbyte mit Informationen über immediate ($40), restrict ($80) und Alias ($20).
- ⟨***Count***⟩ Zeichen
- mögliches Alignment, aufgefüllt wird mit BL
- Bei ALIAS–Wörtern: Zeiger auf den Code, sonst folgt nun der Body.

Ein Body sieht folgendermaßen aus:

- 2 Bytes Länge und Makroinformation. Bei bigFORTH 68k wird ein Makro durch ein gesetztes LSB gekennzeichnet, bei bigFORTH 386 durch ein negatives Längenfeld
- Entsprechend der Länge Code. Bei mit CREATE erzeugten Wörtern ist die Länge 0, der Sprungcode wird nicht mitgezählt. Er ist bei bigFORTH 68k 6, bei bigFORTH 386 5 Bytes lang.
- Ggf. 2 Bytes Makroinfo, die aus je einem Byte Push und einem Byte Take–Byte besteht. Die Makroinfo kann sich bei zukünftigen bigFORTH–Versionen ändern.
- Bei mit CREATE erzeugten Wörtern folgt nun das Datenfeld.

Anzahl Bits in einer Adreß–Einheit 8

Zahlendarstellung und Arithmetik Zweierkomplement

Bereiche für Ganzzahlen Entsprechend der Environment–Queries für MAX–N, MAX–U, MAX–D und MAX–UD. bigFORTH ist in allen aktuellen Implementierungen ein 32–Bit–System.

Datenregionen, die nur lesbar sind Der ganze FORTH–Datenraum ist beschreibbar.

Puffergröße von WORD `pad here - .`, aktuell 102. Der Puffer wird mit dem Zahlenausgabepuffer geteilt. Wenn PAD überschrieben werden darf, kann das ganze restliche Dictionary als Puffer verwendet werden.

Größe einer Zelle in Adreß–Einheiten `1 cells`, aktuell immer 4

Größe eines Zeichens in Adreß–Einheiten `1 chars`, aktuell immer 1

Größe des Tastatureingabepuffers Variabel: `max#tib @ .` Normal 255.

Größe des Zahlenausgabenpuffers `pad here - .`, aktuell 102. Der Puffer wird mit dem von WORD geteilt.

Größe der von PAD zurückgegebenen Scratch–Area `unused .`, der Rest des aktuellen Dictionaries.

Case–sensivity–Charakteristik des Systems Die Wörterbuchsuche ist Case–insensitiv, ebenfalls die Wandlung von Ziffern größer 9. Die Wandlung von Base–Char–Ziffern ist allerdings Case–sensitiv, also `'a` gibt $61, `'A` dagegen $41. Die Case–Wandlung von nicht–ASCII–Zeichen hängt von der jeweiligen Konversions–Tabelle ab und ist damit systemspezifisch.

System–Prompt ‘ ok’ im Interpretermodus, ‘ compiled’ im Compilationsmodus. Spezielle Modi wie Kommentare über mehrere Zeilen können eigene Prompts, etwa ‘ skipped’ haben.

Runden bei der Division Es wird floored (gegen $-\infty$) gerundet.

Werte von STATE, wenn true -1

Rückgabewerte bei arithmetischem Überlauf Arithmetik wird modulo 2^n durchgeführt (n = bits/cell bzw. $n = 2 * \text{bits/cel}$ bei doppelt genauer Arithmetik), mit entsprechender Abbildung für vorzeichenbehaftete Typen. Bei Division durch 0 wird `-10 throw` ausgeführt, bei Überlauf evtl. `-11 throw`.

Ob die aktuelle Definition nach DOES> gefunden werden kann Nein

Ambiguous Conditions

Der Standard läßt einige Bedingungen offen, und verlangt lediglich von der Implementierung, daß diese „mehrdeutigen Bedingungen“ beschrieben werden.

Die folgenden Bedingungen können aufgrund einer Kombination von Faktoren auftreten:

Ein *Name* ist weder ein Wort noch eine Zahl `-13 throw` (Wort nicht definiert)

Der Name einer Definition überschreitet die maximale Länge `-19 throw` (Wortname zu lang)

Adressierung einer Region außerhalb des Data Spaces Alles innerhalb des Heaps ist adressierbar. Der Code des Loaders sowie dessen Stack sind ebenfalls adressierbar, zumindest lassen sie sich lesen. Andere Regionen hängen vom Betriebssystem ab. Unter OS/2 und Linux sind die Bereiche der Shared Libraries adressierbar. Unter DOS/GO32 sind alle von GO32 gemappten Bereiche adressierbar (siehe Seite **??**).

Argumenttyp mit spezifiziertem Inputparamter inkompatibel Dies wird üblicherweise nicht überprüft.

Versuch, das Execution Token einer Definition ohne Interpretationssemaktik zu bekommen (z. B. mit ' oder FIND) Es wird ersatzweise das Execution Token für die Compilationssemantik zurückgegeben. FIND gibt 2 oder −2 als Immediate–State zurück.

Division durch 0 `-10 throw` (Division durch 0)

Zu wenig Datenstack oder Returnstack (Stacküberlauf) Wird normalerweise nicht überprüft. Im äußeren Interpreter wird zwischen jedem Wort mit ?STACK überprüft, ob der Datenstack oder das Dictionary voll sind. Ist der Datenstack voll, wird `-3 throw` ausgeführt. Returnstacküberlauf läuft in die Userarea hinein und zerstört diese. Stacküberlauf läuft aus dem aktuellen Stackbereich heraus und zerstört die Informationen der Speicherverwaltung, sowie evtl. davorliegende Datenbereiche.

Zu wenig Platz für Schleifenkontrollparameter Wird nicht überprüft. Führt effektiv zum Returnstacküberlauf.

Zu wenig Platz im Dictionary Wird nur im äußeren Interpreter überprüft. Führt dort zu `-8 throw` (Dictionary voll), zuvor wird das zuletzt erzeugte Wort vergessen.

Interpretation eines Wortes mit undefinierter Interpretationssemantik Normalerweise `-14 throw`, wurde das Execution–Token mit ' oder FIND geholt, wird die Compilationssemantik ausgeführt.

Modifikation des Inhalts eines Inputpuffers oder eines Stringliterals Da sich diese im beschreibbaren Speicher befinden, können sie modifiziert werden

Überlauf bei der Zahlenausgabe Wird nicht überprüft. Die letzten Einträge des Dictionaries werden überschrieben.

Überlauf eines geparsten Strings PARSE läuft nicht über

Erzeugen eines Resultats außerhalb des Ergebnisbereichs Siehe „Rückgabewerte bei arithmetischem Überlauf“ CONVERT und >NUMBER laufen ohne Meldung über.

Lesen von einem leeren Daten- oder Returnstack (Stackunterlauf) Wird nur vom Interpreter gecheckt. Dort wird beim Datenstacküberlauf `-4 throw` ausgeführt. Rücksprung bei leeren Returnstack beendet die aktuelle Task.

Unerwartetes Ende des Eingabepuffers, welches im Lesen eines leeren Wortes resultiert Alle Wörter, die HEADER verwenden: `-16 throw`. Ansonsten wird das nicht speziell überprüft.

Die folgenden Bedingungen sind im Standard in den Glossary–Einträgen zu den relevanten Wörtern notiert:

>IN größer als Input–Puffer Der nächste Aufruf eines parsenden Wortes gibt einen String der Länge 0 zurück

RECURSE nach DOES> Compiliert das Wort, in dem DOES> vorkommt, nicht das definierte Wort

Das Input–Source–Argument für RESTORE–INPUT ist vom aktuellen Input–Source verschieden Ist es ein gültiger Input–Source, so wird er wiederhergestellt

Data–Space, der Definitionen enthält, wird deallokiert Deallokieren mit ALLOT wird nicht überprüft. Die Definitionen werden ggf. später überschrieben; Zugriffe können deshalb Speicherzugriffsfehler ergeben.

Data–Space–Lesen/Schreiben mit inkorrektem Alignment evtl. `-23 throw`

Data–Space–Zeiger (DP) nicht korregt aligned (, oder C,) Analog, je nach Prozessor: `-23 throw`

Weniger als $u+2$ Stackelemente (PICK, ROLL) Wird nicht überprüft. ROLL kann den Wordheaderheap und die Userarea zerstören. PICK kann ggf. einen Speicherzugriffsfehler auslösen, wenn das Stackelement außerhalb des adressierbaren Bereichs läge.

Schleifenkontroll–Parameter nicht vorhanden Wird nicht überprüft. Es werden die aktuellen Werte im Schleifenregister bzw. auf dem Returnstack verwendet.

Die letzte Definition hat keinen *Namen* (IMMEDIATE) Es wird die letzte Definition mit Namen verändert

Name* ist nicht von VALUE definiert, wird aber mit TO benutzt** Wird nicht überprüft. Der Wert wird in ' ⟨Name***⟩ >BODY gespeichert.

***Name* nicht gefunden (', POSTPONE, ['], [COMPILE]) `-13 throw`**

Parameter nicht vom selben Typ (DO, ?DO, WITHIN) Wird nicht überprüft. Diese Wörter verhalten sich so, als wären die Parameter vom selben Typ (z. B. Integer).

POSTPONE oder [COMPILE] wird auf TO angewendet `: X POSTPONE TO ; immediate` ist equivalent zu TO

String länger als ein counted String mit WORD Wird nicht überprüft. Der String ist ok, überschreibt aber PAD. Die Länge wird modulo 256 gespeichert.

u **größer oder gleich der Anzahl Bits einer Zelle (LSHIFT, RSHIFT)** Prozessorabhängig. In der Regel werden nur die letzten *n* Bits von *u* interpretiert, oder es wird 0 zurückgegeben.

Wort nicht mit CREATE definiert (>BODY, DOES>) DOES>: Es wird ohne Rücksicht auf Verluste die Sprungadresse nachgetragen, das Wort wird zerstört. >BODY: Versucht, soweit möglich, etwas equivalentes zu berechnen, ansonsten wird der Execution Token unverändert zurückgegeben. >BODY verhält sich korrekt bei mit USER, DEFER und PATCH definierten Wörtern, nicht jedoch bei Konstanten und mit : definierten Wörtern.

Wort außerhalb von <# und #> verwendet (#, #S, HOLD, SIGN) Wird nicht überprüft, es können Speicherzugriffsfehler auftreten.

Andere Systemdokumentation

Nichtstandard–Wörter, die PAD verwenden Keine

"operator's terminal facility available" Ein Terminal für den Operator ist vorhanden.

Vorhandener Programm–Daten–Space, in Adreßeinheiten `unused .`, durch Anlegen eines neuen Moduls erneut 64k, nur durch den freien Speicher `memory freemem . forth` limitiert.

Vorhandener Returnstack–Space, in Zellen `rp@ up@ udp @ + - cell/ .`

Vorhandener Stack–Space in Zellen `sp@ ^s @ - cell/ .`

Platz, den das Systemwörterbuch verbraucht, in Adreßeinheiten : Schwierig zu berechnen. `modules` listet alle Module auf, sowie deren Platzbedarf.

3.2. Die Block–Wörter

Implementation–defined options

Das von LIST verwendete Format Zuerst wird die Datei und die Screennummer ausgegeben, dann 16 jeweils 64 Zeichen lange Zeilen; am Zeilenanfang steht jeweils die Zeilennummer

Die Länge der von \ betroffenen Zeile 64 Zeichen

Ambiguous Conditions

Korrektes Lesen des Blocks war nicht möglich Resultiert üblicherweise in einem `-33 throw`, davor wird die Fehlernummer des Betriebssystems ausgegeben (evtl. in einen String gewandelt)

I/O–Exception bei der Block–Transfer Analog `-33 throw` beim Lesen, `-34 throw` beim Schreiben und vorherige Ausgabe des Betriebssystemfehlers

Ungültige Blocknummer Analog `-33 throw` (`-34 throw` beim Schreiben), es wird vorher evtl. ausgegeben, daß das ein ungültiger Block war. Es gibt keine ungültigen Nummern für BUFFER.

Ein Programm ändert BLK direkt Es wird ab sofort der in BLK gespeicherte Block an der aktuellen Position (>IN) interpretiert

Kein aktueller Block für UPDATE UPDATE hat keinen Effekt

Andere Systemdokumentation

Restriktionen, die ein Multiprogramming–System an die Benutzung von Pufferadressen stellt Die Inputpufferadressen können sich nach jedem Aufruf von PAUSE, von I/O–Operationen, die implizit PAUSE aufrufen, von Warte–Operationen wie WAIT oder TILL und bei Speicheranforderungen über das Memory Management ändern oder ungültig werden

Anzahl verfügbarer Blöcke für Text und Daten Abhängig vom Platz auf der Disk. Es können aktuell pro Datei höchstens 4 Gigabytes genutzt werden. DF zeigt den freien Plattenplatz an.

3.3. Die Double–Number–Wörter

Ambiguous Conditions

d **außerhalb des Wertebereichs von** n **in D>N** Die niederwertige Hälfte von d wird zurückgegeben

3.4. Die Exception–Wörter

Implementation–defined options

Im System von THROW und CATCH benutzte Werte -256 wird von ERROR“ verwendet und signalisiert, daß der Stack nicht bereinigt wurde[2]. Werte zwischen -257 und -512 werden für Signals verwendet, falls das Betriebssystem solche verwendet. Werte zwischen -513 und -1023 werden für Exceptions verwendet, die bigFORTH selbst definiert. Werte zwischen -1024 und -2048 werden für Fehlermeldungen des Betriebssystems verwendet.

[2]Das hat aber nur einen Effekt, wenn mit INTERPRET direkt gearbeitet wird. Alle Standard–Eingabemethoden bereinigen den Stack

3.5. Die Facility–Wörter

Implementation–defined options

Codierung von Keyboard–Events (EKEY) $s * 256 + c$ wobei s der Scan–Code der Taste ist, c das druckbare Zeichen. Mausklicks werden $8000 + x * 256 + y$ codiert.

Dauer des Systemclock–Ticks Systemabhängig. Die Zeit für MS wird in Millisekunden angegeben. bigFORTH 386/GO32 kann den Timer bis auf 20ns auflösen, bigFORTH ST auf 26ns.

Wiederholbarkeit, die von MS erwartet werden kann Systemabhängig, sowie abhängig von den anderen Tasks im FORTH–System. Unter GO32/DOS (keine DOS–Box!) sowie auf dem ST ohne MiNT sehr hoch, ansonsten abhängig von der Systemlast.

Ambiguous Conditions

AT–XY kann nicht auf dem Ausgabegerät ausgeführt werden Geräteabhängig. Mit +BUFFER kann auf jedem Gerät innerhalb einer Zeile positioniert werden, ansonsten passiert nichts.

3.6. Die File–Wörter

Implementation–defined options

Dateizugriffsmethode von BIN, CREATE–FILE, OPEN–FILE, R/O, R/W, W/O Im Moment immer read/write+binary.

Datei–Exceptions Die Dateiwörter lösen keine Exceptions aus, höchstens Speicherzugriffsfehler

Zeilenende in Dateien (READ–LINE) Systemabhängig. Ein LF reicht aus, auf CR/-LF–Systemen wird das CR, wenn vorhanden, weggeschnitten.

Dateinamenformat Systemabhängig. bigFORTH nutzt das Dateinamenformat des Betriebssystems, evtl. werden '/' in '\' gewandelt.

Von FILE–STATUS zurückgegeben Information Systemabhängig. Auf 386/GO32 und ST wird die Disk Transfer Area (DTA) nach erfolgreicher Suche nach der Datei zurückgegeben.

Inputdatei–Status nach einer Exception Alle Dateien, die durch die Exception verlassen werden, werden geschlossen. Sind sie auch von anderen Tasks geöffnet, bleiben sie offen, die Zahl der notwendigen CLOSEs wird um eins reduziert.

Werte und Bedeutung von *ior* ior ist die Fehlernummer des Betriebssystems.

Maximale Tiefe verschachtelter Inputdateien Nur abhängig von der maximalen Anzahl geöffneter Dateien

Maximale Länge der Eingabezeile `cmax` . 256 Zeichen.

Methoden, nach denen Block–Bereiche auf Dateien zugeordnet werden Es wird auf die Datei zugegriffen, die in der Uservariable ISFILE steht. Die Blocknummern sind 0 bis $n - 1$, mit n angefangenen Kilobytes in der Datei.

Anzahl Stringbuffer für S“ Einer. S“ gibt nur den Teil des Eingabepuffers zurück, in dem der String steht.

Größe des von S“ genutzten Puffers Höchstens so groß wie die/der aktuelle Eingabezeile/block

Ambiguous Conditions

Versuch, außerhalb der Grenzen einer Datei zu positionieren Abhängig vom Betriebssystem. Im Fehlerfall wird die Fehlernummer des Betriebssystems zurückgegeben.

Versuch, von einer Dateiposition zu lesen, an der noch nichts geschrieben wurde Dateiende, es wird kein Zeichen gelesen und bei READ–LINE wird Dateiende signalisiert. Es wird kein Fehler gemeldet.

***fileid* ist nicht gültig** Fehlermeldungen des Systems oder ein Speicherzugriffsfehler sind möglich

I/O–Exception beim Lesen oder Schließen (INCLUDE–FILE, INCLUDED) Der *ior* der Operation wird mit THROW weitergegeben.

Eine benannte Datei kann nicht geöffnet werden (INCLUDED) Der von OPEN–FILE zurückgegebene *ior* wird mit THROW weitergegeben

Anforderung einer nicht zugeordneten Blocknummer `-5 throw`

Verwendung von SOURCE–ID, wenn BLK nicht 0 ist Gibt die gerade benutzte Block–Datei, von der der geladene Block stammt, zurück

3.7. Die Fließkomma–Wörter

Implementation–defined options

Format und Wertebereich von Fließkommazahlen Systemabhängig; teilweise vom Benutzer wählbar. bigFORTH 68k verwendet ein eigenes Format (siehe hierzu Kapitel 10.7.1), bigFORTH 386 verwendet das IEEE–Extended–Format und nutzt den Koprozessor für die Fließkommarechnung.

Resultat von REPRESENT, wenn *float* außerhalb des Wertebereichs ist Implementationsabhängig. REPRESENT führt keine Checks in dieser Hinsicht durch; es hängt auch davon ab, ob solche Zahlen überhaupt berechnet werden können; oder ob bereits bei der Berechnung eine Exception auftritt. bigFORTH 68k läßt keine Resultate außerhalb des Wertebereichs zu, bigFORTH 387 rechnet mit den Zahlen weiter, die der 387 liefert; entsprechend werden auch Überläufe behandelt.

Runden oder Abschneiden von Fließkommazahlen Es wird, wenn nötig, ein Round to nearest (even) durchgeführt

Größe des Fließkommastacks Bei bigFORTH 68k entsprechend dem mit FSTACK oder FHSTACK angelegten Bereich, bei bigFORTH 386 8 Elemente.

Breite des Fließkommastacks Jedes Element des Fließkommastacks ist `1 FLOATS` breit

Ambiguous Conditions

DF@ oder DF! mit nicht Double–Float–alignten Adresse Wie andere Alignment–Fehler

F@ oder F! mit nicht–Float–alignten Adresse Ditto

Fließkommaresulat außerhalb des Wertebereichs Implementations–abhängig. Es wird, wenn möglich, eine entsprechende Exeption gethrowt. Fehlermeldungen können, etwa beim 387, auch verzögert auftreten.

SF@ oder SF! mit nicht–Single–Float–alignten Adresse Wie andere Alignment–Fehler

BASE ist nicht dezimal (REPRESENT, F., FE., FS.) Die Zahl wird entsprechend der aktuellen Zahlenbasis gewandelt

Beide Argumente 0 (FATAN2) $\pi/2$

FTAN von einem Argument r **mit** $\cos(r) = 0$ Entsprechend Division durch 0. Wegen kleiner Fehler sind eher sehr große Werte zu erwarten.

d **kann nicht exakt als Fließkommazahl repräsentiert werden (D>F)** Es wird zur nächsten Fließkommazahl gerundet

Divison durch 0 (F/) `-42 throw`, wenn die Exception nicht maskiert ist (387)

Exponent zu groß für Konvertierung (DF!, DF@, SF!, SF@) Wie Überlauf. Die Emulation für bigFORTH 68k überprüft das nicht und speichert den niederwertigen Teil des Exponenten. Kann nur bei DF! und SF! vorkommen.

float < 1 **bei FACOSH** `-46 throw`

float ≤ -1 **bei FLNP1** `-46 throw`. Bei $= -1$ wird evtl. $-\infty$ zurückgegeben (387).

float < 0 **bei FASINH, FSQRT** `-46 throw`

float > 1 **bei FACOS, FASIN, FATANH** `-46 throw`

Integerteil von float kann nicht in d **dargestellt werde (F>D)** `-43 throw`

String größer als Zahlenausgabenarea (F. FE. FS.) Kann nicht vorkommen

3.8. Die Locals–Wörter

Implementation–defined options

Maximale Anzahl Locals während einer Definition Beliebig. Die Anzahl hängt vom Platz auf dem Wortheaderheap ab.

Ambiguous Conditions

Ausführung einer benannten lokalen Variable im Interpretermodus `-14 throw` (compile only)

Name nicht von VALUE oder LOCAL definiert (TO) Entsprechend dem normalen Verhalten von TO.

3.9. Die Memory–Wörter

Implementation–defined options

Werte und Bedeutung von *ior* Throw–Codes für Speicherfehler.

−$201 Kein Speicher mehr frei

−$202 Keine gültige Adresse

3.10. Die Programming–Tools–Wörter

Implementation–defined options

Ende–Sequenz für Eingaben nach ;CODE und CODE `Next end-code`

Art und Weise der Eingabeverarbeitung nach ;CODE und CODE Es wird auf das Assembler–Vokabular geschaltet und weiter interpretiert.

Search–Order–Fähigkeiten für EDITOR und ASSEMBLER Diese verwenden die normale Suchreihenfolge entsprechend dem Search–Order–Wortschatz.

Quelle und Anzeigeformat von SEE Die Quelle ist der compilierte und optimierte Code des Wortes. Es wird versucht, soweit möglich, FORTH–Sourcecode auszugeben. Nicht ausgebbare Bereiche werden als Disassemblerlisting im Standardformat des jeweiligen Prozessors ausgegeben.

Ambiguous Conditions

Löschen der Compilations–Wörterliste FORTH wird aktuelle Compilations–Wörterliste

Weniger als $u+1$ Elemente auf dem Kontrollfluß–Stack (CS–ROLL, CS–PICK) Wird nicht überprüft, entsprechend PICK und ROLL

***name* kann nicht gefunden werden (FORGET)** Ausgabe von “can’t forget” und **name**, kein Abort

***Name* nicht von CREATE erzeugt (;CODE)** Wie bei DOES>

POSTPONE auf [IF] angewendet `: X POSTPONE [IF] ; immediate` ist äquivalent zu [IF], mit einer Ausnahme: In Verschachtelungen wird X nicht als [IF] erkannt.

Erreichen des Endes des Eingabestroms bevor [ELSE] oder [THEN] gefunden wurden ([IF]) Es wird im nächsten Eingabestrom weitergesucht.

Löschen einer benötigten Definition (FORGET) `-15 throw`

3.11. Die Search–Order–Wörter

Implementation–defined options

Maximale Anzahl Wörterlisten in der Suchreihenfolge 16

Minimale Suchreihenfolge (ONLY) `ROOT ROOT`

Ambiguous Conditions

Ändern der Compilations–Wörterliste beim Compilieren Das Wort wird in die Wörterliste aufgenommen, die Compilations–Wörterliste ist, während REVEAL aufgerufen wird (;, END–CODE)

Suchreihenfolge leer (PREVIOUS) Es geschieht nichts

Zu viele Wörterlisten in der Suchreihenfolge (ALSO) `-49 throw`

13 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered

independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and a brief idea of what it does.*
> Copyright (C) 19*yy name of author*
>
> This program is free software; you can redistribute it and/or modify
> it under the terms of the GNU General Public License as published by
> the Free Software Foundation; either version 2 of the License, or
> (at your option) any later version.
>
> This program is distributed in the hope that it will be useful,
> but WITHOUT ANY WARRANTY; without even the implied warranty of
> MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
> GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License
> along with this program; if not, write to the Free Software
> Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19*yy name of author*
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
> type 'show w'.
> This is free software, and you are welcome to redistribute it
> under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> *signature of Ty Coon*, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

14 GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.

- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant

Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ör any later versionäpplies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

15 Glossar

1. Index

2. Literaturverzeichnis

[1] Dick Pountain; "Object–Oriented Forth"; Academic Press 1987

[2] Intel; "i486 Microprocessor Programmer's Reference Manual"; Osborne McGraw–Hill 1990; ISBN 0–07–881674–2