bigFORTH+MINOΣ

Documentation

contains And so Forth...

Bernd Paysan J.L. Bezemer (And so Forth...)

Introduction

1. "Real Programmers don't Read Manuals"

Well, it should be more "Real Programmers don't Write Manuals", as there have been some complaints that the free version of bigFORTH doesn't contain a free manual — the only available manual so far was the German manual for the commercial version. Since bigFORTH is a highly complex system, a manual is necessary. However, to get started, you could as well just start bigFORTH and play around with it. If you then still have questions, the lecture of this manual should clairify the issues.

Note however that this is still work in process and the manual is neither complete nor accurate. If you feel able to contribute, do so!

We propose to read the documentation part completely and try the examples. If there are questions left, you can use index and reference part to learn more.

This manual assumes that you can use your operating system. The final version of the manual should contain a real Forth course and it should be possible to replace a Forth introduction with it; however this is far from complete now.

2. History

bigFORTH bases on a 32 bit port of the volksFORTH-83 authors. volksFORTH-83 is a public domain system that is available for Atari ST, IBM PC and C64 (there called UltraForth). It's a 16 bit implementation with the corresponding limitations like a 64 KByte large address space. volksFORTH bases to some extents on the Perry/Laxen-F83 from the Forth Interest Group.

The volksFORTH authors started at the end of 1987 to port the system to 32 bits, and convert the compiler so that it creates real 68k machine code. To some engaged volksFORTH users they gave running prototypes. The system then was called turboFORTH and should have been distributed commercially, but the project almost died.

However, one of these engaged volksFORTH users continued development and after two years of development, bigFORTH 1.0 for the Atari ST was released in summer 1990.

In 1992, at the German Forth-Tagung, Ewald Rieger and Friedl Amend asked me for a 386 version of bigFORTH, and Ewald Rieger sponsored the PC. 1995, this version (bigFORTH-DOS 1.20) was released, together with the last Atari ST version. The DOS version contained a object oriented widget toolkit which used VGA textmode to display.

To continue development of the 386 version, Ewald Rieger sponsored another PC, with the goal to have bigFORTH run on OS/2, Windows NT, or Linux. I first started porting to OS/2, and after Linux got shared libraries, I ported over to Linux.

In august 1996, after finishing my diploma thesis, I started on MINOΣ, the graphical user interface for bigFORTH, a sequel to the text mode UI of bigFORTH-DOS. On the Forth-Tagung 1997, I presented the first results and soon afterwards put a demo on my homepage. Later in 1997, the license has been decided, it's GPL (see page 249).

3. Returning Thanks

I want to thank the volksFORTH authors Dietrich Weineck, Georg Rehfeld and Klaus Schleisiek, since without their work and without their volksFORTH bigFORTH woudn't exist. Especially I'd like to thank Bernd Pennemann here, for his support cleared all open questions (including those of copyright), and whos recommendations allowed to complete the system. Furthermore he suggested (among a list of other suggestions) the name.

Further special thanks to Ewald Rieger, who supported the development over the time, and many thanks to the testers for their bug reports and suggestions. Thanks also to Nikolaus Heusler who lectured the original manual.

I'd also like to thank Charles H. Moore, for his wonderful Forth, Donald E. Knuth, for his TEX, Linus Torvalds for Linux, and Richard M. Stallman for the GNU public license and the GNU tools he started to develop.

4. Target Market

Forth has traditinally been much more a public domain language, that — unlike COBOL, Fortran, C or ADA — wasn't supported by neither the industry nor the military, and — unlike Pascal resp. Modula II — also wasn't developed on universities. Forth's development lies mostly in the hand of engaged small companies or users, who have some difficulties to match offerings of other languages.

bigFORTH should fill this gap and provide a modern development environment, that doesn't capitulate from large problems and create code that is in speed competitive to other high level languages.

5. Copyleft

München, in December 1999

Bernd Paysan

Contents

1 Installation

1. Installing on Linux

BigFORTH comes in several tarballs, of which only one — the source tarball — is absolutely necessary. The other parts are nice gimmicks, or in case of the documentation, something to read for you.

The package is split up into six parts. You need source and documentation for start, the other packets are either optional or for convenience. To unpack it, unpack all the files from the same directory. They all unpack into the subdirectory `bigforth`. cd there and type `make`, to create the rest. Type `make install` if you want a system-wide installation. You can use the configure script (`./configure --prefix=`⟨***your path***⟩) to let it install in another path. It's a good idea to copy the file `xbigforth.cnf` into your home directory, and modify the editor ID there. There are other options you can choose there, and the final version very likely will present you a dialog to edit those options comfortably.

Source `bigforth-`⟨***version***⟩`.tar.bz2`: This is sufficient for a minimal installation, and also for frequent updates. Just unpack it over your old MINOS source tree and type `make` to compile the new things.

Pattern `bigforth-pattern-`⟨***version***⟩`.tar.bz2`: Stylish pattern pixmaps. You won't need to download them every time, since they won't change often.

Wood style `bigforth-edata-wood-`⟨***version***⟩`.tar.bz2`: Icons and Povray sources for wood Enlightenment style. If you don't intent to use this style, you don't need to load it.

ShinyMetal style `bigforth-edata-ShinyMetal-`⟨***version***⟩`.tar.bz2`: Icons for ShinyMetal Enlightenment style. If you don't intent to use this style, you don't need to load it.

2. Installation on Windows

For those who don't have decided to switch to Linux yet, the windows version is distributed as auto-installing program, `bigforth-`⟨***version***⟩`.exe`. Includes everything you want on Windows. Install with a doubleclick. Uninstall before installing the next verion, please.

During installation, parts of the system are compiled; this saves download time. Furthermore, the configuration files, `bigforth.cnf` and `xbigforth.cnf` are updated. You can choose the paths you want to load bigFORTH sources from, and an editor ID there.

3. Editor Commands

There are several ways to get into the editor. The simplest one is `ed` ⟨***file***⟩ `[`⟨***screen/line***⟩ `[`⟨***cursor***⟩`]]`. You can decompose that with `use` ⟨***file***⟩, and open the editor at the current position with `v` or on a specific screen or line with ⟨***screen***⟩ `l`.

4. Notation von Befehlen

Befehle können in bigFORTH groß oder klein geschrieben werden, das System macht keinen Unterschied. Ebenso Dateinamen, wobei allerdings zu beachten ist, daß TOS die Umlaute (ä, ö und ü) nicht umwandelt und es daher einen Unterschied macht, ob man auf die Datei "Sätze" oder "SÄTZE" zugreift. Da manche Shells (z. B. die PD-Shell Guläm) keine Umlaute erlauben, ist es besser, bei Dateinamen darauf zu verzichten. Man kann sonst von diesen Shells aus nur eingeschränkt auf die Dateien zugreifen.

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. Starting the System

The main program you want to use is `xbigforth`, which contains the GUI library. `bigforth` uses the text-terminal as user interface, and `forthker` is the spartanic kernel.

bigFORTH consists of two parts, an executable loader, and an image file. The image file is by default the file with the same basename as the loader, and the suffix `.fi`. Typically, bigFORTH is invoked like this:

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

This interprets the contents of the files and the Forth code in the order they are given. I tried to make bigFORTH as much compatible with Gforth as reasonable.

In general, the command line looks like this:

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

The loader options must come before the rest of the command line. They are:

`--image-file` ⟨***file***⟩
`-i` ⟨***file***⟩ Loads the Forth image ⟨***file***⟩ instead of the default ⟨***loader***⟩`.fi`.

`--dictionary-size` ⟨***size***⟩
`-d` ⟨***size***⟩ Allocate ⟨***size***⟩ space for the Forth dictionary (all currently open modules) instead of using the default specified in the module (typically 256K). The ⟨***size***⟩ specification for this and subsequent options consists of an integer and a unit (e. g. `4M`). The unit can be one of `b` (bytes), `k` (kilobytes), `M` (Megabytes), and `G` (Gigabytes). If no unit is specified, `b` is used.

`--mem-size` ⟨***size***⟩
`-m` ⟨***size***⟩ Allocate ⟨***size***⟩ space to be used in the dynamic memory management (typically 16M).

`--stack-size` ⟨***size***⟩
`-s` ⟨***size***⟩ Allocate ⟨***size***⟩ space for both stacks together (typically 64k).

`--verbose`
`-v` Increment the level of verbosity of the loader.

After starting, bigFORTH displays a greeting message in the second line:

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

where x.xx is the version number, and “ANS” shows that it is an American National Standard Forth.

2. End of a Session

Type BYE⟨ret⟩ to exit.

3. File Suffixes

File Suffixes are generally only conventions. I use them as follows:

`.fs` Forth sources as stream, these text files can be edited with other editors, too.

`.fb` Forth sources as blocks, thus you can edit them only with the built-in editor (the Gforth-mode for Emacs understands blocks, too).

`.fi` Forth image. The loader loads them, SAVESYSTEM generates them.

`.m` MINOS Sources, for the GUI editor Theseus.

4. The Line Editor

- you can enter text in insert mode
- The cursor keys ⟨←⟩ and ⟨→⟩ move left and right through the text
- ⟨BS⟩ deletes the character left to the cursor, and moves the rest of the text left.
- ⟨DEL⟩ deletes the character under the cursor. The rest of the line moves one character left.
- ⟨↑⟩ moves backward in the command history, ⟨↓⟩ moves forward.
- Press ⟨RET⟩ to finish editing a line. The Forth interpreter then evaluates the line, and finally writes " `ok`" to signal that everything went ok, or " `compiling`" to indicate that the compiler is still turned on.
- The funcion keys are bound to some frequently used commands:

  ⟨F1⟩: .S
  ⟨F2⟩: ORDER
  ⟨F3⟩: WORDS
  ⟨F4⟩: FILE?
  ⟨F5⟩: LS
  ⟨F6⟩: PWD
  ⟨F7⟩: PATH
  ⟨F8⟩: FREE?
  ⟨F9⟩: not used
  ⟨F10⟩: V

5. Error Messages

If Forth can't evaluate the input line, it displays an error message. Often a typo is the cause of the error — then Forth reacts with "don't know ⟨***name***⟩". You'll see an error message instead of the "ok". Try the following example:

`hallo`⟨RET⟩ `don't know hallo`

An error message is displayed.

The most frequent mistakes are:

don't know? This word is not defined. Or: this number is not converted. It's likely a typo, wrong base, or you wanted to use a word in a vocabulary that's not in the search order.

unstructured The program is not well-structured. You wrote an IF without THEN, a BEGIN without REPEAT or UNTIL, or vice versa. Forth doesn't say what's missing, it bails out at the first control structure word that doesn't work.

compile only You can't execute this word in the interpreter, you only can compile it. Example: return stack manipulation words, control structures.

Stack empty The Stack is empty, i. e. the last word took more elements from the stack than the stack contained.

3 And so Forth...

1. Preface

1.1. Copyright

1.2. Introduction

Don't you hate it? You've just got a new programming language and you're trying to write your first program. You want to use a certain feature (you know it's got to be there) and you can't find it in the manual.

I've had that experience many times. So in this manual you will find many short features on all kind of topics. How to input a number from the keyboard, what a cell is, etc.

I hope this will enable you to get quickly on your way. If it didn't, email me at 'hansoft@bigfoot.com'. You will not only get an answer, but you will help future Forth users as well.

You can use this manual two ways. You can either just get what you need or work your way through. Every section builds on the knowledge you obtained in the previous sections. All sections are grouped into levels. We advise you to use what you've learned after you've worked your way through a level.

If this isn't enough to teach you Forth you can always get a real good textbook on Forth, like "Starting Forth" by Leo Brodie. Have fun!

1.3. About this primer

This primer was originally written for 4tH, my own Forth compiler. 4tH isn't ANS-Forth compliant, by ANS-Forth standards not even a ANS-Forth system. After a while I got questions why certain examples weren't working. Since I tested every single one of them I wondered why. Until it dawned on me: people learning Forth were using my primer!

So due to high demand I started to rewrite it for ANS-Forth compliant systems. Most of these systems don't even have a manual *at all* so the need for it should be great. The next question was: which format. Since I wanted to learn LyX anyway, I settled for that. You can produce various formats with it which are readable on most systems, including MS-DOS, MS-Windows and Linux.

The next question was: how far do you go. The original version was heavily geared towards 4tH, which reflects my own views on Forth. And those views have some criticism

towards ANS-Forth in it. However, since Leo Brodie took the liberty in "Thinking Forth" to express his views, I thought I should have the freedom to express mine.

Some examples, especially in the "Advanced topics" chapter, use special 4tH extensions. Fortunately Wil Baden had helped me to write a 4tH-to-ANS-Forth interface. Since some of these extensions cover functionalities commonly found in other languages I decided to keep those sections in, using the Easy4tH definitions. In the previous chapters you'll find some 4tH words as well, but very sparingly.

You may find that some examples are not working with your specific Forth compiler. That may have several reasons. First, your compiler may not support all ANS-Forth wordsets. Second, your compiler may not be completely ANS-Forth compliant. I've tested most of these examples with GForth or Win32Forth, which are (almost) 100% ANS-Forth compliant. Third, your compiler might be case-sensitive.

The ANS-Forth standard is a very important document. I can only advise you to get it. You should have no trouble finding it on the internet. I can only hope that the compiler you chose *at least* documented its ANS-Forth compatibility.

This primer was written in the hope that it will be useful and that starting Forthers aren't put off by the high price of Forth textbooks. It is dedicated to Leo Brodie, who taught me much more than just Forth.

Hans Bezemer

Den Haag, 2001-03-07

2. Forth fundamentals

2.1. Making calculations without parenthesis

To use Forth you must understand Reverse Polish Notation. This is a way to write arithmetic expressions. The form is a bit tricky for people to understand, since it is geared towards making it easy for the computer to perform calculations; however, most people can get used to the notation with a bit of practice.

Reverse Polish Notation stores values in a stack. A stack of values is just like a stack of books: one value is placed on top of another. When you want to perform a calculation, the calculation uses the top numbers on the stack. For example, here's a typical addition operation:

```
1 2 +
```

When Forth reads a number, it just puts the value onto the stack. Thus 1 goes on the stack, then 2 goes on the stack. When you put a value onto the stack, we say that you push it onto the stack. When Forth reads the operator '+', it takes the top two values off the stack, adds them, then pushes the result back onto the stack. This means that the stack contains:

```
3
```

after the above addition. As another example, consider:

```
2 3 4 + *
```

(The '*' stands for multiplication.) Forth begins by pushing the three numbers onto the stack. When it finds the '+', it takes the top two numbers off the stack and adds them. (Taking a value off the stack is called popping the stack.) Forth then pushes the result of the addition back onto the stack in place of the two numbers. Thus the stack contains:

```
2 7
```

When Forth finds the '*' operator, it again pops the top two values off the stack. It multiplies them, then pushes the result back onto the stack, leaving:

```
14
```

The following list gives a few more examples of Reverse Polish expressions. After each, we show the contents of the stack, in parentheses.

```
7 2 -           (5)
2 7 -           (-5)
12 3 /          (4)
-12 3 /         (-4)
4 5 + 2 *       (18)
4 5 2 + *       (28)
4 5 2 * -       (-6)
```

2.2. Manipulating the stack

You will often find that the items on the stack are not in the right order or that you need a copy. There are stack-manipulators which can take care of that.

To display a number you use '.', pronounced "dot". It takes a number from the stack and displays it. 'SWAP' reverses the order of two items on the stack. If we enter:

```
2 3 . . cr
```

Forth answers:

```
3 2
```

If you want to display the numbers in the same order as you entered them, you have to enter:

```
2 3 swap . . cr
```

In that case Forth will answer:

```
2 3
```

You can duplicate a number using 'DUP'. If you enter:

```
2 . . cr
```

Forth will complain that the stack is empty. However, if you enter:

```
2 dup . . cr
```

Forth will display:

```
2 2
```

Another way to duplicate a number is using 'OVER'. In that case not the topmost number of the stack is duplicated, but the number beneath. E.g.

```
2 3 dup . . . cr
```

will give you the following result:

```
3 3 2
```

But this one:

```
2 3 over . . . cr
```

will give you:

```
2 3 2
```

Sometimes you want to discard a number, e.g. you duplicated it to check a condition, but since the test failed, you don't need it anymore. 'DROP' is the word we use to discard numbers. So this:

```
2 3 drop .
```

will give you "2" instead of "3", since we dropped the "3".

The final one I want to introduce is 'ROT'. Most users find 'ROT' the most complex one since it has its effects deep in the stack. The thirdmost item to be exact. This item is taken from its place and put on top of the stack. It is 'rotated', as this small program will show you:

```
1 2 3                                   \ 1 is the thirdmost item
     . . . cr                           \ display all numbers
     ( This will display '3 2 1' as expected)
     1 2 3                              \ same numbers stacked
     rot                                \ performs a 'ROT'
     . . . cr                           \ same operation
     ( This will display '1 3 2'!)
```

2.3. Deep stack manipulators

There are two manipulators that can dig deeper into the stack, called 'PICK' and 'ROLL' but I cannot recommend them. A stack is NOT an array! So if there are some Forth-83 users out there, I can only tell you: learn Forth the proper way. Programs that have so many items on the stack are just badly written. Leo Brodie agrees with me.

If you are in 'deep' trouble you can always use the returnstack manipulators. Check out that section.

2.4. Pass arguments to functions

There is no easier way to pass arguments to functions as in Forth. Functions have another name in Forth. We call them "words". Words take their "arguments" from the stack and leave the "result" on the stack.

Other languages, like C, do exactly the same. But they hide the process from you. Because passing data to the stack is made explicit in Forth it has powerful capabilities. In other languages, you can get back only one result. In Forth you can get back several!

All words in Forth have a stack-effect-diagram. It describes what data is passed to the stack in what order and what is returned. The word '*' for instance takes numbers from the stack, multiplies them and leaves the result on the stack. It's stack-effect-diagram is:

```
n1 n2 -- n3
```

Meaning it takes number n1 and n2 from the stack, multiplies them and leaves the product (number n3) on the stack. The rightmost number is always on top of the stack, which means it is the first number which will be taken from the stack. The word '.' is described like this:

```
n --
```

Which means it takes a number from the stack and leaves nothing. Now we get to the most powerful feature of it all. Take this program:

```
2     ( leaves a number on the stack)
3     ( leaves a number on the stack on top of the 2)
*     ( takes both from the stack and leaves the result)
.     ( takes the result from the stack and displays it)
```

Note that all data between the words '*' and '.' is passed implicitly! Like putting LEGO stones on top of another. Isn't it great?

2.5. Making your own words

Of course, every serious language has to have a capability to extend it. So has Forth. The only thing you have to do is to determine what name you want to give it. Let's say you want to make a word which multiplies two numbers and displays the result.

Well, that's easy. We've already seen how you have to code it. The only words you need are '*' and '.'. You can't name it '*' because that name is already taken. You could name it 'multiply', but is that a word you want to type in forever? No, far too long.

Let's call it '*.'. Is that a valid name? If you've programmed in other languages, you'll probably say it isn't. But it is! The only characters you can't use in a name are whitespace characters (¡CR¿, ¡LF¿, ¡space¿, ¡TAB¿). It depends on the Forth you're using whether it is case- sensitive or not, but usually it isn't.

So '*.' is okay. Now how do we turn it into a self-defined word. Just add a colon at the beginning and a semi-colon at the end:

```
: *. * . ;
```

That's it. Your word is ready for use. So instead of:

```
2 3 * .
```

We can type:

```
: *. * . ;
2 3 *.
```

And we can use our '*.' over and over again. Hurray, you've just defined your first word in Forth!

2.6. Adding comment

Adding comment is very simple. In fact, there are two ways to add comment in Forth. That is because we like programs with a lot of comment.

The first form you've already encountered. Let's say we want to add comment to this little program:

```
: *. * . ;
2 3 *.
```

So we add our comment:

```
: *. * . ;     This will multiply and print two numbers
2 3 *.
```

Forth will not understand this. It will desperately look for the words 'this', 'will', etc. However the word '\' will mark everything up to the end of the line as comment. So this will work:

```
: *. * . ;     \ This will multiply and print two numbers
2 3 *.
```

There is another word called '(' which will mark everything up to the next ')' as comment. Yes, even multiple lines. Of course, these lines may not contain a ')' or you'll make Forth very confused. So this comment will be recognized too:

```
: *. * . ;     ( This will multiply and print two numbers)
2 3 *.
```

Note that there is a whitespace-character after both '\' and '('. This is mandatory!

2.7. Text-format of Forth source

Forth source can be simple ASCII-files. And you can use any layout as long a this rule is followed:

> All words are separated by at least one whitespace character!

Well, in Forth everything is a word or becoming a word. Yes, even '\' and '(' are words! And you can add all the empty lines or spaces or tabs you like, Forth won't care and your harddisk supplier either.

However, some Forths still use a special line editor, which works with screens. Screens are usually 1K blocks, divided into 16 lines of 64 characters. Explaining how these kind of editors work goes beyond the scope of this manual. You have to check the documentation of your Forth compiler on that. The files these editors produce are called blockfiles.

2.8. Displaying string constants

Displaying a string is as easy as adding comment. Let's say you want to make the ultimate program, one that is displaying "Hello world!". Well, that's almost the entire program. The famous 'hello world' program is simply this in Forth:

```
.( Hello world!)
```

Enter this and it works. Yes, that's it! No declaration that this is the main function and it is beginning here and ending there. May be you think it looks funny on the display. Well, you can add a carriage return by adding the word 'CR'. So now it looks like:

```
.( Hello world!) cr
```

Still pretty simple, huh?

2.9. Declaring variables

One time or another you're going to need variables. Declaring a variable is easy.

```
variable one
```

The same rules for declaring words apply for variables. You can't use a name that already has been taken or you'll get pretty strange results. A variable is a word too! And whitespace characters are not allowed. Note that Forth is usually not case-sensitive!

2.10. Using variables

Of course variables are of little use when you can't assign values to them. This assigns the number 6 to variable 'ONE':

```
6 one !
```

We don't call '!' bang or something like that, we call it 'store'. Of course you don't have to put a number on the stack to use it, you can use a number that is already on the stack. To retrieve the value stored in 'ONE' we use:

```
one @
```

The word '@' is called 'fetch' and it puts the number stored in 'one' on the stack. To display it you use '.':

```
one @ .
```

There is a shortcut for that, the word '?', which will fetch the number stored in 'ONE' and displays it:

```
one ?
```

2.11. Built-in variables

Forth has two built-in variables you can use for your own purposes. They are called 'BASE' and '¿IN'. 'BASE' controls the radix at run-time, '¿IN' is used by 'WORD' and 'PARSE'.

2.12. What is a cell?

A cell is simply the space a number takes up. So the size of a variable is one cell. The size of a cell is important since it determines the range Forth can handle. We'll come to that further on.

2.13. Declaring and using constants

Declaring a simple constant is easy too. Let's say we want to make a constant called 'FIVE':

```
5 constant five
```

Now you can use 'FIVE' like you would '5'. E.g. this will print five spaces:

```
five spaces
```

The same rules for declaring words apply for constants. You shouldn't use a name that already has been taken. A constant is a word too! And whitespace characters are not allowed. Note that Forth is usually not case-sensitive.

2.14. Built-in constants

There are several built-in constants. Of course, they are all literals in case you wonder. Here's a list. Refer to the glossary for a more detailed description:

1. BL
2. FALSE
3. PAD
4. TIB
5. TRUE

2.15. Using booleans

Booleans are expressions or values that are either true or false. They are used to conditionally execute parts of your program. In Forth a value is false when it is zero and true when it is non-zero. Most booleans come into existence when you do comparisons. This one will determine whether the value in variable 'VAR' is greater than 5. Try to predict whether it will evaluate to true or false:

```
variable var
4 var !
var @ 5 > .
```

No, it wasn't! But hey, you can print booleans as numbers. Well, they are numbers. But with a special meaning as we will see in the next section.

2.16. IF-ELSE constructs

Like most other languages you can use IF-ELSE constructs. Let's enhance our previous example:

```
variable var
4 var !

: test
      var @ 5 >
      if ." Greater" cr
      else ." Less or equal" cr
      then
;

test
```

So now our program does the job. It tells you when it's greater and when not. Note that contrary to other languages the condition comes before the 'IF' and 'THEN' ends the IF-clause. In other words, whatever path the program takes, it always continues after the 'THEN'. A tip: think of 'THEN' as 'ENDIF'..

2.17. FOR-NEXT constructs

Forth does also have FOR-NEXT constructs. The number of iterations is known in this construct. E.g. let's print the numbers from 1 to 10:

```
: test
      11 1 do i . cr loop
;

test
```

The first number presents the limit. When the limit is goes beyond the limit minus one the loop terminates. The second number presents the initial value of the index. That's where it starts of. So remember, this loop iterates at least once! You can use '?DO' instead of 'DO'. That will not enter the loop if the limit and the index are the same to begin with:

```
: test
      0 0 ?do i . cr loop
;

test
```

'I' represents the index. It is not a variable or a constant, it is a predefined word, which puts the index on the stack, so '.' can get it from the stack and print it.

But what if I want to increase the index by two? Or want to count downwards? Is that possible. Sure. There is another construct to do just that. Okay, let's take the first question:

```
: test
      11 1 do i . cr 2 +loop
;

test
```

This one will produce exactly what you asked for. An increment by two. This one will produce all negative numbers from -1 to -11:

```
: test
      -11 -1 do i . cr -1 +loop
;

test
```

Why -11? Because the loop terminates when it reached the limit minus one. And when you're counting downward, that is -11. You can change the step if you want to, e.g.:

```
: test
      32767 1 do i . i +loop
;

test
```

This will print: 1, 2, 4, 8, all up to 16384. Pretty flexible, I guess. You can break out of a loop by using 'LEAVE':

```
: test
      10 0 do i dup 5 = if drop leave else . cr then loop
:

test
```

2.18. WHILE-DO constructs

A WHILE-DO construction is a construction that will perform zero or more iterations. First a condition is checked, then the body is executed. Then it will branch back to the condition. In Forth it looks like this:

```
BEGIN <condition> WHILE <body> REPEAT
```

The condition will have to evaluate to TRUE in order to execute the body. If it evaluates to FALSE it branches to just after the REPEAT. This example does a Fibbonaci test.

```
: fib 0 1
      begin
           dup >r rot dup r> >        \ condition
      while
           rot rot dup rot + dup .    \ body
      repeat
      drop drop drop ;                \ after loop has executed
```

You might not understand all of the commands, but we'll get to that. If you enter "20 fib" you will get:

```
1 2 3 5 8 13 21
```

This construct is particularly handy if you are not sure that all data will pass the condition.

2.19. REPEAT-UNTIL constructs

The counterpart of WHILE-DO constructs is the REPEAT-UNTIL construct. This executes the body, then checks a condition at 'UNTIL'. If the expression evaluates to FALSE, it branches back to the top of the body (marked by 'BEGIN') again. It executes at least once. This program calculates the largest common divisor.

```
: lcd
     begin
           swap over mod                    \ body
           dup 0=                           \ condition
     until drop . ;
```

If you enter "27 21 lcd" the programs will answer "3".

2.20. Infinite loops

In order to make an infinite loop one could write:

```
: test
      begin ." Diamonds are forever" cr 0 until
;

test
```

But there is a nicer way to do just that:

```
: test
      begin ." Diamonds are forever" cr again
;

test
```

This will execute until the end of times, unless you exit the program another way.

2.21. Getting a number from the keyboard

Let's start with "you're not supposed to understand this". If you dig deeper in Forth you'll find out why it works the way it works. But if you define this word in your program it will read a number from the keyboard and put it on the stack. If you haven't entered a valid number, it will prompt you again.

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number             ( a -- n)
  0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string then >number nip
  0= if d>s r> if negate then else r> drop 2drop (error) then ;

:    input#                                ( -- n)
     begin
          refill drop bl word number       ( n)
          dup (error) <>                   ( n f)
          dup 0=                           ( n f -f)
          if swap drop then                ( f | n f)
     until ;
```

2.22. Aligning numbers

You may find that printing numbers in columns (I prefer "right-aligned") can be pretty hard. That is because the standard word to print numbers ('.') prints the number and then a trailing space. That is why '.R' was added.

The word '.R' works just like '.' but instead of just printing the number with a trailing space '.R' will print the number right-aligned in a field of N characters wide. Try this and you will see the difference:

```
140 . cr
150 5 .r cr
```

In this example the field is five characters wide, so '150' will be printed with two leading spaces.

3. Arrays and strings

3.1. Declaring arrays of numbers

You can make arrays of numbers very easily. It is very much like making a variable. Let's say we want an array of 16 numbers:

```
create sixteen 16 cells allot
```

That's it, we're done!

3.2. Using arrays of numbers

You can use arrays of numbers just like variables. The array cells are numbered from 0 to N, N being the size of the array minus one. Storing a value in the 0th cell is easy. It works just like a simple variable:

```
5 sixteen 0 cells + !
```

Which will store '5' in the 0th cell. So storing '7' in the 8th cell
is done like this:

```
7 sixteen 8 cells + !
```

Isn't Forth wonderful? Fetching is done the same of course:

```
sixteen 0 cells + @
sixteen 4 cells + @
```

Plain and easy.

3.3. Creating arrays of constants

Making an array of constants is quite easy. First you have to define the name of the array by using the word 'CREATE. Then you specify all its elements. All elements (even the last) are terminated by the word ','. An example:

```
create sizes 18 , 21 , 24 , 27 , 30 , 255 ,
```

Please note that ',' is a word! It has to be separated by spaces on both ends.

3.4. Using arrays of constants

Accessing an array of constants is exactly like accessing an array of numbers. In an array of numbers you access the 0th element like this:

```
sixteen 0 cells + @
```

When you access the first element of an array of constants you use this construction:

```
sizes 0 cells + @
```

So I don't think you'll have any problems here.

3.5. Creating strings

In Forth you have to define the maximum length of the string, like Pascal:

```
create name 10 chars allot
```

Note that the string variable includes the count byte. That is a special character that tells Forth how long a string is. You usually don't have to add that yourself because Forth will do that for you. But you will have to reserve space for it.

That means that the string "name" we just declared can contain up to nine characters *AND* the count byte. These kind of strings are usually referred to as counted strings.

E.g. when you want to define a string that has to contain "Hello!" (without the quotes) you have to define a string that is at least 7 characters long:

```
create hello 7 chars allot
```

When you later refer to the string you just defined its address is thrown on the stack. An address is simply a number that refers to its location. As you will see you can work with string-addresses without ever knowing what that number is. But *because* it is a number you can manipulate it like any other number. E.g. this is perfectly valid:

```
hello                                   \ address of string on stack
     dup                                    \ duplicate it
     drop drop                              \ drop them both
```

In the next section we will tell you how to get "Hello!" into the string.

3.6. Initializing strings

You can initialize a string with the 'S"' word. You haven't seen this one yet, but we will discuss it in more depth later on. If you want the string to contain your first name use this construction:

```
: place over over >r >r char+ swap chars cmove r> r> c! ; ( a1 n a2 --)
   : scopy dup >r place r> ;  ( a1 n a2 -- a2)

   create name 16 chars allot
   s" Hello! " name scopy
```

The word "SCOPY" copies the contents of a string constant into a string-variable. As a matter of fact, it does even more. It will leave the stack in the same state as if you had just written:

```
   name
```

So you can continue to manipulate the string with all the words that we'll introduce to you in the following sections. This value on the stack is the 'string address'. For this moment you can ignore it and simply drop the value. BTW, "PLACE" is a more common word[1], which does *not* leave the address.

If you still don't understand it yet, don't worry. As long as you use this construction, you'll get what you want. Just remember that assigning a string constant to a string that is too short will result in an error or even worse, corrupt other strings.

3.7. Getting the length of a string

You get the length of a string by using the word 'COUNT'. It will not only return the length of the string, but also the string address. It is illustrated by this short program:

```
: place over over >r >r char+ swap chars cmove r> r> c! ; ( a1 n a2 --)
     : scopy dup >r place r> ; ( a1 n a2 -- a2)

     32 string greeting                 \ define string greeting
     S" Hello!" greeting scopy          \ set string to 'Hello!'
```

[1] Although not part of the ANS-Forth standard.

```
drop                              \ drop the value SCOPY left
greeting count                    \ get string length
.( String length: ) . cr          \ print the length
drop                              \ discard the address
```

You usually have nothing to do with the string address. However, it may be required by other words like we will see in the following section. If you just want the bare length of the string you can always define a word like 'length$':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;
: length$ count swap drop ;

create greeting 32 cells allot    \ define string greeting
s" Hello!" greeting scopy         \ set string to 'Hello!'
drop                              \ drop the value SCOPY left
greeting length$                  \ get string length
.( String length: ) . cr          \ print the length
```

We wrote in previous sections that "SCOPY" left a value on the stack that was equivalent with writing the name of the string variable. We can now see why "SCOPY" does that. Instead of writing:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;
: length$ count swap drop ;

create greeting 32 cells allot    \ define string greeting
s" Hello!" greeting scopy         \ set string to 'Hello!'
drop                              \ drop the value SCOPY left
greeting length$                  \ get string length
```

We could write:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;
: length$ count swap drop ;

create greeting 32 cells allot    \ define string greeting
s" Hello!" greeting scopy         \ set string to 'Hello!'
length$                           \ get string length
```

This kind of optimization makes a Forth program faster and more compact, which is a rare thing these days!

3.8. Printing a string variable

Printing a string variable is pretty straight forward. The word that is required to print a string variable is 'TYPE'. It requires the string address and the number of characters it has to print. Yes, that are the values that are left on the stack by 'COUNT'! So printing a string means issuing both 'COUNT' and 'TYPE':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

create greeting 32 cells allot   \ define string greeting
s" Hello!" greeting scopy        \ set string to 'Hello!'
count type cr                    \ print the string
```

If you don't like this you can always define a word like 'PRINT$':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

: print$ count type ;
create greeting 32 cells allot   \ define string greeting
s" Hello!" greeting scopy        \ set string to 'Hello!'
print$ cr                        \ print the string
```

3.9. Copying a string variable

You might want to copy one string variable to another. There is a special word for that, named 'CMOVE'. It takes the two strings and copies a given number of characters from the source to the destination. Let's take a look at this example:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

create one 16 chars allot        \ define the first string
create two 16 chars allot        \ define the second string

s" Greetings!" one scopy         \ initialize string one
dup                              \ save the real address
count                            \ get the length of string one
1+                               \ account for the count byte
swap drop                        \ get the real address
two swap                         \ get the order right
cmove                            \ copy the string
two count type  cr               \ print string two
```

The most difficult part to understand is probably why and how to set up the data for 'CMOVE'. Well, 'CMOVE' wants to see these values on the stack:

```
source destination #chars
```

With the expression:

```
one count
```

We get these data on the stack:

```
source+1 length
```

But the count byte hasn't been accounted for so far. That's why we add:

```
1+
```

So now this parameter has the right value. Now we have to restore the true address of the string and tell 'CMOVE' where to copy the contents of string one to. Initially, 'SCOPY' returned the correct address. That is why we saved it using:

```
dup
```

Now we're getting rid of the "corrupted" address by issuing:

```
swap drop
```

This is what we got right now:

```
source #chars
```

If we simply add:

```
two
```

The data is still not presented in the right order:

```
source #chars destination
```

So we add the extra 'SWAP' in order to get it right. Of course you may define a word that takes care of all that:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

: copy$ swap dup count 1+ swap drop rot swap cmove ;

create one 32 chars allot
create two 32 chars allot
s" Greetings!" one scopy
two copy$
```

You may wonder why we keep on defining words to make your life easier. Why didn't we simply define these words in the compiler instead of using these hard to understand words? Sure, but I didn't write the standard. However, most Forths allow you to permanently store these words in their vocabulary. Check your documentation for details.

3.10. Slicing strings

Slicing strings is just like copying strings. We just don't copy all of it and we don't always start copying at the beginning of a string. We'll show you what we mean:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

: nextchar dup dup c@ 1- swap char+ c! char+ ;

create one 32 chars allot        \ define string one
s" Hans Bezemer" one scopy       \ initialize string one
dup count type cr                \ duplicate and print it
nextchar                         \ move one character forward
dup count type cr                \ duplicate and print it again
nextchar                         \ move one character forward
dup count type cr                \ duplicate and print it again
nextchar                         \ move one character forward
count type cr                    \ print it for the last time
```

First it will print "Hans Bezemer", then "ans Bezemer", then "ns Bezemer" and finally "s Bezemer". The word CHAR+ is usually equivalent to 1+, but Forth was defined to run on unusual hardware too - the CPU of a pocket calculator could be a nibble-machine (4-bit) so each CHAR occupies in fact two addresses. And of course, some Forth systems may treat CHAR to be a 16-bit unicode. If we want to discard the first name at all we could even write:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

create one 32 chars allot        \ define string one
s" Hans Bezemer" one scopy       \ initialize string one
dup c@ 5 -                       \ copy address and get count
swap 5 chars + dup rot swap c!   \ save new count
count type cr                    \ print sliced string
```

The five characters we want to skip are the first name (which is four characters) and a space (which adds up to five). There is no special word for slicing strings. There is a smarter way to handle strings in Forth, which we will discuss later on. But if you desperately need slicing you might want to use a word like this. It works just like 'CMOVE' with an extra parameter:

```
: slice$
      swap                       \ reverse dest and #chars
      over over                  \ copy the dest and #chars
      >r >r >r >r                \ store on the return stack
      +                          \ make address to the source
      r> r>                      \ restore dest and #chars
      char+                      \ make address to destination
      swap cmove                 \ copy the string
```

```
      r> r>                       \ restore dest and #chars
      c!                          \ save
;
```

This is another example of "you're not supposed to understand this". You call it with:

```
source index-to-source destination #chars
```

The index-to-source starts counting at one. So this will copy the first name to string "two" and print it:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

: slice$
      swap                        \ reverse dest and #chars
      over over                   \ copy the dest and #chars
      >r >r >r >r                 \ store on the return stack
      +                           \ make address to the source
      r> r>                       \ restore dest and #chars
      char+                       \ make address to destination
      swap cmove                  \ copy the string
      r> r>                       \ restore dest and #chars
      c!                          \ save
;

create one 32 chars allot         \ declare string one
create two 32 chars allot         \ declare string two
s" Hans Bezemer" one scopy        \ initialize string one
1 two 4 slice$                    \ slice the first name
two count type cr                 \ print string two
```

This will slice the last name off and store it in string "two":

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

: slice$
      swap                        \ reverse dest and #chars
      over over                   \ copy the dest and #chars
      >r >r >r >r                 \ store on the return stack
      +                           \ make address to the source
      r> r>                       \ restore dest and #chars
      char+                       \ make address to destination
      swap cmove                  \ copy the string
      r> r>                       \ restore dest and #chars
      c!                          \ save
;

create one 32 chars allot         \ declare string one
```

```
create two 32 chars allot        \ declare string two
s" Hans Bezemer" one scopy       \ initialize string one
6 two 7 slice$                   \ slice the first name
two count type cr                \ print string two
```

Since the last name is seven characters long and starts at position six (start counting with one!). Although this is very "Basic" way to slice strings, we can do this kind of string processing the Forth way. It will probably require less stack manipulations.

3.11. Appending strings

There is no standard word in Forth to concatenate strings. As a matter of fact, string manipulation is one of Forths weakest points. But since we are focused here on doing things, we will present you a word which will get the work done.

The word 'APPEND' appends two strings. In this example string "one" holds the first name and "Bezemer" is appended to string "one" to form the full name. Finally string "one" is printed.

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

: append                         ( a1 n2 a2 --)
      over over                  \ duplicate target and count
      >r >r                      \ save them on the return stack
      count chars +              \ calculate offset target
      swap chars move            \ now move the source string
      r> r>                      \ get target and count
      dup >r                     \ duplicate target and save one
      c@ +                       \ calculate new count
      r> c!                      \ get address and store
;

create one 32 chars allot        \ define string one
s" Hans " one place              \ initialize first string
s" Bezemer" one append           \ append 'Bezemer' to string
one count type cr                \ print first string
```

Of course, you can also fetch the string to be appended from a string variable by using 'COUNT'.

3.12. Comparing strings

If you ever sorted strings you know how indispensable comparing strings is. As we mentioned before, there are very few words in Forth that act on strings. But here is a word that can compare two strings.

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

create one 32 chars allot        \ define string one
s" H. Bezemer" one scopy count   \ initialize string one
create two 32 chars allot        \ define string two
```

```
s" R. Bezemer" two scopy count     \ initialize string two

compare                           \ compare two strings
if
      ." Strings differ"          \ message: strings ok
else
      ." Strings are the same"    \ message: strings not ok
then
cr                                \ send CR
```

Simply pass two strings to 'COMPARE' and it will return a TRUE flag when the strings are different. This might seem a bit odd, but strcmp() does exactly the same. If you don't like that you can always add '0=' to the end of 'COMPARE' to reverse the flag.

3.13. Removing trailing spaces

You probably know the problem. The user of your well-made program types his name and hits the spacebar before hitting the enter-key. There you go. His name will be stored in your datafile with a space and nobody will ever find it.

In Forth there is a special word called '-TRAILING' that removes the extra spaces at the end with very little effort. Just paste it after 'COUNT'. Like we did in this example:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: scopy dup >r place r> ;

create one 32 chars allot         \ define a string
s" Hans Bezemer    "              \ string with trailing spaces
one scopy                         \ now copy it to string one

dup                               \ save the address

." ["                             \ print a bracket
count type                        \ old method of printing
." ]" cr                          \ print bracket and newline

." ["                             \ print a bracket
count -trailing type              \ new method of printing
." ]" cr                          \ print a bracket and newline
```

You will see that the string is printed twice. First with the trailing spaces, second without trailing spaces. And what about leading spaces? Patience, old chap. You've got a lot of ground to cover.

3.14. String constants and string variables

Most computer languages allow you to mix string constants and string variables. Not in Forth. In Forth they are two distinct datatypes. To print a string constant you use the word '."'. To print a string variable you use the 'COUNT TYPE' construction.

There are only two different actions you can do with a string constant. First, you can define one using 's"'. Second, you can print one using '."'.

There are two different ways to represent a string variable in Forth. First, by using just its address, the so-called counted string. Forth relies on the count byte to find the end of the string. Second, by using its address *and* its length. This requires two values.

The word 'TYPE' requires the latter form. Therefore, you have to convert a counted string in order to print it. You can convert an counted string to an "address-count string" with the word 'COUNT'. If you moved a string (by using 'CMOVE') without taking the count byte into account you have to set it yourself.

This may seem a bit mind-boggling to you now, but we'll elaborate a bit further on this subject in the following sections.

3.15. The count byte

The count byte is used to set the length of a counted string. It has nothing to do with British royalty! It is simply the very first byte of a string, containing the length of the actual string following it.

3.16. Printing individual characters

"I already know that!"

Sure you do. If you want to print "G" you simply write:

```
.( G)
```

Don't you? But what if you want to use a TAB character (ASCII 9)? You can't type in that one so easily, huh? You may even find it doesn't work at all!

Don't ever use characters outside the ASCII range 32 to 127 decimal. It may or may not work, but it won't be portable anyway. the word 'EMIT' may be of some help. If you want to use the TAB-character simply write:

```
9 emit
```

That works!

3.17. Getting ASCII values

Ok, 'EMIT' is a nice addition, but it has its drawbacks. What if you want to emit the character "G". Do you have to look up the ASCII value in a table? No. Forth has another word that can help you with that. It is called 'CHAR'. This will emit a "G":

```
char G emit
```

The word 'CHAR' looks up the ASCII-value of "G" and leave it on the stack. Note that 'CHAR' only works with printable characters (ASCII 33 to 127 decimal).

3.18. When to use [CHAR] or CHAR

There is not one, but *two* words for getting the ASCII code of a character, '[CHAR]' and 'CHAR'. Why is that? Well, the complete story is somewhat complex, but one is for use inside colon definitions and one is for use outside colon definitions. And 'CHAR' isn't the only word which is affected. We've put it all together in a neat table for you:

Inside a definition	Outside a definition
."	.(
[CHAR]	CHAR
[']	'

For example, this produces the same results:

```
: Hello ." Hello world" [char] ! emit cr ; Hello
.( Hello world!) char ! emit cr
```

You should also have noticed in the meanwhile that you can't use control structures like DO..LOOP or IF..THEN outside colon definitions. And not only these, others like 'C"" can't be used as well. Real Forth-ers call this "inside a colon definition" thing *compilation mode* and working from the prompt *interpretation mode*. You can do really neat things with it, but that is still beyond you now.

3.19. Printing spaces

If you try to print a space by using this construction:

```
char   emit
```

You will notice it won't work. Sure, you can also use:

```
.(  )
```

But that isn't too elegant. You can use the built-in constant 'BL' which holds the ASCII-value of a space:

```
bl emit
```

That is much better. But you can achieve the same thing by simply writing:

```
space
```

Which means that if you want to write two spaces you have to write:

```
space space
```

If you want to write ten spaces you either have to repeat the command 'SPACE' ten times or use a DO-LOOP construction, which is a bit cumbersome. Of course, Forth has a more elegant solution for that:

```
10 spaces
```

Which will output ten spaces. Need I say more?

3.20. Fetching individual characters

Take a look at this small program:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot        \ define string one
s" Hans" one place               \ initialize string one
```

What is the second character of string "one"? Sure, its an "a". But how can you let your program determine that? You can't use '@' because that word can only access variables.

Sure, you can do that in Forth, but it requires a new word, called 'C@'. Think of a string as an array of characters and you will find it much easier to picture the idea. Arrays in Forth always start with zero instead of one, but that is the count byte. So accessing the first character might be done with:

```
one 1 chars + c@
```

This is the complete program:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot        \ define string one
s" Hans" one place               \ initialize string one
one 2 chars + c@                 \ get the second character
emit cr                          \ print it
```

3.21. Storing individual characters

Storing individual characters works just the same. Keep that array of characters in mind. When we want to fetch a variable we write:

```
my_var @
```

When we want to store a value in a variable we write:

```
5 my_var !
```

Fetching only requires the address of the variable. Storing requires both the address of the variable *AND* the value we want to store. On top of the stack is the address of the variable, below that is value we want to store. Keep that in mind, this is very important. Let's say we have this program:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot        \ define string one
s" Hans" one place               \ initialize string one
```

Now we want to change "Hans" to "Hand". If we want to find out what the 4th character of string "one" is we write:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot        \ define string one
s" Hans" one place               \ initialize string one
one 4 chars + c@                 \ get the fourth character
```

Remember, we start counting from one! If we want to store the character "d" in the fourth character, we have to use a new word, and (yes, you guessed it right!) it is called 'C!':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot        \ define string one
s" Hans" one place               \ initialize string one
one 4 chars +                    \ address of the fourth char
char d                           \ we want to store 'd'
swap                             \ get the order right
c!                               \ now store 'd'
```

If we throw the character "d" on the stack before we calculate the address, we can even remove the 'SWAP':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot        \ define string one
char d                           \ we want to store 'd'
s" Hans" one place               \ initialize string one
one 4 chars +                    \ address of the fourth char
c!                               \ now store 'd'
```

We will present the very same programs, but now with stack-effect-diagrams in order to explain how this works. We will call the index 'i', the character we want to store 'c' and the address of the string 'a'. By convention, stack-effect-diagrams are enclosed by parenthesis.

If you create complex programs this technique can help you to understand more clearly how your program actually works. It might even save you a lot of debugging. This is the first version:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot        ( --)
s" Hans" one place               ( --)
one                              ( a)
4 chars                          ( a i)
+                                ( a+i)
char d                           ( a+i c)
swap                             ( c a+i)
c!                               ( --)
```

Now the second, optimized version:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot        ( --)
char d                           ( c)
s" Hans" one place               ( c)
one                              ( c a)
4 chars                          ( c a i)
+                                ( c a+i)
c!                               ( --)
```

3.22. Getting a string from the keyboard

Of course, you don't want to initialize strings all your life. Real applications get their input from the keyboard. We've already shown you how to get a number from the keyboard. Now we turn to strings.

When programming in BASIC, strings usually have an undefined length. Some BASICs move strings around in memory, others have to perform some kind of "garbage-collection". Whatever method they use, it takes up memory and processor-time.

Forth forces you to think about your application. E.g. when you want to store somebodies name in a string variable, 16 characters will be too few and 256 characters too many. But 64 characters will probably do.

But that poses a problem when you want to get a string from the keyboard. How can you prevent that somebody types a string that is just too long?

The word 'ACCEPT' takes two arguments. First, the string variable where you want to save the input and second, the maximum number of characters it can take. But there is a catch. This program can get you into trouble:

```
64 constant #name                \ length of string
create name #name chars allot    \ define string 'name'

name #name accept                \ input string
name 1+ swap type cr             \ swap count and print
```

Since 64 characters *PLUS* the count byte add up to 65 characters. You will probably want to use this definition instead:

```
: saccept 1- accept ;            \ define safe 'ACCEPT'

64 constant #name                \ length of string
create name #name chars allot    \ define string 'name'

name #name saccept               \ input string
name 1+ swap type cr             \ print string
```

This "safe" version decrements the count so the user input will fit nicely into the string variable. In order to terminate it you write:

```
: saccept 1- accept ;            \ define safe 'ACCEPT'

64 constant #name                \ length of string
```

```
create name #name chars allot    \ define string 'name'

name dup #name saccept           \ input string
swap c!                          \ set count byte
```

The word 'ACCEPT' always returns the number of characters it received. This is the end of the second level. Now you should be able to understand most of the example programs and write simple ones. I suggest you do just that. Experience is the best teacher after all.

3.23. What is the TIB?

The TIB stands for "Terminal Input Buffer" and is used by one single, but very important word called 'REFILL'. In essence, 'REFILL' does the same thing as 'ACCEPT', except that it has a dedicated area to store its data and sets up everything for parsing. Whatever you type when you call 'REFILL', it is stored in the TIB.

3.24. What is the PAD?

The PAD is short for "scratch-pad". It is a temporary storage area for strings. It is heavily used by Forth itself, e.g. when you print a number the string is formed in the PAD. Yes, that's right: when you print a number it is first converted to a string. Then that string is 'COUNT'ed and 'TYPE'd. You can even program that subsystem yourself as we will see when we encounter formatted numbers.

3.25. How do I use TIB and PAD?

In general, you don't. The TIB is a system-related area and it is considered bad practice when you manipulate it yourself. The PAD can be used for temporary storage, but beware! Temporary really means temporary. A few words at the most, provided you don't generate any output or do any parsing.

Think of both these areas as predefined strings. You can refer to them as 'TIB' and 'PAD'. You don't have to declare them in any way. This program is perfectly alright:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

s" Hello world" pad place        \ store a string in pad
one count type cr                \ print contents of the pad
```

3.26. Temporary string constants

Hey, haven't we already seen this? Yes, you have.

```
s" This is a string" type cr
```

No 'COUNT'? No. 'S"' leaves its address and its length on the stack, so we can call 'TYPE' right away. Note that this string doesn't last forever. If you wait too long it will be overwritten. It depends on your system how long the string will last.

3.27. Simple parsing

We have already discussed 'REFILL' a bit. We've seen that it is closely related to 'ACCEPT'. 'REFILL' returns a true flag if all is well. When you use the keyboard it usually is, so we can safely drop it, but we will encounter a situation where this flag comes in handy. If you want to get a string from the keyboard, you only have to type:

```
refill drop                          \ get string from keyboard
```

Every next call to 'REFILL' will overwrite any previously entered string. So if you want to do something with that string you've got to get it out of there, usually to one of your own strings.

But if accessing the TIB directly is not the proper way, what is? The use of 'REFILL' is closely linked to the word 'WORD', which is a parser. 'WORD' looks for the delimiter, whose ASCII code is on the stack.

If the string starts with the delimiter, it will skip this and all subsequent occurrences until it finds a string. Then it will look for the delimiter again and slice the string right there. It then copies the sliced string to PAD and returns its address. This extremely handy when you want to obtain filtered input. E.g. when you want to split somebodies name into first name, initials and lastname:

```
Hans L. Bezemer
```

Just use this program:

```
: test
      ." Give first name, initials, lastname: "
      refill drop                    \ get string from keyboard
      bl word                        \ parse first name
      ." First name: "               \ write message
      count type cr                  \ type first name
      bl word                        \ parse initials
      ." Initials  : "               \ write message
      count type cr                  \ type initials
      bl word                        \ parse last name
      ." Last name : "               \ write message
      count type cr                  \ write last name
;

test
```

You don't have to parse the entire string with the same character. This program will split up an MS-DOS filename into its components:

```
: test
      ." DOS filename: " refill      \ input a DOS filename
      drop cr                        \ get rid of the flag

      [char] : word                  \ parse drive
      ." Drive: " count type ." :" cr
                                     \ print drive
```

```
        begin
             [char] \ word              \ parse path
             dup count 0<>              \ if not a NULL string
        while                           \ print path
             drop ." Path : " count type cr
        repeat                          \ parse again
        drop drop                       \ discard addresses
;

test
```

If 'WORD' reaches the end of the string and the delimiter is still not found, it returns the remainder of that string. If you try to parse beyond the end of the string, it returns a NULL string. That is an empty string or, in other words, a string with length zero.

Therefore, we checked whether the string had zero length. If it had, we had reached the end of the string and further parsing was deemed useless.

3.28. Converting a string to a number

We now learned how to parse strings and retrieve components from them. But what if these components are numbers? Well, there is a way in Forth to convert a string to a number, but like every number-conversion routine it has to act on invalid strings. That is, strings that cannot be converted to a valid number.

This implementation uses an internal error-value, called '(ERROR)'. The constant '(ERROR)' is a strange number. You can't negate it, you can't subtract any number from it and you can't print it. If 'NUMBER' can't convert a string it returns that constant. Forth has its own conversion word called '¿NUMBER', but that is a lot harder to use. Let's take a look at this program:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
  2drop (error) then ;

: test
        ." Enter a number: "            \ write prompt
        refill drop                     \ enter string
        bl word                         \ parse string
        number dup                      \ convert to a number
        (error) =                       \ test for valid number
        if                              \ if not valid
             ." You didn't enter a valid number!" drop cr
        else                            \ print if valid
             ." The number was: " . cr
        then
;
```

```
test
```

You first enter a string, then it parsed and 'WORD' returns the address where that string is stored. 'NUMBER' tries to convert it. If 'NUMBER' returns '(ERROR)' it wasn't a valid string. Otherwise, the number it right on the stack, waiting to be printed. That wasn't so hard, was it?

3.29. Controlling the radix

If you are a programmer, you know how important this subject is to you. Sometimes, you want to print numbers in octal, binary or hex. Forth can do that too. Let's take the previous program and alter it a bit:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
  2drop (error) then ;

: test
      ." Enter a number: "            \ write prompt
      refill drop                     \ enter string
      bl word                         \ parse string
      number dup                      \ convert to a number
      (error) =                       \ test for valid number
      if                              \ if not valid
           ." You didn't enter a valid number!" drop cr
      else                            \ print if valid
           hex
           ." The number was: " . cr
      then
;

test
```

We added the word 'HEX' just before printing the number. Now the number will be printed in hexadecimal. Forth has a number of words that can change the radix, like 'DECIMAL' and 'OCTAL'. They work in the same way as 'HEX'.

Forth always starts in decimal. After that you are responsible. Note that all radix control follows the flow of the program. If you call a self-defined word that alters the radix all subsequent conversion is done too in that radix:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]
```

```
: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
  2drop (error) then ;

: .hex hex . ;                    \ print a number in hex

: test
  ." Enter a number: "            \ write prompt
  refill drop                     \ enter string
  bl word                         \ parse string
  number dup                      \ convert to a number
  (error) =                       \ test for valid number
  if                              \ if not valid
      ." You didn't enter a valid number!" drop cr
  else                            \ print if valid
      ." The number was: " .hex cr
  then
;

test
```

In this example not only that single number is printed in hex, but also all subsequent numbers will be printed in hex! A better version of the ".HEX" definition would be:

```
: .hex hex . decimal ;
```

Since that one resets the radix back to decimal. Words like 'HEX' do not only control the output of a number, but the input of numbers is also affected:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
  2drop (error) then ;

: test
  ." Enter a number: "              \ write prompt
  refill drop                       \ enter string
  bl word                           \ parse string
  hex                               \ convert hexadecimal
  number dup                        \ convert to a number
  (error) =                         \ test for valid number
  if                                \ if not valid
      ." You didn't enter a valid number!" drop cr
  else                              \ print if valid
      dup
```

```
        ." The number was: " decimal . ." decimal" cr
        ." The number was: " hex . ." hex" cr
    then
;

test
```

'NUMBER' will now also accept hexadecimal numbers. If the number is not a valid hexadecimal number, it will return '(ERROR)'. You probably know there is more to radix control than 'OCTAL', 'HEX' and 'DECIMAL'. No, we have not forgotten them. In fact, you can choose any radix between 2 and 36. This slightly modified program will only accept binary numbers:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
  2drop (error) then ;

: binary 2 base ! ;

: test
  ." Enter a number: "              \ write prompt
  refill drop                       \ enter string
  bl word                           \ parse string
  binary                            \ convert hexadecimal
  number dup                        \ convert to a number
  (error) =                         \ test for valid number
  if                                \ if not valid
      ." You didn't enter a valid number!" drop cr
  else                              \ print if valid
      dup                           \ both decimal and hex
      ." The number was: " decimal . ." decimal" cr
      ." The number was: " hex . ." hex" cr
  then
;

test
```

'BASE' is a predefined variable that enables you to select any radix between 2 and 36. This makes Forth very flexible:

```
hex 02B decimal . cr
```

However, this won't work:

```
: wont-work hex 02B decimal . cr ;
```

But this will:

```
hex
: will-work 02B decimal . cr ;
```

Why that? Well, 'HEX' will just be compiled, not executed. So when Forth tries to compile "02B", it doesn't recognize it as a hexadecimal number and will try to find word '02B'. Which it can't of course. Note that after "WILL-WORK" has been compiled all numbers following it will stuill be compiled as hexadecimal numbers. Why? Because 'DECIMAL' is compiled too! You should place a 'DECIMAL' outside the definition in order to reset the radix. BTW, it is always a good idea to add a leading zero to a hexadecimal number. For example, is this a hex number or a word:

```
face
```

3.30. Pictured numeric output

You probably have used this before, like when writing Basic. Never heard of "PRINT USING.."? Well, it is a way to print numbers in a certain format. Like telephone-numbers, time, dates, etc. Of course Forth can do this too. In fact, you've probably used it before. Both '.' and '.R' use the same internal routines. They are called just before a number is printed.

This numeric string is created in the PAD and overwritten with each new call. But we'll go into that a bit later on.

What you have to remember is that you define the format reverse. What is printed first, is defined last in the format. So if you want to print:

```
060-5556916
```

You have to define it this way:

```
6196555-060
```

Formatting begins with the word '¡#' and ends with the word '#¿'. A single number is printed using '#' and the remainder of the number is printed using '#s' (which is always at least one digit). Let's go a bit further into that:

```
: print# s>d <# #s #> type cr ;
256 print#
```

This simply prints a single number (since only '#S' is between the '¡#' and the '#¿' and goes to a new line. There is hardly any difference with '.'. You can try any (positive) number. Note that the values that '#¿' leaves on the stack can directly be used by 'TYPE'. You can forget about the 'S¿D' word. Just don't forget to put it there.

This is a slightly different format:

```
: print3# s>d <# # # # #> type cr ;
256 print3#
1 print3#
1000 print3#
```

This one will print "256", "001" and "000". Always the last three positions. The '#' simply stands for 'print a single digit'. So if you want to print a number with at least three digits, the format would be:

```
#s # #
```

That is: print the remainder of the number (at least one digit) and then two more. Now reverse it:

```
# # #s
```

Enclose it by 'S¿D', '¡#' and '#¿' and add 'TYPE CR':

```
s>d <# # # #s #> type cr
```

And that's it! Is it? Not quite. So far we've only printed positive numbers. If you try a negative number, you will find it prints garbage. This behavior can be fixed with the word 'SIGN'.

'SIGN' simply takes the number from the stack and prints a "-" when it is negative. The problem is that all other formatting words can only handle positive numbers. So we need the same number twice. One with the sign and one without. A typical signed number formatting word looks like:

```
: signed# dup >r abs s>d <# #s r> sign #> type ;
```

Note the 'DUP ABS' sequence. First the number is duplicated (for 'SIGN') and then the absolute value is taken (for the other formatting words). So we got the on the stack twice. First on the returnstack with sign (for 'SIGN'), second without sign (for the other formatting words). Does that make sense to you?

We can place 'SIGN' wherever we want. If we want to place the sign after the number (like some accountants do) we would write:

```
: account# dup >r abs s>d <# r> sign #s #> type ;
```

But that is still not enough to write "$2000.15" is it? Well, in order to do that there is another very handy word called 'HOLD'. The word 'HOLD' just copies any character into the formatted number. Let's give it a try:

```
$2000.16
```

Let's reverse that:

```
61.0002$
```

So we first want to print two numbers, even when they are zero:

```
# # .0002$
```

Then we want to print a dot. This is where 'HOLD' comes in. 'HOLD' takes an ASCII code and places the equivalent character in the formatting string. We don't have to look up the ASCII code for a dot of course. We can use 'CHAR':

```
# # char . hold 0002$
```

Then we want to print the rest of the number (which is at least one digit):

```
# # char . hold #s $
```

Finally we want to print the character "$". Another job for 'HOLD':

```
# # char . hold #s char $ hold
```

So this is our formatting word:

```
: currency <# # # [char] . hold #s [char] $ hold #> type cr ;
```

And we call it like this:

```
200016 currency
```

You can do some pretty complex stuff with these formatting words. Try to figure out this one from the master himself, Leo Brodie:

```
: sextal 6 base ! ;
: :00 # sextal # decimal 58 hold ;
: time# s>d <# :00 :00 #S #> type cr ;
3615 time#
```

Yeah, it prints the time! Pretty neat, huh? Now try the telephone-number we discussed in the beginning. That shouldn't be too hard.

3.31. Converting a number to a string

Since there is no special word in Forth which will convert a number to a string, we'll have to create it ourselves. In the previous section we have seen how a numeric string is created in the PAD. We can use this to create a word that converts a number to a string.

Because the PAD is highly volatile, we have to move the string immediately after its creation. So we'll create a word that not only creates the string, but moves it directly to its proper location:

```
: >string >r dup >r abs s>d <# #s r> sign #>
  r@ char+ swap dup >r cmove r> r> c! ;
  ( n a -- )
```

It takes a number, the address of a string and returns nothing. Example:

```
create num$ 16 chars allot
-1024 num$ >string
num$ count type cr
```

4. Stacks and colon definitions

4.1. The address of a colon-definition

You can get the address of a colon definition by using the word "' (tick):

```
: add + ;                    \ a colon definition
' add . cr                   \ display address
```

Very nice, but what good is it for? Well, first of all the construction "' ADD" throws the address of "ADD" on the stack. You can assign it to a variable, define a constant for it, or compile it into an array of constants:

```
' add constant add-address

variable addr
' add addr !

create addresses ' add ,
```

Are you with us so far? If we would simply write "ADD", "ADD" would be executed right away and no value would be left on the stack. Tick forces Forth to throw the address of "ADD" on the stack instead of executing "ADD". What you can actually do with it, we will show you in the next section.

4.2. Vectored execution

This is a thing that can be terribly difficult in other languages, but is extremely easy in Forth. Maybe you've ever seen a BASIC program like this:

```
10 LET A=40
20 GOSUB A
30 END
40 PRINT "Hello"
50 RETURN
60 PRINT "Goodbye"
70 RETURN
```

If you execute this program, it will print "Hello". If you change variable "A" to "60", it will print "Goodbye". In fact, the mere expression "GOSUB A" can do two different things. In Forth you can do this much more comfortable:

```
: goodbye ." Goodbye" cr ;
: hello ." Hello" cr ;

variable a

: greet a @ execute ;

' hello a !
greet

' goodbye a !
greet
```

What are we doing here? First, we define a few colon-definitions, called "HELLO" and "GOODBYE". Second, we define a variable called "A". Third, we define another colon-definition which fetches the value of "A" and executes it by calling 'EXECUTE'. Then, we get the address of "HELLO" (by using "' HELLO") and assign it to "A" (by using "A !"). Finally, we execute "GREET" and it says "Hello".

It seems as if "GREET" is simply an alias for "HELLO", but if it were it would print "Hello" throughout the program. However, the second time we execute "GREET", it prints "Goodbye". That is because we assigned the address of "GOODBYE" to "A".

The trick behind this all is 'EXECUTE'. 'EXECUTE' takes the address of e.g. "HELLO" from the stack and calls it. In fact, the expression:

```
hello
```

Is equivalent to:

```
' hello execute
```

This can be extremely useful. We'll give you a little hint:

```
create subs ' hello , ' goodbye ,
```

Does this give you any ideas?

4.3. Using values

A value is a cross-over between a variable and a constant. May be this example will give you an idea:

declaration:

```
variable a      ( No initial value)
1 constant b    ( Initial value, can't change)
2 b + value c   ( Initial value, can change)
```

fetching:

```
a @             ( Variable throws address on stack)
b               ( Constant throws value on stack)
c               ( Value throws value on stack)
```

storing:

```
2 b + a !       ( Expression can be stored at runtime)
                ( Constant cannot be reassigned)
2 b + to c      ( Expression can be stored at runtime)
```

In many aspects, values behave like variables and can replace variables. The only thing you cannot do is make arrays of values.

In fact, a value is a variable that behaves in certain aspects like a constant. Why use a value at all? Well, there are situations where a value can help, e.g. when a constant CAN change during execution. It is certainly not a good idea to replace all variables by values.

4.4. The stacks

Forth has two stacks. So far we've talked about one stack, which is the Data Stack. The Data Stack is heavily used, e.g. when you execute this code:

```
2 3 + .
```

Only the Data Stack is used. First, "2" is thrown on it. Second, "3" is thrown on it. Third, '+' takes both values from the stack and returns the sum. Fourth, this value is taken from the stack by '.' and displayed. So where do we need the other stack for?

Well, we need it when we want to call a colon-definition. Before execution continues at the colon-definition, it saves the address of the currently executed definition on the other stack, which is called the Return Stack for obvious reasons.

Then execution continues at the colon-definition. Every colon-defini tion is terminated by ';', which compiles into 'EXIT'. When 'EXIT' is encountered, the address on top of the Return Stack is popped. Execution then continues at that address, which in fact is the place where we came from.

If we would store that address on the Data Stack, things would go wrong, because we can never be sure how many values were on that stack when we called the colon-definition, nor would be know how many there are on that stack when we encounter 'EXIT'. A separate stack takes care of that.

Try and figure out how this algorithm works when we call a colon-definition from a colon-definition and you will see that it works (Forth is proof of that).

It now becomes clear how 'EXECUTE' works. When 'EXECUTE' is called, the address of the colon-definition is on the Data Stack. All 'EXECUTE' does is copy its address on the Return Stack, take the address from the Data Stack and call it. 'EXIT' never knows the difference..

But the Return Stack is used by other words too. Like 'DO' and 'LOOP'. 'DO' takes the limit and the counter from the Data Stack and puts them on the Return Stack. 'LOOP' takes both of them from the Return Stack and compares them. If they don't match, it continues execution after 'DO'. That is one of the reasons that you cannot split a 'DO..'LOOP'.

However, if you call a colon-definition from within a 'DO'..'LOOP' you will see it works: the return address is put on top of the limit and the counter. As long as you keep the Return Stack balanced (which isn't too hard) you can get away with quite a few things as we will see in the following section.

4.5. Saving temporary values

We haven't shown you how the Return Stack works just for the fun of it. Although it is an area that is almost exclusively used by the system you can use it too.

We know we can manipulate the Data Stack only three items deep (using 'ROT'). Most of the time that is more than enough, but sometimes it isn't.

In Forth there are special words to manipulate stack items in pairs, e.g. "2DUP" (n1 n2 – n1 n2 n1 n2) or "2DROP" (n1 n2 –). In most Forths they are already available, but we could easily define those two ourselves:

```
: 2dup over over ;
: 2drop drop drop ;
```

You will notice that "2SWAP" (n1 n2 n3 n4 – n3 n4 n1 n2) becomes a lot harder. How can we get this deep? You can use the Return Stack for that..

The word '¿R' takes an item from the Data Stack and puts it on the Return Stack. The word 'R¿' does it the other way around. It takes the topmost item from the Return Stack and puts it on the Data Stack. Let's try it out:

```
: 2swap    ( n1 n2 n3 n4)  \ four items on the stack
     rot   ( n1 n3 n4 n2)  \ rotate the topmost three
     >r    ( n1 n3 n4)     \ n2 is now on the Return Stack
     rot   ( n3 n4 n1)     \ rotate other items
     r>    ( n3 n4 n1 n2)  \ get n2 from the Return Stack
;
```

And why does it work in this colon-definition? Why doesn't the program go haywire? Because the Return Stack is and was perfectly balanced. The only thing we had to do was to get off "n2" before the semi-colon was encountered. Remember, the semi-colon compiles into 'EXIT' and 'EXIT' pops a return-address from the Return Stack. Okay, let me show you the Return Stack effects:

```
: 2swap    ( r1)
     rot   ( r1)
     >r    ( r1 n2)
     rot   ( r1 n2)
     r>    ( r1)
;          ( --)
```

Note, these are the Return Stack effects! "R1" is the return-address. And it is there on top on the Return Stack when 'EXIT' is encountered. The general rule is:

> Clean up your mess inside a colon-definition

If you save two values on the Return Stack, get them off there before you attempt to leave. If you save three, get three off. And so on. This means you have to be very careful with looping and branching. Otherwise you have a program that works perfectly in one situation and not in another:

```
: this-wont-work      ( n1 n2 -- n1 n2)
     >r               ( n1)
     0= if            ( --)
          r>          ( n2)
          dup         ( n2 n2)
     else
          1 2         ( 1 2)
     then
;
```

This program will work perfectly if n1 equals zero. Why? Let's look at the Return Stack effects:

```
: this-wont-work      ( r1)
      >r              ( r1 n2)
      0= if           ( r1 n2)
           r>         ( r1)
           dup        ( r1)
      else            ( r1 n2)
           1 2        ( r1 n2)
      then
;
```

You see when it enters the 'ELSE' clause the Return Stack is never cleaned up, so Forth attempts to return to the wrong address. Avoid this, since this can be very hard bugs to fix.

4.6. The Return Stack and the DO..LOOP

We've already told you that the limit and the counter of a DO..LOOP (or DO..+LOOP) are stored on the Return Stack. But how does this affect saving values in the middle of a loop? Well, this example will make that quite clear:

```
: test
  1                   ( n)
  10 0 do             ( n)
       >r             ( --)
       i .            ( --)
       r>             ( n)
  loop                ( n)
  cr                  ( n)
  drop                ( --)
;

test
```

You might expect that it will show you the value of the counter ten times. In fact, it doesn't. Let's take a look at the Return Stack:

```
: test
  1                   ( --)
  10 0 do             ( l c)
       >r             ( l c n)
       i .            ( l c n)
       r>             ( l c)
  loop                ( --)
  cr                  ( --)
  drop                ( --)
;

test
```

You might have noticed (unless you're blind) that it prints ten times the number "1". Where does it come from? Usually 'I' prints the value of the counter, which is on top of the Return Stack.

This time it isn't: the number "1" is there. So 'I' thinks that "1" is actually the counter and displays it. Since that value is removed from the Return Stack when 'LOOP' is encountered, it doesn't do much harm.

We see that we can safely store temporary values on the Return Stack inside a DO..LOOP, but we have to clean up the mess, before we encounter 'LOOP'. So, this rule applies here too:

> Clean up your mess inside a DO..LOOP

But we still have to be prepared that the word 'I' will not provide the expected result (which is the current value of the counter). In fact, 'I' does simply copy the topmost value on the Return Stack. Which is usually correct, unless you've manipulated the Return Stack yourself.

Note that there are other words beside 'I', which do exactly the same thing: copy the top of the Return Stack. But they are intended to be used outside a DO..LOOP. We'll see an example of that in the following section.

4.7. Other Return Stack manipulations

The Return Stack can avoid some complex stack acrobatics. Stack acrobatics? Well, you know it by now. Sometimes all these values and addresses are just not in proper sequence, so you have to 'SWAP' and 'ROT' a lot until they are.

You can avoid some of these constructions by just moving a single value on the Return Stack. You can return it to the Data Stack when the time is there. Or you can use the top of the Return Stack as a kind of local variable.

No, you don't have to move it around between both stacks all the time and you don't have to use 'I' out of its context. There is a well-established word, which does the same thing: 'R@'. This is an example of the use of 'R@':

```
: delete                              ( n --)
      >r #lag +                       ( a1)
      r@ - #lag                       ( a1 a2 n2)
      r@ negate                       ( a1 a2 n2 n3)
      r# +!                           ( a1 a2 n2)
      #lead +                         ( a1 a2 n2 a3)
      swap cmove                      ( a1)
      r> blanks                       ( --)
;
```

'R@' copies the top of the Return Stack to the Data Stack. This example is taken from a Forth-editor. It deletes "n" characters left of the cursor. By putting the number of characters on the Return Stack right away, its value can be fetched by 'R@' without using 'DUP' or 'OVER'. Since it can be fetched at any time, no 'SWAP' or 'ROT' has to come in.

4.8. Altering the flow with the Return Stack

The mere fact that return addresses are kept on the stack means that you can alter the flow of a program. This is hardly ever necessary, but if you're a real hacker you'll try this anyway, so we'd better give you some pointers on how it is done. Let's take a look at this program. Note that we comment on the Return Stack effects:

```
: soup ." soup " ;                    ( r1 r2)
: dessert ." dessert " ;              ( r1 r6)
: chicken ." chicken " ;              ( r1 r3 r4)
: rice ." rice " ;                    ( r1 r3 r5)
: entree chicken rice ;               ( r1 r3)
: dinner soup entree dessert ;        ( r1)
dinner cr                             ( --)
```

And this is the output:

```
soup chicken rice dessert
```

Before we execute "DINNER" the Return Stack is empty. When we enter "DINNER" the return address to the main program is on the Return Stack (r1).

"DINNER" calls "SOUP". When we enter "SOUP" the return address to "DINNER" is on the Return Stack (r2). When we are done with "SOUP", its return address disappears from the Return Stack and execution continues within "DINNER".

Then "ENTREE" is called, putting another return address on the Return Stack (r3). "ENTREE" on its turn, calls "CHICKEN". Another return address (r4) is put on the Return Stack. Let's take a look on what currently lies on the Return Stack:

```
- Top Of Return Stack (TORS) -
r4    -    returns to ENTREE
r3    -    returns to DINNER
r1    -    returns to main program
```

As we already know, ';' compiles an 'EXIT', which takes the TORS and jumps to that address. What if we lose the current TORS? Will the system crash?

Apart from other stack effects (e.g. too few or the wrong data are left on the Data Stack) nothing will go wrong. Unless the colon-definition was called from inside a DO..LOOP, of course. But what DOES happen? The solution is provided by the table: it will jump back to "DINNER" and continue execution from there.

```
: soup ." soup " ;                    ( r1 r2)
: dessert ." dessert " ;              ( r1 r6)
: chicken ." chicken " r> drop ;      ( r1 r3 - r4 gets lost!)
: rice ." rice " ;                    ( r1 r3 r5)
: entree chicken rice ;               ( r1 r3)
: dinner soup entree dessert ;   ( r1)
dinner cr                             ( --)
```

Since "CHICKEN" gets rid of the return address to "ENTREE", "RICE" is never called. Instead, a jump is made to "DINNER" that assumes that "ENTREE" is done, so it continues with "DESSERT". This is the output:

```
soup chicken dessert
```

Note that this is *not* common practice and we do not encourage its use. However, it gives you a pretty good idea how the Return Stack is used by the system.

4.9. Leaving a colon-definition

You can sometimes achieve the very same effect by using the word 'EXIT' on a strategic place. We've already encountered 'EXIT'. It is the actual word that is compiled by ';'.

What you didn't know is that you can compile an 'EXIT' without using a ';'. And it does the very same thing: it pops the return address from the Return Stack and jumps to it. Let's take a look at our slightly modified previous example:

```
: soup ." soup " ;                    ( r1 r2)
: dessert ." dessert " ;               ( r1 r6)
: chicken ." chicken " ;               ( r1 r3 r4)
: rice ." rice " ;                     ( is never reached)
: entree chicken exit rice ;          ( r1 r3)
: dinner soup entree dessert ;    ( r1)
dinner cr                             ( --)
```

After "CHICKEN" has been executed by "ENTREE", an 'EXIT' is encountered. 'EXIT' works just like ';', so Forth thinks the colon-definition has come to an end and jumps back to "DINNER". It never comes to calling "RICE", so the output is:

```
soup chicken dessert
```

'EXIT' is mostly used in combination with some kind of branching like IF..ELSE..THEN. Compare it with 'LEAVE' that leaves a DO..LOOP early.

But now for the big question: what is the difference between 'EXIT' and ';'? Both compile an 'EXIT', but they are not aliases. Forth shuts down the compiler when encountering ';'. This is not performed by 'EXIT'.

4.10. How deep is your stack?

You can ask Forth how many values are on the Data Stack using 'DEPTH'. It will report the number of values, before you executed 'DEPTH'. Let's elaborate on that a little more:

```
.( Begin) cr    ( no values on the stack)
10              ( 1 value on the stack)
5               ( 2 values on the stack)
9               ( 3 values on the stack)
depth           ( 4 values on the stack)
. cr            ( Forth reports "3")
```

5. Advanced topics

5.1. Booleans and numbers

You might have expected we had discussed this subject much earlier. But we haven't and for one very good reason. We've told you a few chapters ago that 'IF' branches if the top of the stack is non-zero. Any number will do. So you would expect that this program will print "I'm here":

```
: test
      1 2 and
      if
           ." I'm here"
      then
;

test
```

In fact, it doesn't! Why? Well, 'AND' is a BINARY operator, not a LOGICAL operator. That means it reacts on bit-patterns. Given two numbers, it will evaluate bits at the same position.

The number "1" is "01" in binary. The number "2" is "10" in binary. 'AND' will evaluate the first bit (binary digit, now you know where that came from!). The first bit is the rightmost bit, so "0" for the number "2" and "1" for the number "1".

'AND' works on a simple rule, if both bits are "1" the result will be "1" on that position. Otherwise it will be "0". So "1" and "0" are "0". The evaluation of the second bit has the same result: "0". We're stuck with a number that is "0". False. So 'IF' concludes that the expression is not true:

```
2 base !              \ set radix to binary
10                    ( binary number "2")
01 AND                ( binary number "1")
= . cr                ( binary result after AND)
```

It will print "0". However, "3" and "2" would work just fine:

```
2 base !              \ set radix to binary
10                    ( binary number "2")
11 AND                ( binary number "3")
. cr                  ( binary result after AND)
```

It will print "10". The same applies to other binary operators as 'OR' and 'INVERT'. 'OR' works just like 'AND' but works the other way around. If both bits are "0" the result will be "0" on that position. Otherwise it will be "1":

```
2 base !              \ set radix to binary
10                    ( binary number "2")
01 OR                 ( binary number "1")
. cr                  ( binary result after OR)
```

It will print "11". We do not encourage the use of 'INVERT' for logical operations, although the standard allows it. You should use '0=' instead. '0=' takes the top of the stack and leave a true-flag if it is zero. Otherwise it will leave a false-flag. That means that if a condition istrue (non-zero), it will leave a false-flag. Which is exactly what a logical NOT should do.

Take a look at his brother '0¡¿'. '0¡¿' takes the top of the stack and leaves a true-flag if it is non-zero. Otherwise it will leave a false- flag. The funny thing is 'AND' and 'OR' work perfectly with flags and behave as expected. '0¡¿' will convert a value to a flag for you. So this works:

```
: test
      1 0<>
      2 0<>
      and if
           ." I'm here" cr
      then
;

test
```

Of course, you don't have to use '0¡¿' when a word returns a flag. You should check the standard for details on that.

5.2. Including your own definitions

At a certain point you may have written a lot of definitions you're very fond of. You use them in most of your programs, so before you actually get to the programs you have to work your way through all these standard definitions. Even worse, when you change one of them you have to edit all your programs. Most Forths have a way to permanently include them in the kernel, but if you're not up to that or want your programs to be as portable as possible you can solve this in a better way.

Just put all of your definitions in a single file and start your program with:

```
s" mydefs.f" included
```

The compiler will now first compile all the definitions in "mydefs.f" before starting with the main program. We've done exactly the same in the following sections. Most of the code you'll find there uses the Easy4tH extensions, so instead of listing them every single time, we've just included them. Easy4tH has old favorites like "PLACE" and "NUMBER" already available to you.

You have to define the constant "/STRING-SPACE" first in order to use it. A value of 16384 should be fine in most cases. If you get an error, you can always increase it.

5.3. Conditional compilation

This is something which can be very handy when you're designing a Forth program for different environments or even different Forth compilers. Let's say you've written a general ledger program in Forth that is so good, you can sell it. Your customers want a demo, of course. You're willing to give one to them, but you're afraid they're going to use the demo without ever paying for it.

One thing you can do is limit the number of entries they can make. So, you copy the source and make a special demo version. But you have to do that for every new release. Wouldn't it just be easier to have one version of the program and just change one single constant? You can with conditional compilation:

```
true  constant DEMO

DEMO [if]
256 constant #Entries
[else]
65536 constant #Entries
```

```
[then]

variable CurrentEntry

create Entries #Entries cells allot
```

We defined a constant, called "DEMO", which is true. So, when the compiler reaches the "DEMO [if]" line, it knows that it has to compile "256 constant Entries", since "DEMO" is true. When it comes to "[else]", it knows it has to skip everything up to "[then]". So, in this case the compiler behaves like you've written:

```
256 constant #Entries
variable CurrentEntry
create Entries #Entries cells allot
```

Would you change "DEMO" to false, the compiler would behave as if you wrote:

```
variable CurrentEntry
65536 constant #Entries
create Entries #Entries cells allot
```

The word '[IF]' only works at compile time and is *never* compiled into the object. '[IF]' takes a expression. If this expression is true, the code from '[IF]' until '[ELSE]' is compiled, just as '[IF]' wasn't there. Is this expression is false, everything '[IF]' up to '[ELSE]' is discarded as if it wasn't there.

That also means you can discard any code that is superfluous in the program. E.g. when you're making a colon-definition to check whether you can make any more entries. If you didn't use conditional compilation, you might have written it like this:

```
: CheckIfFull                    ( n -- n)
      dup #Entries =             ( n f)
      if                         ( n)
           drop                  ( --)
           DEMO                  ( f)
           if                    ( --)
                ." Buy the full version"
           else                  \ give message and exit program
                ." No more entries"
           then                  ( --)

           cr quit
      then                       ( n)
;
```

But this one is nicer and will take up less code:

```
DEMO [IF]
: .Message ." Buy the full version" ;
[ELSE]
: .Message ." No more entries" ;
```

```
[THEN]

: CheckIfFull                    ( n -- n)
      dup #Entries =             ( n f)
      if                         ( n)
           drop                  ( --)
           .Message
           cr quit
      then                       ( n)
;
```

You can also use conditional compilation to discard large chunks of code. This is a much better way than to comment all the lines out, e.g. this won't work anyway:

```
(
      : room?                    \ is it a valid variable?
           dup                   ( n n)
           size 1- invert and    ( n f)
           if                    \ exit program
                drop ." Not an element of ROOM" cr quit
           then
      ;
)
```

This is pretty cumbersome and prone to error:

```
\     : room?                    \ is it a valid variable?
\          dup                   ( n n)
\          size 1- invert and    ( n f)
\          if                    \ exit program
\               drop ." Not an element of ROOM" cr quit
\          then
\     ;
```

But this is something that can easily be handled:

```
false [if]
      : room?                    \ is it a valid variable?
           dup                   ( n n)
           size 1- invert and    ( n f)
           if                    \ exit program
                drop ." Not an element of ROOM" cr quit
           then
      ;
[then]
```

Just change "false" to "true" and the colon-definition is part of the program again. Note that '[IF] .. [THEN]' can be nested! Conditional compilation is very powerful and one of the easiest features a language can have.

5.4. Exceptions

You know when you violate the integrity of Forth, it will exit and report the cause and location of the error. Wouldn't it be nice if you could catch these errors within the program? It would save a lot of error-checking anyway. It is quite possible to check every value within Forth, but it takes code and performance, which makes your program less compact and slower.

Well, you can do that too in Forth. And not even that, you can trigger your own errors as well. This simple program triggers an error and exits Forth when you enter a "0":

```
16384 constant /string-space
s" easy4th.f" included

: input#                                \ get a number
      begin
           refill drop                  ( --)
           bl word number               ( n )
           dup (error) <>               ( n f )
           dup 0=                       ( n f -f )
           if swap drop then            ( f | n f )
      until                             ( input routine )
;
                                        \ get a number
                                        \ if non-zero, return it
                                        \ if zero, throw exception
: could-fail                            ( -- n)
      input# dup 0=
      if 1 throw then
;
                                        \ drop numbers and
                                        \ call COULD-FAIL
: do-it                                 ( --)
      drop drop could-fail
;
                                        \ put 2 nums on stack and
                                        \ execute DO-IT
: try-it                                ( --)
      1 2 ['] do-it execute
      ." The number was" . cr
;
                                        \ call TRY-IT
try-it
```

"TRY-IT" puts two numbers on the stack, gets the execution token of "DO-IT" and executes it. "DO-IT" drops both numbers and calls "COULD-FAIL". "COULD-FAIL" gets a number and compares it against "0". If zero, it calls an exception. If not, it returns the number.

The expression "1 THROW" has the same effect as calling 'QUIT'. The program exits, but with the error message "Unhandled exception". You can use any positive number for

'THROW', but "0 THROW" has no effect. This is called a "user exception", which means you defined and triggered the error.

There are also system exceptions. These are triggered by the system, e.g. when you want to access an undefined variable or print a number when the stack is empty. These exceptions have a negative number, so:

```
throw -4
```

Will trigger the "Stack empty" error. You can use these if you want but we don't recommend it, since it will confuse the users of your program.

You're probably not interested in an alternative for 'QUIT'. Well, 'THROW' isn't. It just enables you to "throw" an exception and exceptions can be caught by your program. That means that Forth won't exit, but transfers control back to some routine. Let's do just that:

```
16384 constant /string-space
s" easy4th.f" included

: input#
      begin
           refill drop               ( --)
           bl word number            ( n )
           dup (error) <>            ( n f )
           dup 0=                    ( n f -f )
           if swap drop then         ( f | n f )
      until                          ( input routine )
;

: could-fail                         ( -- n)
      input# dup 0=
      if 1 throw then
;

: do-it                              ( --)
      drop drop could-fail
;

: try-it                             ( --)
      1 2 ['] do-it catch
      if drop drop ." There was an exception" cr
      else ." The number was" . cr
      then
;

try-it
```

The only things we changed is a somewhat more elaborate "TRY-IT" definition and we replaced 'EXECUTE' by 'CATCH'.

'CATCH' works just like 'EXECUTE', except it returns a result-code. If the result-code is zero, everything is okay. If it isn't, it returns the value of 'THROW'. In this case it

would be "1", since we execute "1 THROW". That is why "0 THROW" doesn't have any effect.

If you enter a nonzero value at the prompt, you won't see any difference with the previous version. However, if we enter "0", we'll get the message "There was an exception", before the program exits.

But hey, if we got that message, that means Forth was still in control! In fact, it was. When "1 THROW" was executed, the stack-pointers were restored and we were directly returned to "TRY-IT". As if "1 THROW" performed an 'EXIT' to the token following 'CATCH'.

Since the stack-pointers were returned to their original state, the two values we discarded in "DO-IT" are still on the stack. But the possibility exists they have been altered by previous definitions. The best thing we can do is discard them.

So, the first version exited when you didn't enter a nonzero value. The second version did too, but not after giving us a message. Can't we make a version in which we can have another try? Yes we can:

```
16384 constant /string-space
s" easy4th.f" included

: input#
      begin
           refill drop                ( --)
           bl word number             ( n )
           dup (error) <>             ( n f )
           dup 0=                     ( n f -f )
           if swap drop then          ( f | n f )
      until                           ( input routine )
;

: could-fail                          ( -- n)
      input# dup 0=
      if 1 throw then
;

: do-it                               ( --)
      drop drop could-fail
;

: retry-it                            ( --)
      begin
           1 2 ['] do-it catch
      while
           drop drop ." Exception, keep trying" cr
      repeat
      ." The number was " . cr
;

retry-it
```

This version will not only catch the error, but it allows us to have another go! We can keep on entering "0", until we enter a nonzero value. Isn't that great? But it gets even better! We can exhaust the stack, trigger a system exception and still keep on going. But let's take it one step at the time. First we change "COULD-FAIL" into:

```
: could-fail                              ( -- n)
      input# dup 0=
      if drop ." Stack: " depth . cr 1 throw then
;
```

This will tell us that the stack is exhausted at his point. Let's exhaust is a little further by redefining "COULD-FAIL" again:

```
: could-fail                              ( -- n)
      input# dup 0=
      if drop drop then
;
```

Another 'DROP'? But wouldn't that trigger an "Stack empty" error? Yeah, it does. But instead of exiting, the program will react as if we wrote "-4 THROW" instead of "DROP DROP". The program will correctly report an exception when we enter "0" and act accordingly.

This will work with virtually every runtime error. Which means we won't have to protect our program against every possible user-error, but let Forth do the checking.

We won't even have to set flags in every possible colon-definition, since Forth will automatically skip every level between 'THROW' and 'CATCH'. Even better, the stacks will be restored to the same depth as they were before 'CATCH' was called.

You can handle the error in any way you want. You can display an error message, call some kind of error-handler, or just ignore the error. Is that enough flexibility for you?

5.5. Lookup tables

Leo Brodie wrote: "I consider the case statement an elegant solution to a misguided problem: attempting an algorithmic expression of what is more aptly described in a decision table". And that is exactly what we are going to teach you.

Let's say we want a routine that takes a number and then prints the appropriate month. In ANS-Forth, you could do that this way:

```
: Get-Month
      case
            1 of ."  January " endof
            2 of ." February " endof
            3 of ."   March  " endof
            4 of ."   April  " endof
            5 of ."    May   " endof
            6 of ."   June   " endof
            7 of ."   July   " endof
            8 of ."  August  " endof
            9 of ." September" endof
            10 of ."  October " endof
```

```
            11 of ." November " endof
            12 of ." December " endof
        endcase
        cr
;
```

This takes a lot of code and a lot of comparing. In this case (little wordplay) you would be better of with an indexed table, like this:

```
16384 constant /string-space
s" easy4th.f" included

create MonthTable
      $"  January " ,
      $" February " ,
      $"   March  " ,
      $"   April  " ,
      $"    May   " ,
      $"   June   " ,
      $"   July   " ,
      $"  August  " ,
      $" September" ,
      $"  October " ,
      $" November " ,
      $" December " ,

: Get-Month                ( n -- )
      12 min 1- MonthTable swap cells + @ pad copy[2] count type cr
;
```

Which does the very same thing and will certainly work faster. Normally, you can't do that this easily in ANS-Forth, but with this primer you can, so use it! But can you use the same method when you're working with a random set of values like "2, 1, 3, 12, 5, 6, 4, 7, 11, 8, 10, 9". Yes, you can. But you need a special routine to access such a table. Of course we designed one for you:

```
: Search-Table             ( n1 a1 n2 n3 -- n4 f)
      swap >r              ( n1 a1 n3)
      rot rot              ( n3 n1 a1)
      over over            ( n3 n1 a1 n1 a1)
      0                    ( n3 n1 a1 n1 a1 n2)

      begin                ( n3 n1 a1 n1 a1 n2)
            swap over      ( n3 n1 a1 n1 n2 a1 n2)
            cells +        ( n3 n1 a1 n1 n2 a2)
            @ dup          ( n3 n1 a1 n1 n2 n3 n3)
            0> >r          ( n3 n1 a1 n1 n2 n3)
```

[2] "COPY" is part of the Easy4tH extensions and will copy a counted string from one address to another (addr1 addr2 – addr2).

```
            rot <>          ( n3 n1 a1 n2 f)
            r@ and          ( n3 n1 a1 n2 f)
      while                 ( n3 n1 a1 n2)
            r> drop         ( n3 n1 a1 n2)
            r@ +            ( n3 n1 a1 n2+2)
            >r over over    ( n3 n1 a1 n1 a1)
            r>              ( n3 n1 a1 n1 a1 n2+2)
      repeat                ( n3 n1 a1 n2)

      r@ if
            >r rot r>       ( n1 a1 n3 n2)
            + cells + @     ( n1 n4)
            swap drop       ( n3)
      else
            drop drop drop  ( n1)
      then

      r>                    ( n f)
      r> drop               ( n f)
;
```

This routine takes four values. The first one is the value you want to search. The second is the address of the table you want to search. The third one is the number of fields this table has. And on top of the stack you'll find the field which value it has to return. The first field must be the "index" field. It contains the values which have to be compared. That field has number zero.

This routine can search zero-terminated tables. That means the last value in the index field must be zero. Finally, it can only lookup positive values. You can change all that by modifying the line with "0¿ ¿r". It returns the value in the appropriate field and a flag. If the flag is false, the value was not found.

Now, how do we apply this to our month table? First, we have to redefine it:

```
16384 constant /string-space
s" easy4th.f" included

0 Constant NULL

create MonthTable
      1 , $"  January " ,
      2 , $" February " ,
      3 , $"   March  " ,
      4 , $"   April  " ,
      5 , $"    May   " ,
      6 , $"   June   " ,
      7 , $"   July   " ,
      8 , $"  August  " ,
      9 , $" September" ,
      10 , $"  October " ,
      11 , $" November " ,
```

```
        12 , $" December " ,
        NULL ,
```

Note that this table is sorted, but that doesn't matter. It would work just as well when it was unsorted. Let's get our stuff together: the address of the table is "MonthTable", it has two fields and we want to return the address of the string, which is located in field 1. Field 0 contains the values we want to compare. We can now define a routine which searches our table:

```
: Search-Month MonthTable 2 1 Search-Table ;     ( n1 -- n2 f)
```

Now, we define a new "Get-Month" routine:

```
: Get-Month                             ( n --)
      Search-Month                      \ search table

      if                                \ if month is found
           pad copy count type          \ print its name
      else                              \ if month is not found
           drop ." Not found"           \ drop value
      then                              \ and show message

      cr
;
```

Is this flexible? Oh, you bet! We can extend the table with ease:

```
16384 constant /string-space
s" easy4th.f" included

0 Constant NULL
3 Constant #MonthFields

create MonthTable
      1 , $"  January " , 31 ,
      2 , $" February " , 28 ,
      3 , $"   March  " , 31 ,
      4 , $"   April  " , 30 ,
      5 , $"    May   " , 31 ,
      6 , $"   June   " , 30 ,
      7 , $"   July   " , 31 ,
      8 , $"  August  " , 31 ,
      9 , $" September" , 30 ,
      10 , $"  October " , 31 ,
      11 , $" November " , 30 ,
      12 , $" December " , 31 ,
      NULL ,
```

Now we make a slight modification to "Search-Month":

```
: Search-Month MonthTable #MonthFields 1 Search-Table ;
```

This enables us to add more fields without ever having to modify "Search-Month" again. If we add another field, we just have to modify "#MonthFields". We can now even add another routine, which enables us to retrieve the number of days in a month:

```
: Search-#Days MonthTable #Monthfields 2 Search-Table ;
```

Of course, there is room for even more optimization, but for now we leave it at that. Do you now understand why Forth shouldn't have a CASE construct?

5.6. Fixed point calculation

We already learned that if we can't calculate it out in dollars, we can calculate it in cents. And still present the result in dollars using pictured numeric output:

```
: currency <# # # [char] . hold #s [char] $ hold #> type cr ;
```

In this case, this:

```
200012 currency
```

will print this:

```
$2000.12
```

Well, that may be a relief for the bookkeepers, but what about us scientists? You can do the very same trick. We have converted some Forth code for you that gives you very accurate results. You can use routines like SIN, COS and SQRT. A small example:

```
31415 CONSTANT PI
10000 CONSTANT 10K          ( scaling constant )
VARIABLE XS                 ( square of scaled angle )

: KN  ( n1 n2 -- n3, n3=10000-n1*x*x/n2 where x is the angle )
      XS @ SWAP /           ( x*x/n2 )
      NEGATE 10K */         ( -n1*x*x/n2 )
      10K +                 ( 10000-n1*x*x/n2 )
;

: (SIN)                     ( x -- sine*10K, x in radian*10K )
      DUP DUP 10K */        ( x*x scaled by 10K )
      XS !                  ( save it in XS )
      10K 72 KN             ( last term )
      42 KN 20 KN 6 KN      ( terms 3, 2, and 1 )
      10K */                ( times x )
;

: SIN                       ( degree -- sine*10K )
      PI 180 */             ( convert to radian )
      (SIN)                 ( compute sine )
;
```

If you enter:

```
45 sin . cr
```

You will get "7071", because the result is multiplied by 10000. You can correct this the same way you did with the dollars: just print the number in the right format.

```
: /10K. <# # # # # [char] . hold #S #> type cr ;
45 sin /10K.
```

This one will actually print:

```
0.7071
```

But note that Forth internally still works with the scaled number, which is "7071". Another example:

```
: SQRT                    ( n1 -- n2, n2**2<=n1 )
      0                   ( initial root )
      SWAP 0              ( set n1 as the limit )
      DO 1 + DUP          ( refresh root )
           2* 1 +         ( 2n+1 )
      +LOOP               ( add 2n+1 to sum, loop if )
;                         ( less than n1, else done )

: .fp <# # [char] . hold #S #> type cr ;
```

If you enter a number of which the root is an integer, you will get a correct answer. You don't even need a special formatting routine. If you enter any other number, it will return only the integer part. You can fix this by scaling the number.

However, scaling it by 10 will get you nowhere, since "3" is the square root of "9", but "30" is not the square root of "90". In that case, we have to scale it by 100, 10,000 or even 1,000,000 to get a correct answer. In order to retrieve the next digit of the square root of "650", we have to multiply it by 100:

```
650 100 * sqrt .fp
```

Which will print:

```
25.4
```

To acquire greater precision we have to scale it up even further, like 10,000. This will show us, that "25.49" brings us even closer to the correct answer.

5.7. Recursion

Yes, but can she do recursion? Of course she can! In order to let a colon-definition call itself, you have to use the word 'RECURSE'. Everybody knows how to calculate a factorial. In Forth you can do this by:

```
: factorial           ( n1 -- n2)
      dup 2 >
      if
           dup 1-
           recurse *
      then
;

10 factorial . cr
```

If you use the word 'RECURSE' outside a colon-definition, the results are undefined. Note that recursion lays a heavy burden on the return stack. Sometimes it is wiser to implement such a routine differently:

```
: factorial
      dup

      begin
           dup 2 >
      while
           1- swap over * swap
      repeat

      drop
;

10 factorial . cr
```

So if you ever run into stack errors when you use recursion, keep this in mind.

5.8. Forward declarations

It doesn't happen very often, but sometimes you have a program where two colon-definitions call each other. There is no special instruction in Forth to do this, like Pascals "FORWARD" keyword, but still it can be done. It even works the same way. Let's say we've got two colon-definitions called "STEP1" and "STEP2". "STEP1" calls "STEP2" and vice versa. First we create a value called "(STEP2)". We assign it the value '-1' since it is highly unlikely, there will ever be a word with that address:

```
-1 value (Step2)
```

Then we use vectored execution to create a forward declaration for "STEP2":

```
: Step2 (Step2) execute ;
```

Now we can create "STEP1" without a problem:

```
: Step1 1+ dup . cr Step2 ;
```

But "STEP2" does not have a body yet. Of course, you could create a new colon-definition, tick it and assign the execution token to "(STEP2)", but this creates a superfluous word.

It is much neater to use ':NONAME'. ':NONAME' can be used like a normal ':', but it doesn't require a name. Instead, it pushes the execution token of the colon-definition it created on the stack. No, ':NONAME' does *NOT* create a literal expression, but it is just what we need:

```
:noname 1+ dup . cr Step1 ; to (Step2)
```

Now we are ready! We can simply execute the program by calling "STEP1":

```
1 Step1
```

Note that if you run this program, you'll get stack errors! Sorry, but the example has been taken from a Turbo Pascal manual ;).

5.9. This is the end

This is the end of it. If you mastered all we have written about Forth, you may be just as proficient as we are. Or even better. In the meanwhile you may even have acquired a taste for this strange, but elegant language. If you do, there is plenty left for you to learn.

If you find any errors in this primer or just want to make a remark or suggestion, you can contact us by sending an email to:

hansoft@bigfoot.com

We do also have a web-site:

http://hansoft.come.to

You will find there lots of documentation and news on 4tH, our own Forth compiler.

Bibliography

ANSI X3/X3J14 (1993).

Draft proposed American National Standrad for Information Systems — Programming Languages — Forth. Global Engineering Documents, 15 Inverness Way East, Englewood, CO 80122-5704, USA, sixth edition, 1993. Document Number: ANSI/IEEE X3.215-1994.

Leo Brodie (1982).

Starting Forth. Prentice Hall International, second edition, 1982.

Leo Brodie (1984).

Thinking Forth. Prentice Hall International, 1984.

Hans Bezemer / Benjamin Hoyt (1997).

Lookup Tables. Forth Dimensions, Volume XIX, Number 3, September 1997 October.

History

Version	Author	Date	Modification
0.1	Hans Bezemer	2001-03-07	Initial document
0.2	Hans Bezemer	2001-03-11	used 'COMUS' APPEND and changed " to $" in section 'Lookup Tables'

Easy4tH.f

```
\ Easy4tH  V1.0c             A 4tH to ANS Forth interface

\ Typical usage:
\    4096 constant /string-space
\    s" easy4th.f" included

\ This is an ANS Forth program requiring:
\      1. The word NIP in the Core Ext. word set
\      2. The word /STRING in the String word set
\      3. The word D>S in the Double word set
\      4. The words MS and TIME&DATE in the Facility Ext. word set
\      5. The words [IF] and [THEN] in the Tools Ext. word set.

\ (c) Copyright 1997,9 Wil Baden, Hans Bezemer.  Permission is granted by the
\ authors to use this software for any application provided this
\ copyright notice is preserved.


\ Uncomment the next line if REFILL does not function properly
\ : refill query cr true ;

\ 4tH datatypes
: ARRAY CREATE CELLS ALLOT ;
: STRING CREATE CHARS ALLOT ;
: TABLE CREATE ;

\ 4tH constants
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]
```

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
CONSTANT MAX-N                  \ create constant MAX-N
[THEN]

S" STACK-CELLS" ENVIRONMENT?    \ query environment
[IF]                            \ if successful
CONSTANT STACK-CELLS            \ create constant STACK-CELLS
[THEN]

S" /PAD" ENVIRONMENT?           \ query environment
[IF]                            \ if successful
CONSTANT /PAD                   \ create constant /PAD
[THEN]

\ 4tH compiletime words
: [NOT] 0= ;
: [*] * ;
: [+] + ;

\ 4tH wordset
: TH CELLS + ;
: @' @ ;
: COPY ( a b -- b ) >R  DUP C@ 1+ R@ SWAP MOVE  R> ;
: WAIT 1000 * MS ;

: NUMBER              ( a -- n)
  0. ROT DUP 1+ C@ [CHAR] - = >R COUNT R@ IF 1 /STRING THEN >NUMBER NIP 0=
  IF D>S R> IF NEGATE THEN ELSE R> DROP 2DROP (ERROR) THEN
;

\ 4tHs C" runtime semantics emulation

( Reserve  STRING-SPACE  in data-space. )
CREATE STRING-SPACE           /STRING-SPACE CHARS ALLOT
VARIABLE NEXT-STRING          0 NEXT-STRING !

( caddr n addr -- )
: PLACE OVER OVER >R >R CHAR+ SWAP CHARS MOVE R> R> C! ;

( " ccc" -- caddr )
: $" [CHAR] " PARSE
  DUP 1+ NEXT-STRING @ + /STRING-SPACE >
        ABORT" String Space Exhausted. "
  STRING-SPACE NEXT-STRING @ CHARS + >R
  DUP 1+ NEXT-STRING +!
  R@ PLACE
  R>
```

```
;

\ 4tHs Random generator

( Default RNG from the C Standard.  'RAND' has reasonable
( properties, plus the advantage of being widely used. )
VARIABLE RANDSEED

32767 CONSTANT MAX-RAND

: RAND                                  ( -- random )
  RANDSEED @ ( random) 1103515245 *   12345 +   DUP RANDSEED !
  16 RSHIFT   MAX-RAND AND
;

: SRAND ( n -- ) RANDSEED ! ;  1 SRAND

( Don't mumble. )
: random        ( -- n )        RAND ;

: set-random    ( n -- )        SRAND ;

( Mix 'em up. )
: randomize                                         ( -- )
 TIME&DATE  12 * + 31 * + 24 * + 60 * + 60 * +  set-random
;

randomize
```

4 A Web-Server in Forth

1. Introduction

Since I have always given bigFORTH/MINOS-related presentations in the last few years, I'll do something with Gforth this time. Gforth is another tool you can do neat things with, and in contrast to what you here elsewhere, Forth is suitable for almost anything. Even a web-server.

In this age of the "new economy", the Internet is important. Everybody is "in there" except Forth, which hides in the embedded control niche. There isn't any serious reason for that. The following code was created in just a few hours of work and mostly operates on strings. The old prejudice, that Forth was good at biting bits, but has troubles with strings, is thus disproved.

1.1. Motivation

What do you need a web-server for in Forth? Forth is used for measurement and control in remote locations such as the sea-bed or the crater of a volcano. Less remotely, Forth may be used in a refrigerator and, if that stops working, things soon get messy. So a communication thingy is built in.

How much better would it be if instead of "some communication thingy built in", there was a standard protocol. HTTP is accessible from the web-cafe in Mallorca, or from mobile yuppie toys such as PDAs or cell phones. Perhaps one should build such a web-server into each stove and into the bath, so that people can use their cell phone on holidays to check repeatedly (every three minutes?) if they *really* turned their stove off.

Anyway, the customer, boss or whoever buys the product, wants to hear that there is some Internet-thingy build in, especially if one isn't in *e*-Business already. And the costs must be zero too.

But let's take this slowly, step by step.

2. A Web Server, Step by Step

Actually, you had to study the RFC[1]-documents. The RFCs in question are RFC 945 (HTTP/1.0) and RFC 2068 (HTTP/1.1), which both refer to other RFCs. Since these documents alone are much longer than the source code presented below (and reading them would take longer than writing the sources), we will defer that for later. The web server thus won't be 100% RFC conforming (i.e. implement all features), and conforms only as far as necessary for a typical client like Netscape. However additions are easy to achieve.

A typical HTTP-Request looks like this:

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

[1] Request For Comments — the documents of Internet standards are called like that.

(Note the empty line at the end). And the response is

```
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>
...
```

This looks quite trivial, so let's start. The web server should run under Unix/Linux. That takes one problem out of our hands — how we get to our socket — since that's what *inetd*, the Internet daemon, does for us. We only need to tell it on which port our web server expects data, and enter that into the file `/etc/inetd.conf`:

```
# Gforth web server
gforth stream tcp nowait.10000   wwwrun   /usr/users/bernd/bin/httpd
```

We won't replace the default web server just yet (something might not work straight away), so we shall need a new port and that one goes into the file `/etc/services`:

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

When we do a restart or a `killall -HUP inetd` inetd will realize the changes and starts our web server for all requests on port 4444. What we need next is an executable program. Gforth supports scripting with `#!`, as common for scripting languages in Unix. In the line below, the blank is significant:

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

warnings off
```

We better disable any warning. Let's load a small string library (see attachment):

```
include string.fs
```

We shall need a few variables for the URL requested from the server, the arguments, posted arguments, protocol and states.

```
Variable url       \ stores the URL (string)
Variable posted    \ stores arguments of POST (string)
Variable url-args  \ stores arguments in the URL (string)
Variable protocol  \ stores the protocol (string)
Variable data      \ true, when data is returned
Variable active    \ true for POST
Variable command?  \ true in the request line
```

A request consist of two parts, the request line and the header. Spaces are separators. The first word in a line is a "token" indicating the protocol, the rest of the line, or one/two words are parameters.

Since we can process a request only once the whole header has been parsed, we save all the information. Therefore we define two small words which take a word representing the rest of a line and store it in a string variable:

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

As told above, we have header values and request commands. To interpret them, we define two wordlists:

```
wordlist constant values
wordlist constant commands
```

But before we can really start, the URL might contain spaces and other special characters, what to do with them? HTTP advises to transmit these special characters in the form %xx, where xx are two hex digits. We thus must replace these characters in the finished URL:

```
\ 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 ! ;
```

So, that's done. But stop! URLs consist of two parts: path and the optional arguments. Separator is '?'. So first split the string into two parts:

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

So we've defined the basics and can start. Each requests fetches a URL and the protocol, splits the URL into path and arguments and replaces the special character glyphs by the real characters (but those in the arguments remain as we don't yet know what should happen to them). Finally, we must switch over to another vocabulary, since the header follows after the request.

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

So now we can define the commands. According to the RFC, we only need GET and HEAD, POST is then a bonus.

```
commands set-current

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

And now for the header values. Since we need a string variable for each value, and otherwise want only to store the string, we build that with `CREATE-DOES>`. Again: we need a variable *and* a word, which stores the rest of the line there. In two different vocabularies. The latter with a colon behind.

Fortunately, Gforth provides `nextname`, an appropriate tool for this. We construct exactly the string we need and call `VARIABLE` and `CREATE` afterwards

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

And now we set to work and define all the necessary variables:

```
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:
```

There are some more (see RFC), but those are all we need at the moment.

3. Parsing a Request

Now we must parse the request. This should be completely trivial, we could just let the Forth interpreter chew it but for one little caveat:

1. Each line ends with CR LF, while Gforth under Unix expects lines to end with an LF only. We thus must remove the CR. And
2. each header ends with an empty line, not some executable Forth word. We thus must read line for line with `refill`, remove CRs from the line end, and look then if the line was empty.

```
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, the key things are done now. Since we can't let the Forth interpreter loose on the raw input stream *stdin*, we pre-process the stream ourselves. We initialize a few variables which we need to interpret anyway, and steal some code from `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
```

Waiiiit! The request isn't done here. The method `POST`, which was added as bonus, expects the data now. The length fortunately is stored as base 10 number in the field "`Content-Length:`".

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

4. Answer a Request

OK, we've handled a request, and now we must answer. The path of the URL is unfortunately not as we want it: we want to be somehow Apache compatible, i.e. we have a "global document root" and a subdirectory in the home directory of each user, where he can put his personal home page. Thus we can't do anything else but look at the URL again and finally check, if the requested file really is available:

```
Variable htmldir

: rework-htmldir ( addr u -- addr' u' / ior )
  htmldir $!
  htmldir $@ 1 min s" ~" 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 $@ ;
```

Next, we must decide how the client should render the file — i.e. which MIME type it has. The file suffix is all we need to decide, so we extract it next.

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

Normally, we'd transfer the file as is to the client (transparent). Then you tell the client how long the file is (otherwise, we'd have to close the connection after each request). We open a file, find its size and report that to the 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 ;
```

We do all the work with `transparent`, using `TYPE` to send the file in chunks to support "keep-alive" connections, which modern web browsers prefer. The creation of a new connection is significantly more "expensive" than to continue with an established one. We benefit on our side also, since starting Gforth again isn't for free either. If the connection is keep-alive, we return that, reduce maxnum by one, and report to the client how often he may issue further requests. When it's the last request, or no further are pending, we send that back, too.

```
: .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 ;
```

Now we just need some means to recognise MIME file suffixes and sned the appropriate transmissions. For the response, we must also first send a header. We build it from back to front here, since the top definitions add their stuff ahead. To make the association between file suffixes and MIME types easy, we simply define one word per suffix. That gets the MIME type as string. `transparent:` does all that for all the file types that are handled using `transparent`:

```
: 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 ;
```

There are hundreds of MIME types, but who wants to enter all of them? Nothing could be easier than this, we steal the MIME types that are already known to the system, say from `/etc/mime.types`. The file lists the mime type on the left paired with the file suffixes on the right (sometimes none).

```
: 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 ;
```

One more thing we need: for active content we want to use server side scripting (in Forth, of course). Since we don't know the size of these requests in advance, we don't report it but close the connection instead. That relieves us of the problem of cleaning up the trash the user is creating with his active content (that's Forth code!).

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

So let's start with the definition of MIME types. Get a new wordlist. Active content ends with `shtml` and is `included`. We provide a few special types and the rest we get from the system file mentioned above. For unknown file types, we need a default type, `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. Error Reports

Sometimes a request goes wrong. We must be prepared for that and respond with an appropriate error message to the client. The client wants to know which protocol we speak, what happened (or if everything is OK), who we are, and in the error case, a error report in plain text (coded in HTML) would be nice:

```
: .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 Definitions

We are almost done now. We simply glue together all the pieces above to process a request in sequence — first fetch the input, then transform the URL, recognize the MIME type, work on it including error exits and default paths. We need to flush the output, so that the next request doesn't stall. And do that all over again times, until we reach the last request.

```
: 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 ;
```

To make Gforth run that at the start, we patch the boot message and then save the result as a new system image.

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

7. Scripting

As a special bonus, we can process active content. That's really simple: We just write our HTML file as usual and indicate the Forth code with "¡$" and "$¿ " (the space for the closing parenthesis is certainly intentional!). Let's define two words, `$>`, and to get the whole thing started, `<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>" $> ;
```

That's quite enough, we don't need more. The rest is all done by Forth, as in the following example:

```
<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. Outlook

That was a few hundred lines of code — far too much. I have delivered an "almost" complete Apache clone. That won't be necessary for the sea-bed or the refrigerator. Error handling is ballast, too. And if you restrict to single connection (performance isn't the goal), you can ignore all the protocol variables. One MIME type (text/html) is sufficient – we keep the images on another server. There is some hope that one can get a working HTTP protocol with server-side scripting in one screen.

9. Appendix: String Functions

Certainly we need some string functions, it doesn't work without. The following string library stores strings in ordinary variables, which then contain a pointer to a counted string stored allocated from the heap. Instead of a count byte, there's a whole count cell, sufficient for all normal use. The string library originates from bigFORTH and I've ported it to Gforth (ANS Forth). But now we consider the details of the functions. First we need two words bigFORTH already provides:

```
: 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` deletes the first n bytes from a buffer and fills the rest at the end with blanks.

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

`insert` inserts as string at the front of a buffer. The remaining bytes are moved on.

Now we can really start:

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

To avoid exhausting our memory management, there are only certain string sizes; $padding takes care of rounding up to multiplies of four cells.

```
: $! ( 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! ;
```

`$!` stores a string at an address. If there was a string in before, this string will be released.

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

`$@` returns the stored string.

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

`$@len` returns just the length of a string.

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

`$!len` changes the length of a string. Therefore we must change the memory area and adjust address and count cell as well.

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

`$del` deletes u bytes from a string with offset off.

```
: $ins ( addr1 u addr2 off -- ) >r
  2dup dup $@len rot + swap $!len  $@ 1+ r> /string insert ;
```

`$ins` inserts a string at offset off.

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

`$+!` appends a string to another.

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

`$off` releases a string.

As a bonus there are functions to split strings up.

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

`$split` divides a string into two, with one char as separator (e.g. '? for arguments)

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

`$iter` takes a string apart piece for piece, also with a character as separator. For each part a passed token will be called. With this you can take apart arguments — separated with '& — at ease.

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 *addr*1 + *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 OS Interface

This chapter explains how to interface to OS libraries. It will also give descriptions of important OS functions where OS library bindings are already available for bigFORTH, and provide a tutorial to implement your own library bindings.

1. Library Bindings

Modern operating systems provide shared libraries or "dynamic link libraries". These libraries are loaded at run-time, and provide functions to resolve symbols. Most libraries use standard calling conventions, like C calling conventions, or, in case of Windows, Pascal calling conventions.

LIBRARY (−−) ⟨*name*⟩ ⟨*library*⟩:⟨*name*⟩ (−−) ⟨*binding*⟩:⟨*binding*⟩ (args −− ret): Creates a word ⟨*name*⟩ to access ⟨*library*⟩. ⟨*name*⟩ creates bindings to library functions.

DEPENDS (−−) ⟨*library*⟩: Adds library dependencies. Dependent libraries are loaded first.

INT (−−) immediate restrict: Specifies an integer parameter in the library binding.

INTS (n −−) immediate restrict: Specifies **n** integer parameters in the library binding. **n** is on the compilation stack.

SF (−−) immediate restrict: Specifies a single float parameter in the library binding.

DF (−−) immediate restrict: Specifies a double float parameter in the library binding.

<REV> (−−) immediate restrict: Indicates that the parameter order is reverted from C order (first argument in C calling convention as top of stack). Must be first in library binding declaration.

(VOID) (−−) ⟨*symbol*⟩ immediate restrict: Specifies no return value, and finishs library binding definition by creating a binding structure for ⟨*symbol*⟩.

(INT) (−−) ⟨*symbol*⟩ immediate restrict: Specifies integer return value, and finishs library binding definition by creating a binding structure for ⟨*symbol*⟩.

(FP) (−−) ⟨*symbol*⟩ immediate restrict: Specifies floating point return value, and finishs library binding definition by creating a binding structure for ⟨*symbol*⟩.

(INT/FP) (−−) ⟨*symbol*⟩ immediate restrict: Specifies integer return value and tells that the floating point stack might be used inside the library function, and finishs library binding definition by creating a binding structure for ⟨*symbol*⟩.

(VOID/FP) (−−) ⟨*symbol*⟩ immediate restrict: Specifies no return value and tells that the floating point stack might be used inside the library function, and finishs library binding definition by creating a binding structure for ⟨*symbol*⟩.

1.1. Internal Words

@LIB (addr −−): Loads a library and stores the library handle in the library access word's body **addr**

@PROC (lib addr −−): Resolves a library binding. The symbol name stored in the library binding structure at **addr** is searched in **lib**, and the resulting address is patched in the library binding structure **addr**.

@SYMS (lib −−): Resolves all symbols in the library **lib**.

@LIBS (−−): Prepares all libraries. All library addresses are replaced by calls to a function that loads that library and resolves all symbols when a library binding is called.

GETLIB (addr len −− lib/0): Loads a library by name **addr len** and returns the library handle, 0 if unsuccessful.

PROCADDR (addr len lib −− addr/0): Searches a symbol by name **addr len** in the library **lib**, and returns the function address, 0 if unsuccessful.

LEGACY (−− addr): Variable: switches new/old method of defining libraries. 0 means "new", -1 means "old" with reverted parameter order, 1 means "old" with parameter order as new bindings. In "old" mode, a library binding definition takes the number of interger variables from the stack, and inserts something equal to `ints (int)` before the symbol name.

9 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”).

10 Objekt Oriented FORTH

1. What's Object Oriented Programming?

The buzzword of the late 80s and early 90s in the IT industry is without doubt "object oriented". No operating system, no application, and certainly no programming language, that isn't object oriented. Forth isn't excluded as publications like Dick Pountain's "Object–Oriented FORTH" [1] show clearly.

Ewald Rieger had ported Pountain's OOF to bigFORTH and gave it to me on the Forth–Tagung '91 to look at it. Since this OOF lacked several features, I completely rewrote it, to make this interesting programming paradigma available for bigFORTH. This system is in use since 1992, and has proved to be useful even in the rough world of real–time programming (Ewald Rieger uses it to control an automatic chromatography system).

The capsule of data and algorithm to form an object has shown its usefullness — or more proven essential — especially for changing hardware configurations, as they are typical for many tasks in praxis.

To give you an impression, what's possible with object oriented programming, and how to do it, the following is an introduction using a small example.

The sources are in the file `oofsampl.scr`. As example I use a small collection of data types that are known from computer science: integer, lists, arrays, and pointers.

Object oriented programming hides behind a slang, that — after a closer look — has quite some similatities with known concepts like modularity and definition of clean interfaces.

1.1. The Class Concept

The core principle of object oriented programming is to capsule data and the procedures that operate on the data into an *object*. Ideally these procudures ("methods") of an object have the exclusive access to its data and are therefore the only way to operate on the data. Interface to the object then are the names of the methods ("messages") and parameters that are sent with the message. Since many objects return results, the stack is used conventionally for the "message passing" and the method is called like a Forth–word — with a detour over the object, that manages the capsuling.

Now it would be quite some waste to program methods for each single object; especially since many objects have the same or a similar structure and are only distinct by the data. Such similar objects are summed up as a "class". A class is so to speak a template (or a form) for an object; after "instanciation" a real object is created out of a class, with space for data, and the methods common for all objects of a class.

Often you need only some small modifications to a class to obtain a new one, and therefore you use "inheritance" to create a sublcass — a derivative. Additional variables and modified or new methods are just appended to the class.

All objects of a class and of all subclasses have a common message protocoll, thus they understand the same messages and react similar. The differences in detail are called "polymorphism". So may e. g. each graphical object have a method "draw you", but one object may draw a circle, the other a point or a rectangle.

If you emphasize the common protocoll to all objects of a class hierarchy, you create the protocoll separated from the implementation of the subclasses and call the create class — that contains only protocoll, but no implementation — an "abstract data type". Such a data type is the class `data` presented in the following listing:

```
\ 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;
```

Here I must add, that in bigFORTH all classes are originated finally from the same parent class, the class `object`. Also, classes and objects aren't only used for operating on the data, but also for creating new subclasses and instanciation of the objects. Therefore a class is just an object without the data area, that can create new subclasses using the method `class`.

The description of a class consists of two parts: a declaration of variables and mehtods, and the implementation of the methods. All variables, all polymorph and externally accessible methods must be declaired; helpe methods could be declared optionally. In the implementation part, undeclaired methods are automatically declaired as EARLY (private).

The example creates a cell sized variable called `ref`, in the private area of the class that isn't visible from outside, but can be inherited (thus corresponds to `protected:` in C++). `public:`, thus publically available are the six methods `!`, `@`, `.`, `null`, `atom?`, and `#`, which are used to store, read, and display the value, to create "null" object, the query whether the object is atomic or composed, and the number of sub objects if the latter is the case.

The last three methods are already implemented, since they are the same for all simple objects. The not implemented methods can't be executed, more precise, they lead to `abort"`. They must be implemented in real data types, like in the following data type 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<= IF  super dispose  THEN ;
        : null  ( -- addr )  0 new ;
```

```
class;
```

Here I create a new class in the same manner and allocate a (private) variable `value`. The two methods store and fetch (`!` and `@`) access `value` — as interface completely sufficient. The `F` before the words changes the interpretation from the object vocabulary to the normal vocabulary, thus the normal words known as `!` and `@` are used. The method `.` is easy to understand, too. Here the `@` however is interpreted as access method.

I must say a few words about the methdos `init` and `dispose`: `init` is called at creation of the object and is used to initialize it. Here in the example I initialize `value` with a number on the stack. `dispose` removes an object from the dynamic memory managemnt. If you modify this method, you must (unlike in C++) explicitely call the dispose method of the parent class (with `super dispose`). I dispose only, if the reference counter is zero or negative, thus there are no further references. Otherwise, the reference counter is just decremented.

`null` now has the meaning as expected: it creates an object with the value 0 (dynamically) and leaves its address on the stack. `new` is, like `dispose`, a method. Without additional informations (thus without class or object), the method of the current class is used.

1.2. Binding: Late or Early?

Let's take a step back and look at how methods are called. How much must be defined at compile time, and what has to be resolved at runtime (and then should not produce an error)?

As long as it is clear which method of which class is executed, as with `super dispose`, it will be resolved at compile time and create a dicrect call to the specified method. This is called "early binding". It's for sure the fastest method, but unfortunately doesn't allow polymorphism.

Often enough it's not clear in advance, which subclass the object belongs to, when you want to send a method to it. You have to find out the address of the method at runtime, then. A search in the dictionary is prohibitive inefficient, as well as a (possibly even sequential) search over a numerical key isn't what I call run time efficience.

In bigFORTH therefore each object contains a pointer to a jump table as first element, the jump table contains the addresses of all methods. This doesn't only guarantee a response time independent of the number of methods (after all, Forth should still remain a real time language), it is also quite fast. Especially, since this approach is so easy to code, that a simple macro can be directly inserted in the caller's code.

What's prevented — or at least made much more difficult — with this approach is "multiple inheritance". Crossing a new child class from several parent classes is problematic, anyway: methods and variables with the same names must be renamed, if they aren't inherited from the same grandparent class, and the offsets of variables in the object change (so must also be determined at runtime). Or the compiler must ensure in advance, that the necessary space of the mixed class is already reserved in the parent classes (a space-time tradeoff, that can't be done with a one-pass compiler, and creates difficulties even in complex systems).

A very important aspect are the real time properties of the created code. Thus the run times must be known to the programmer; this only leaves completely deterministic approaches for bindings. Only then the programmer doesn't lose control of what he writes. C++ doesn't have this property, and therefore is only of limited use for real time tasks.

1.3. Objects as Instance Variables

Whatsoever, quite often a straight-forward class hierarchy is good enough. Appropriate abstract data types allows to circumvent real multiple inheritance. In case of an emergency, you can reach a sort of "multiple inheritance" by copying the sources.

What's necessary though is to have objects as instance variables in another objects, as well as pointer and directly. This can be shown using the lists as example, since they need pointers to be implemented:

```
\ 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
```

Here, we first create an abstract data class for lists; this needs as both pointer to first and rest of the list as data. Since both may normal data, also "dot pairs" are allowed — like in Lisp. Would the rest of the list have been a list again, the type isn't necessary; that creates a pointer to the object of the current declaired class. `lists ptr next` won't work, sincd the class `lists` isn't completely defined at this point and therefore can't be executed.

Additionally to these pointers you need certainly also a few methods: a list could be empty, so you should be able to ask for that. Also, it would be quite useful to display the first element (with `?`).

A null–list is the empty list, also called "nil". Since this is a list, it must be declaired later, therefore I create a forward reference, which is resolved with the later definition of `nil`.

Empty lists differ from ordinary quite significantly. They always return true to `empty?`, there's only one of them, and this one certainly may not deleted. It displays a pair of parenthesis.

Now I create an element of the class of empty lists, and the address of this element (put on the stack with the method `self`) finally is called `nil`. Both the first as the next element of the empty list is again the empty list. That prevents crashes when a program runs over the end of the list.

The method `bind` allows to bind object references to an object pointer. The object pointer `first` of the object (`nil` behaves, after been bound, exactly like the object that it is bound to, thus (`nil` itself. This is more interesting with real lists:

```
\ 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;
```

A linked list certainly isn't empty. On creation, I bind the references `first` and `next`; appropriate object addresses must have been put on the stack. At binding, I increment the reference counters of the objects — now another pointer points to them. To make them current object, I push them on the object stack, thereby with `ref` *their* reference counter is addressed, and not the one of the list. The object stack isn't a real stack; only the topmost element is put into a register, the rest is on the return stack.

The methods `@`, `!`, and `?` refer to the first element of the list; they are only passed through. No complex pointer arithmetic is necessary, the name of the reference is sufficient.

To print a list, I must walk through the list. Before the first element, I open a parenthesis, otherwise the elements are separated by blanks. The current first element of the list is displayed. Is the next element an atom, it must be printed as dot pair; the list ends then. The list also ends when the next element is the empty list. Afterwards, only the parenthesis has to be closed, and the blank is dropped from the stack.

Surprising is the recursion in `#`, which competes the length of a list. It simply computes the lenght of the rest of the list, increments the result and and finishs. As soon as the list terminates with `nil` or an atom, that definitely has the length 0, the recursion terminates. Here you first see a clear advantage of object oriented programming, that makes lots of `IF..ELSE..THEN` for decisions unnecessary and therefore eases recursions.

At deletion of a list, both parts of the list and the node itself have to be deleted. Here also no case decision is necessary, and the termination question, that is often forgotten in recursive programs, isn't asked here.

Now we just need element objects for the list. We already have numbers, but strings would be nice, too. Here they are:

```
\ 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;
```

We derive the class `string` from `int`. I use it's instance variable `value` as handle, as pointer to a movable memory area. There, the string is stored as counted string. When storing a new string, the size of the memory block must be adusted; at the first time, it must be allocated, and freed at deletion. All the rest is self-explaining, I hope.

Very useful is the pointer class. You can directly create pointer variables, but you can't insert them into e. g. a list.

```
\ 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;
```

Analoguous to the list I create a pointer instance variable (`pointer`); then there's the method `ptr!`, which is used to assign a new object. The methods `@`, `!`, `.`, and `#` are fed through to the container. The init method binds a passed object to the pointer. `ptr!` first releases the previous object, and afterwards stores the new object. I certainly care about reference counting here.

Deletion of a pointer object also means that one pointer less points to the object (and it eventually has to be deleted); afterwards, the pointer is deleted.

Analoguous to a pointer you can create a whole array of pointers:

```
\ 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;
```

Similary to the method `new` you create a new array of objects with `new[]` — which contains elements of this class. The array really is an array of pointers, you can assign other objects at any position of the array with `bind[]`. The array index for accessing an array variable is expected on the stack.

1.4. Tools and Application Examples

The list packet still isn't very easy to use. I've written a few small tools that eases the use — but certainly won't make up a complete Lisp or something like that out of it:

```
\ 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` and `list` help to create a list. `cons` concatenates two objects on the stack to a list (TOS as next, thus should be a list; NOS as first element of the list). `list` takes an object and together with nil creates a list out of it.

`car` and `cdr` should be known from Lisp; they return first resp. rest of the list.

`print` calls the output method of an object.

`ddrop` finally removes and deletes an object.

`make-string` is the string constructor, analog to `list`. `$"` constructs a string constant. As example how to create a list with these tools:

```
$" 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. The Complete Language Description

2.1. Semantics of the Object Interface

The interface to object oriented programming in bigFORTH divides into three parts:

- Tools to manage objects, which are themselves not object related, and the classes from which all other classes and objects are derived from.
- Tools to create instance variables and methods
- and methods of the root class, to create new classes, instances, handling of object pointers and similar things.

Only the words of the first item are directly accessible from the FORTH vocabulary. The words of the second item are only available during declaration of a class, and the words of the third item aren't really words, but methods of objects.

bigFORTH uses a very consequent way to manage classes: classes are objects, although with class global instance variables, that are just used to create and manage new classes and objects. Classes are also used to send messages to objects which address is stored in an object pointer, and that need expicit context (because it's not the current defined object), thus are also used as a sort of type casting.

ˆ (−− o): Returns the pointer **o** to the current object

>O (o −−) (OS −− o): Moves the pointer to object **o** to the object stack. The object thereby becomes the current object. **Attention:** the previously used object is pushed on the return stack, object stack accesses therefore must be balanced with other return stack accesses like DO LOOPs and >R and R>.

O> (−−) (OS o −−): Pops the pointer to the current object from the object stack. The previously used object is restored from the return stack and becomes the current object.

O@ (−− addr): Returns the address of the method table of the current object.

BIND (o −−) ⟨*name*⟩: Binds the object **o** to the object pointer ⟨***name***⟩. **o** must be derived of the class ⟨***name***⟩ or a subclass thereof.

DYNAMIC (−−): Dynamic object creation in the heap on NEW. That's the default behaviour.

STATIC (−−): Static object creation in the dictionary on NEW. You can compile object structures, store them with SAVESYSTEM and reuse them after load, as long as the objects themselves don't use other functions to allocate dynamic memory.

Each object consists of variables and methods, that have to be declaired. The methods afterwords need to be implemented, too. Visibility of variables and objects to the outside is selected: private methods and variables are only visible inside the class and subclasses, external visible method and variables have to be declaired "public".

bigFORTH separates declaration and implementation of classes. Both together form the definition of a class.

TYPES (−−) (VS voc −− TYPES): All the words that are used to declare classes and implement methods are in the vocabulary TYPES. TYPES must be topmost of the search order during class definition, since otherwise conflicts would arise (: e. g. is defined as well in TYPES, the current PUBLIC thread, and in FORTH).

PUBLIC: (−−): Switches to public declaration. All further methods and variables are visible interfaces to the object.

VAR (size −−) ⟨*name*⟩: Creates an instance variable of **size** bytes length.

STATIC (−−) ⟨*name*⟩: Creates a variable that is common to all the objects of a class. This variable is cell sized and created uninitialized as pointer.

METHOD (−−) ⟨*name*⟩: Declaires a method. Methods declaired like this are late bound, if it's not specified in the context which class is used.

EARLY (−−) ⟨*name*⟩: Declaires an early bound method. You can't change such a method in a subclass, if you want to use the same name again, you have to declair the early method again.

DEFER (−−) ⟨*name*⟩: Declaires an object specific method, that can execute object specific actions. Execution tokens are assigned with IS. This is e. g. useful to assign callbacks.

PTR (−−) ⟨*name*⟩: Declaires an object pointer, which must point to the currently declaired class or a subclass, and is initialized with BIND.

ASPTR (class −−) ⟨*name*⟩: "casts" a pointer created with PTR to the currently declaired class, and declaires the casted pointer as ⟨***name***⟩.

F (−−) ⟨*name*⟩: Compiles ⟨***name***⟩ with FORTH as first vocabulary in the search path.

HOW: (−−): Changes from declaration to implementation part. In this part, you initialize static variables, and implement methods.

: (−−) ⟨*name*⟩: Implements the method ⟨***name***⟩. You end the implementation with ;.

CLASS; (−−): Ends the implementation of a class.

The management of classes and objects is task of the classes themselves. Therefore, the root class OBJECT provides some methods and class global variables. These can be separated into the following groups:

- Class browser
- Subclass creation
- Memory management, instance creation
- Binding

Class **OBJECT (... −− ...) ⟨*method*⟩:** Is the root of all object classes. Executes ⟨***method***⟩ resp. compiles it dynamically bound in the context of the current object.

Public:

Static Variable **PARENTO (−− addr):** Points to the parent class

Static Variable **CHILDO (−− addr):** Points to the last derived subclass

Static Variable **NEXTO (−− addr):** Points to the next old subclass of the parent class

Static Variable **NEWLINK (−− addr):** Points to a list of all sub-objects which consume memory in the object, and is used for the internal memory management.

Early Method **CLASS (−−) ⟨*class*⟩:** Starts declaration of a subclass

Early Method **CLASS? (object −− flag):** Checks class relationship. **flag** is only true, when **object** is in the upward derivation chain of the object that executes CLASS?.

EARLY METHOD **SEAL (−−):** Hides the private methods and variables and hinders further use after inheritance.

METHOD **INIT (... −−):** Initializes an object with the parameters **...**. INIT is called also for all objects that are declaired as instance variables, in the order of declaration, but first for the main object. INIT is a polymorph method.

EARLY METHOD **NEW (−− object) immediate:** Creates a (nameless) object of the current class

EARLY METHOD **NEW[] (n −− object) immediate:** Creates an array of (nameless) objects of the current class with **n** elements.

METHOD **DISPOSE (−−):** Frees the object's memory space. DISPOSE is a polymorph method.

EARLY METHOD **: (−−)** ⟨***name***⟩**:** Creates an object under the name ⟨***name***⟩.

EARLY METHOD **PTR (−−)** ⟨***name***⟩**:** Creates a pointer to an object of the current class (or subclass) under the name ⟨***name***⟩. The pointer is empty at first, use BIND to assign an object to the pointer.

EARLY METHOD **ASPTR (class −−)** ⟨***name***⟩**:** "casts" a pointer created with PTR to the current class and creates the casted pointer under the name ⟨***name***⟩.

EARLY METHOD **[] (n −−)** ⟨***name***⟩**:** Creates an array of objects with **n** elements under the name ⟨***name***⟩.

EARLY METHOD **:: (−−)** ⟨***method***⟩ **immediate:** Binds ⟨***method***⟩ of the current class early.

Invoked directly in the implementation part of a class, it inherits methods from other classes, thus allows a limited multiple inheritance. The method must be defined in a common parent class, only the code address of the method is inherited.

EARLY METHOD **SUPER (−−)** ⟨***method***⟩ **immediate restrict:** Binds ⟨***method***⟩ of the parent class early. SUPER is used to modify inherited behaviour, and to access the original behaviour in the modified method. You can use SUPER repeatedly to access methods higher up in the inheritance chain.

EARLY METHOD **GOTO (−−)** ⟨***method***⟩ **immediate restrict:** Is used for end recursions. The method ⟨***method***⟩ is called directly, without pushing a return address (or evtl. the old object class) onto the return stack.

EARLY METHOD **SELF (−− addr):** Returns the address of the object.

EARLY METHOD **BIND (object −−)** ⟨***pointer***⟩ **immediate:** Stores the address **object** in the pointer variable ⟨***pointer***⟩. **object** must belong to the pointer's class or a subclass thereof.

EARLY METHOD **LINK (−− addr)** ⟨***name***⟩**:** Creates a reference to the object pointer ⟨***name***⟩, thus an address where object pointers can be stored with !

A few variables aren't accessible from outside, but can be used in subclasses for debugging purposes:

PRIVATE:

STATIC VARIABLE **PUBLIC (−− addr):** Points to the wordlist of all public variables and methods.

STATIC VARIABLE **PRIVATE (−− addr):** Points to the wordlist of all private variables and methods.

STATIC VARIABLE **METHOD# (−− addr):** Number of the methods and static variables in bytes.

STATIC VARIABLE **SIZE (−− addr):** Number of bytes used as dynamic memory

VARIABLE **OBLINK (−− addr):** First instance variable: points to the method table of an object

For debugging porposes, there is the object DEBUGGING. It contains further methods, which are helpful for debugging.

CLASS **DEBUGGING (... −− ...)** ⟨***method***⟩**:** Is a helper class, that provides necessary tools to debug objects. Otherwise like OBJECT.

PUBLIC:

EARLY METHOD **WORDS (−−):** Lists all the words in the public and private vocabulary

EARLY METHOD **' (−− cfa)** ⟨***name***⟩**:** Finds the **cfa** of a method or object variable

EARLY METHOD **SEE (−−)** ⟨***name***⟩**:** Decompiles ⟨***Name***⟩

EARLY METHOD **VIEW (−−)** ⟨***name***⟩**:** Opens the editor on the declaration of ⟨***Name***⟩

EARLY METHOD **TRACE' (.. −− ..)** ⟨***name***⟩**:** Traces the method ⟨***Name***⟩

EARLY METHOD **DEBUG (−−)** ⟨***name***⟩**:**

2.2. Formal 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*⟩ |

11 MINOΣ Documentation

1. What is MINOΣ?

MINOΣ is the answer to the question "Is there something like Visual BASIC (Microsoft) or Delphi (Inprise) in Forth?"—and the answer is "yes". Basically, these GUI RADs contain two components: a library with a wide variety of elements for a graphical user interface; e. g. windows, buttons, edit-controls, drawing areas, etc.; and an editor to combine the elements with the mouse by drag&drop or click&point actions. You then insert code that acts on buttons being pressed.

Typical applications are often related to data base access. Therefore, many of these systems already contain a data base engine or at least a standardized interface to a data base, such as ODBC.

Another aspect are complex components. With some of these toolkits, you can create a web browser with some mouse clicks and a few keystrokes. However, these components hide their details, a shrink wrapped web browser application is not necessarily worse.

The interactivity of these tools usually is not very high. You create your form, write your actions as code and compile it more (Delphi) or less (Visual Age for C++) fast. Trying it usually isn't possible before the compiler run.

1.1. WYSIWYD—What You See Is What You Do

It isn't really necessary to brush graphical user interfaces together, as it isn't to edit texts WYSIWYG. Many typesetting functions are more semantically than visual, e. g. a text is a headline or emphasized instead of written in bold 18 point Garamond or 11 point Roman italics. All this is true for user interfaces, to some extend much more. It's not the programmer that decides which font and size to use for the UI—that's up to the user. As is color of buttons and texts.

Also to layout individual widgets, more abstraction than defining position, width and height makes sense. Typically buttons are arranged horizontally or vertically, perhaps with a bit distance between them. The size of buttons must follow the containing strings, and should conform to aesthetics (e. g. each button in a row has the same width).

Such an abstract model, related to TEX's boxes&glues, programs quite good even without a visual editor. The programmer isn't responsible for "typesetting" the buttons and boxes. This approach is quite usual in Unix. Motif and Tcl/Tk use neighborhood relations, Interviews uses boxes&glues. I decided for boxes&glues, since it's a fast and intuitive solution, although it needs more objects to get the same result.

These concepts contradict somehow with a graphical editing process, since the editors I know don't provide abstract concepts ("place left of an object" or "place in a row"), but positions.

1.2. Visual Forth—The Man Month Myth

One point makes me think: the packets that allow real visual form programming have many years of programming invested in. Microsoft, Borland, and IBM may hire hundreds of programmers just for one such project. This man-power isn't available for any Forth project. But stop:

1. Forth claims that good programmers can work much more efficient with Forth

2. A team of 300 (wo)men blocks itself. If the boss partitions the work, the programmers need to document functions, and to read documents from other programmer related to other functions and must understand them, or ask questions to figure things out. Everybody knows that documenting takes much longer than writing the code, and explaining is even worse. Thus at a certain project complexity level, no time is left for the programming task; all time is used to specify planned functions and read the specification from other programmers. Or the programmers just chat before the door holes of the much too small and noisy cubicles.

3. A good programmer reportedly works 20 times as fast as a bad, even though he can't type in more key strokes per time. The resulting program is either up to 20 times shorter or has 20 times less bugs (or both)—with more functionality at the same time. Teamwork however prevents good programmers from work, since they are frustrated by bad programmers surrounding them, from their inability to produce required information in time; and the bad programmers are frustrated by the good ones, which makes them even worse.

4. Therefore, even in large projects, the real work is (or should be) done by a small "core team". Then the Dilbert rule applies: what can be done with two people, can be done with one at half of the costs.

Furthermore, bigFORTH-DOS already contains a "Text-GUI", without graphical editor, but with an abstract boxes&glue concept, which, as claimed above, hinders the use of such an editor.

Finally I wanted to get rid of DOS, and port bigFORTH to a real operating system (Linux). In contrast to Windows and OS/2, user interface and screen access are separated there. Drawing on the screen uses the X Window System (short X), the actual user interface is implemented in a library. This is the reason, why there is no common interface, but a lot of different libraries, such as Athena Widgets, Motif, Tcl/Tk, xforms, Qt, gtk, and others. The "look and feel" from Motif-like buttons is quite common, even Windows and MacOS resemble it.

All these libraries have disadvantages. The Athena Widgets are hopelessly outdated. Motif is commercial, even if a free clone (Lesstif) is in creation. It's slow and a memory hog. Tcl/Tk consumes less memory, but it's *even* slower. How do you explain your users that drawing a window takes seconds, while Quake renders animated 3D-graphic on the same machine? Qt is fast, but it's written in C++ and doesn't have a foreign language interface now. gtk, the GIMP toolkit, has more foreign language interfaces, and it's free, but it wasn't available until recently.

Therefore I decided to port the widget classes from bigFORTH-DOS to X, and write an editor for it. Such classes written in Forth naturally fit in an development environment and are—from the Forth point of view—easier to maintain. There are not such many widget libraries in C, because it's a task written in an afternoon, but because the available didn't fit the requests, and a modification looked desperate.

1.3. Codename MINOΣ—Where the Name Came From

"Visual XXX" is an all day's name, and it's too much of a microsoftism for me. "Forth" is a no-word, especially since the future market consists of one billion Chinese, and for

them four is a number of unluck (because "se" (four) sounds much like "se" (death)). However, even Borland doesn't call their system "Visual TurboPascal", but "Delphi".

Greek is good, anyway, since this library relates to the boxes&glues model of TEX, which is pronounced Greek, too. Compared with Motif, the library is quite copact (MINimal), and since it's mainly for Linux, the phonetic distance is small... I pronounce it Greek: "meenoz".

1.4. Windoze—Porting Notes

I ported MINOΣ to Windows 95/NT, on the demand of some potential users. It doesn't run near as stable as under Linux/X, since there are a hideous number of subtle bugs in Windows, and I don't have the time to work around all of them. Drawing polygons doesn't work as well as on X, and all the bugs that are in the memory drawing device can drive me nuts. The Windows port of MINOΣ looks more like the "modern Forth" Claus Vogt portrayed in news:de.comp.lang.forth: it shows random general protection faults. Well, just like any other Windows program.

2. Theseus

2.1. Introduction

Theseus is designed to create **MINOΣ** dialogs. Each dialog is derived form a window class. You can create dialogs by composing boxes (layout managers) and widgets together, glue actions to the widgets, add variables and methods to the derived window classes, and so on.

This is the help system, called "Ariadne", since it is the red line that leads you through the labyrinth of MINOΣ. This chapter will give some informations about the philosophy behind MINOΣ and Theseus, so you understand what you are doing.

MINOΣ has four sets of classes:

Widgets: "window gadgets", these are the basic objects like buttons, labels, icons and text fields.

Displays: these widgets are the drawing area of other widgets. They know how to draw to X or to the parent display. It's also possible to create new displays that e.g. draw into an OpenGL widget or send drawing events over the network, to copy drawing messages to more than one parent display and so on.

Boxes: or layout managers. Although each object in MINOΣ has coordinates and size, it's formating is done by a surrounding layout manager. These either form horizontal boxes, and align their contents from left to right, or vertical boxes, and align their contents from top to bottom. Each widget has a horizontal and a vertical glue, these glue values are used to compute a nice look of the widgets.

Actions: these are the objects that link data and widgets together. An action knows in which state it is, and what to do when the state changes. There are several types of actions, for simpe buttons, toggle buttons, text fields, and sliders.

Each dialog is hierarchically composed out of widgets, boxes, and displays (e.g. viewports). Each active object, thus each button, toggle button, slider, and text field has an associated action, a name, and other attributes.

The next chapter will describe how to use Theseus, and which classes and messages are available.

2.2. Onscreen Fundamentals

Menu

The menu bar contains a file menu, an edit menu and a help menu.

Widget bar

The widget bar contains groups of widgets. To insert a widget into a dialog, select the group and click on a button. The corresponding widget is inserted in the current position (affected by the mode switches).

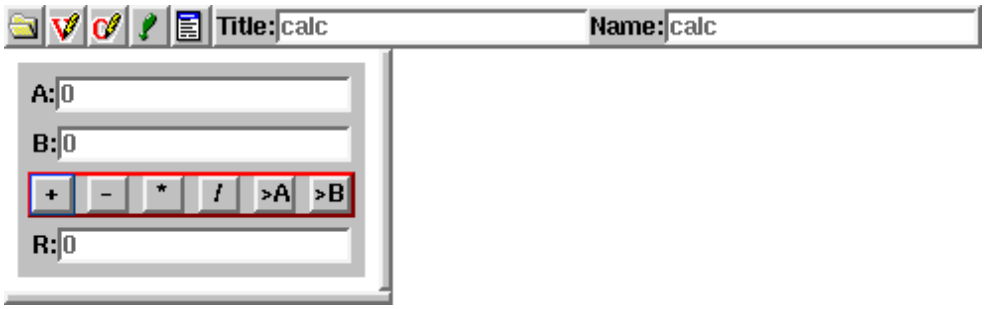
Editing modes

The editing modes are separated into three groups:

- Insert modes:
  - Add object first in current box
  - Add object last in current box
  - Add object before current box
  - Add object after current box
- Navigation, selects the active box. Note that there is an active widget (which you can select with the mouse pointer).
  - Move one box up in hierarchy
  - Move one box to the left/up
  - Move one box to the right/down
  - Move to the first child

- Short cuts:
  - Load dialog
  - Save dialog
  - Try dialog
  - Save as module

Dialog editor



The dialog editor shows a navigation bar for each dialog to edit. The dialog itself is resizable with the split bar below and on the side, to see how it looks resized.

The navigation bar consists of an icon to open/hide the dialog, to open/hide the declarations field, to open/hide the code field, to select whether to show the dialog, a dialog menu, a dialog title and a dialog name. It is important to give every dialog a name, since dialogs without names can't be saved. The dialog name is the name of the derived class, you can refer to this class in your code.

The declaration field contains variable and method declarations of the dialog class.

The code field contains method definitions.

The dialog edit field itself contains the dialog itself.

Click modes:

Edit clicking opens the inspector of that object, with focus on the text/code/name field (left/middle/right mouse button).

Cut&Paste ⟨⇧⟩⟨mouse⟩ clicking left cuts the object to the cut stack, clicking middle or right pastes from the cut stack.

Try ⟨Ctrl⟩⟨mouse⟩ clicking makes the object react as in the dialog, but without executing code.

Box creator

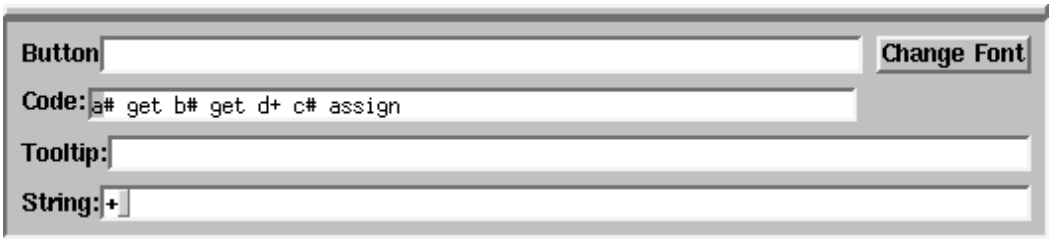
The box creator has two buttons to create horizontal and vertical boxes. Boxes are simple layout managers, that arrange containing objects one after the other. Boxes are created as normal objects, so they go to the same places where a normal object would go. They inherent the settings of the parent object, 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.

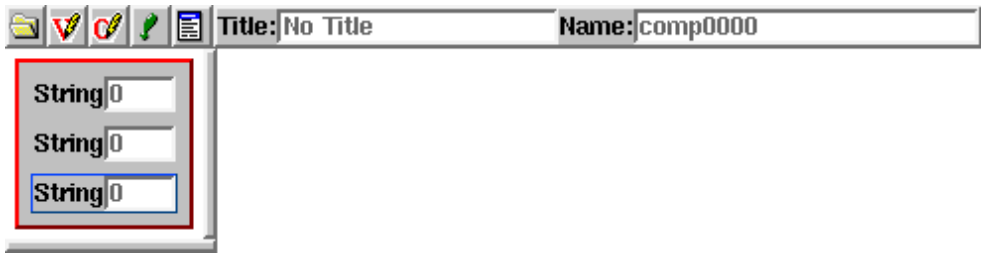
2.3. Step by Step Example

How do you edit such a user interface? Formating buttons and text fields is done by the system, therefore not the task of the programmer, who only has to fix the logical arrangement.

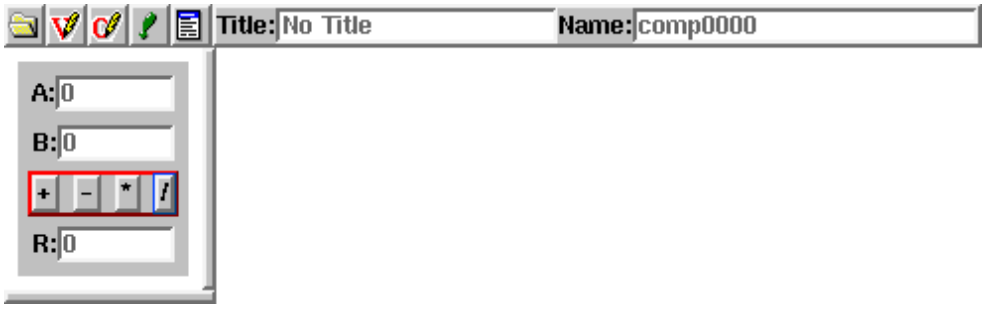
The project therefore is hierarchically arranged. The topmost hierarchy are the dialog windows. These windows understand two additional methods, `open` and `modal-open` which allows to create both non-modal and modal dialogs. The user then creates a framework of horizontal and vertical boxes inside the dialog. These boxes are filled with contents and glues then.

A examples will show how to use Theseus. The first creates a small calculator operating on integers. Create a "New Dialog" with the "Edit" menu. This is the dialog bar at the start of the project:

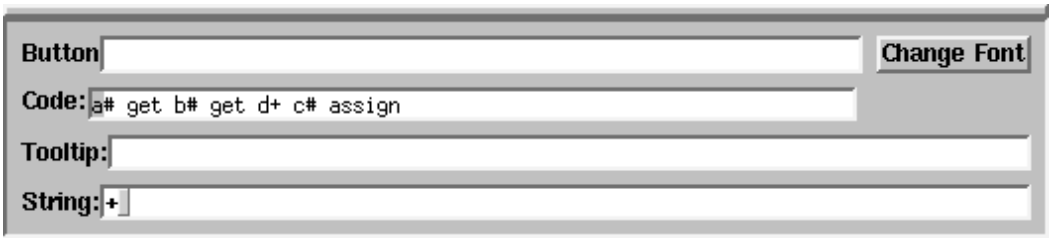
First put the three `infonumberfields` inside the starting box, they will be put under each other.



Beneath the two input fields the operation buttons should be arranged one aside each other. A horizontal box (`hbox`) does the job, with four buttons in it. First click to  , and then to `hbox`. Then set again to  . Click into the newly created box. Click off the vskip button: that makes it nicer. Hit four times on "Button". Now these objects need a useful text. Therefore you left-click each object (in edit mode), and type in the text. Hit Esc to clear the text field first.



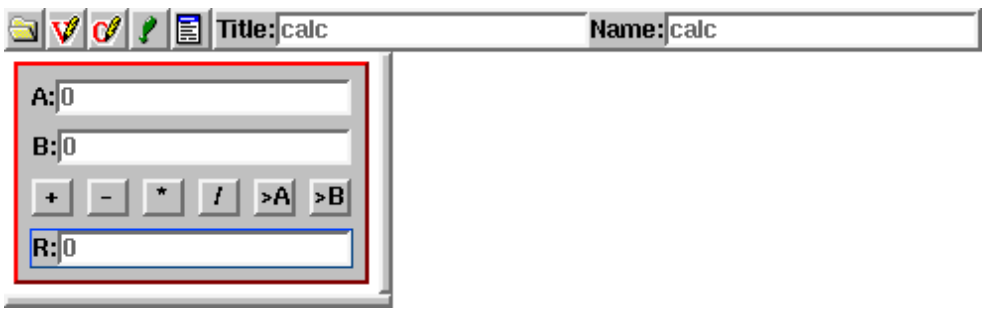
To reference the input field, each one must have an internal name. Choose edit mode, right-click to the fields and enter the name (`a#`, `b#`, and `c#`). Now you can insert code, i.e. for the operation `+`. Corresponding to the example below, the other code is inserted, too. Middle-click on the operation buttons and enter the code



The code looks as follows, for +, −, ∗, and /:

```
a# get b# get d+ c# assign
a# get b# get d- c# assign
a# get b# get d* c# assign
a# get b# get drop ud/mod c# assign drop
```

But stop! Maybe it's useful to take the result and copy it to one of the input buttons for reuse. Thus two additional buttons are required, and to make it nice, all buttons should have the same size (with "tabbing" box style). The window must have a title, and a name, too.



The added code is for >A and >B

```
c# get a# assign
c# get b# assign
```

Now you can try the result by pressing the  icon. Theseus generates the code and starts a new invocation of bigFORTH which compiles it and starts the application.

3. Widget Classes

The primary classes of MINOΣ are gadgets (widgets and displays), actors, and resources (xresource and font).

Resources just provide system-specific data like window handles, display pointers, font informations, etc., they are critical for system-specific parts of display implementations.

Widgets are the central objects represented on screen. Widgets draw through displays, such as windows, viewports, etc. Since many displays can be embedded into a bigger window, they also have all widget functionality. Widgets are composed in boxes (horizontal and vertical), and automatically layouted. Boxes certainly are widgets themselves, and some of the more complicated widgets such as sliders or textboxes are derived from boxes to layout the inner parts easily.

Actors are bound to associated actions when operating a widget (i. e. clicking on it, pointing on it, or entering text). Actors provide the way to script actions.

3.1. Actors

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):**

3.2. 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

This is one of the most important widget type.

3.3. Boxes

3.4. Displays

Resources

Fonts

4. Implementation Details

Programming something like MINOΣ is quite complex. The underlying graphic or GUI libraries are typically badly factored, and also are quite incompatible to each other. Fortunately, they provide a basic set of functions that's somewhat similar, and can be used as foundation for a portable GUI library.

4.1. How to program X with 50 functions

Round Trip Delay

X is a client-server protocol. You might find it strange that the server is the machine you are sitting in front of, while the client can be anywhere on the net — but that's due to the fact that X provides display service. And the display is what you are sitting before.

There are some difficulties to overcome with this approach, one is the time stamp X delivers for actions like mouse-clicks. That time stamp is computed from the clock of the server (your display). To detect double-clicks, the client has to measure the distance between events. As long as the events are coming along smoothly, no problem. But what if no further event is coming, like the user isn't really clicking, but holds the mouse button down? MINOΣ returns that event after a timeout.

Now, the timeout is computed using the local clock of the client (the remote machine where MINOΣ runs). The problem is that these clocks are not always synchronized. Yes, there's the network time protocol, ntp, but not all sysops are aware of that. We have to roll it on our own, and we are limited to the X protocol. Furthermore, ntp won't tell us the round trip delay, which we have to take into account, too (it can take seconds for a packet to come to us).

Now, our time precision requirements aren't as high as ntp, so it's not that difficult. There's a way to force X to generate a timestamp, that's with XChangeProperty. The window that has its property changed responds with an event, and that's what we use as time base (difference between our local computed time and the time stamp we get back).

To make sure we aren't doing anything wrong, we first save the attributes of the window we are going to misuse as target, and set the event mask so that we really get this event. Then we generate a few events (four by default), and record the minimum time. Finally, we restore the windows's properties, so that we can continue unaffected. That's repeated every minute to follow clock skews. Since we calculate the round trip delay here, but have to take only a single trip into account, we get a savety margin for connections farther away.

Event Handlers

MINOΣ handles events in the Displays class. All events are handled by the same code, if there are different things to do for the same event in different window classes, that's implemented by using polymorphic methods of the Displays class.

X fortunately has only a limited number of events, so we can use a simple table to put our handlers in.

Error Handling

One of the most annoying properties of Xlib for interactive applications is the way it handles errors. Xlib provides a default error handler that prints an error message and terminates the application. That's not really what we want. Therefore, the default error handler has to be replaced. All we can do is to save the error informations, and return to Xlib as if nothing happend, since Xlib won't be happy if we just use the exception handling, and don't let it clean up it's own mess.

The result is that errors now become quite silent. The errors are reported in the outer message loop, and quite a number of errors could have happend before that point is reached again. You can mark suspicious places with `x~~`, which shows the last error.

Text Selection

X uses two ways to exchange texts between applications. One uses XStoreBytes/XFetchBytes to put and fetch a single string from a global clipboard (that's the easy one), the other uses events, and is more complicated. Instead of storing information on a clipboard, you just tell X which window is the owner of the selection (XSetSelectionOwner). If you want to fetch the clipboard context, you first ask for the selection owner, then create

an XInternAtom to be passed around as token, and now ask the selection owner with XConvertSelection to send you the data.

The selection owner then receives a SelectionRequest message, which includes the atom passed around. The owner now binds the text to that atom — with XChangePoperty. Now you have to send back a SelectionNotify event to the requester. The requester receives that message, and now can use XGetWindowProperty to get the selection text.

Now the fun is that you don't know which selection protocol your partner is using. Most programs use the complicated way, but some old ones use the simple XStore-Bytes/XFetchBytes way. They aren't really compatible, but you can guess if nobody is an owner, you should look at the simple clipboard. And you can place your text with XStoreBytes on the server, even if nobody will be asking for it. The counterside is that this generates unnecessary traffic (the user might just cancel that selection and not use it for any data exchange).

12 Support Classes

TINOS 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 and 3D-Turtle-Graphics

On the last German Forth-Tagung, I presented direct OpenGL library bindings in Forth. OpenGL is a very powerful 3D graphics library. However, OpenGL is quite low-level, and provides "only" coordinate transformation and drawing of strips, thus strings of triangles or quads. Furthermore, OpenGL needs normal vectors and texture coordinates that could be computed automatically.

My intention was therefore to capsulate OpenGL in an easier to use library, a sort of 3D turtle graphics. Around new year 1999, there was a discussion in comp.lang.forth about such a 3D turtle graphics. Dave Taliaferro introduced a 3D turtle graphics written in pForth. Marcel Hendrix soon afterwards implemented something comparable in iForth.

Both turtles can move through space, and leave a trail composed of OpenGL objects, e. g. cylinders or spheres. You can't compose more complex bodies.

The system introduced here starts with the turtle principle, but it allows to describe bodies. Since it doesn't base on composition of fixed parts, a real sceleton animation is possible, something that even Hollywood tools can only accomplish with a lot of effort. Not accidentally the evening-filling films have insects, thus exosceletons, as actors. Animations with entrosceletons typically restrict to short sequences. 3000 points (the dragon) aren't easy to enter by hand.

1.1. The Principle

Ordinary 2D turtle graphics can walk forward and backward, and turn right or left. Thereby it leaves trails, thus lines. The principle can be extented to areas by filling the turtle drawn polygons.

In space, the turtle is in its right element (under water). Instead of crawling around clumsy, it can swim up and down and roll around its axis. You just must think about how its "trail" should look like, and how to get from strips and polygons to real bodies.

Instead of dropping pre-factured objects, this 3D turtle graphics allows to describe slices through the body. These slice planes then are connected to form a body. E. g. to create a cylinder, you connect two circles together. Circles are approximated by polygons.

The simple 3D turtle graphics doesn't provide a 2D turtle graphics for these slices (which might be somewhat intuitive), but different coordinate systems like cylinder coordinates. You could use the plain 3D turtle, as well, to draw outlines. The origin is defined by the turtle, the orientation of the coordinate system is where the turtle looks at.

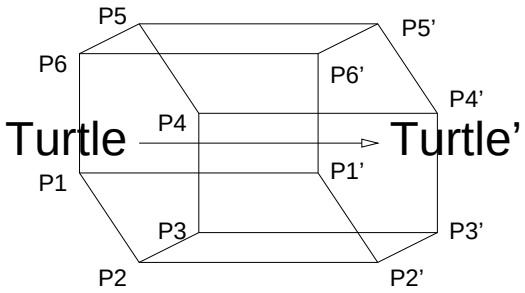


Figure 12.1: 3D-Turtle-Prinzip

1.2. A Simple Example

As simple example I'll choose a tree. A tree is composed of a trunk, and branches, which we'll approximate by hexagon-based cylinders. As leaf, I'll use a simple sphere approximation.

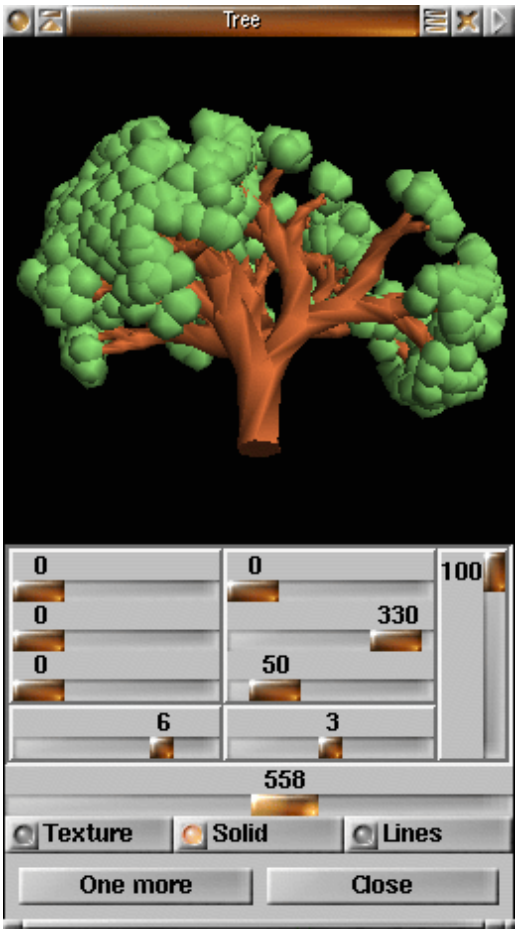


Figure 12.2: Tree

Our tree has a few parameters: the branch depth and the number of branches. The tree shown above also has a likelyhood with which the branches fall, we won't implement that here.

Let's start with the root. First, we need a lower surface, a hexagon. We leave the turtle as is, and open a path with six points per round.

```
: tree ( m n -- )
  .brown .color  6 open-path
```

The hexagons have an angle of $\pi/3$ per step, we can memorize that one now. It defines the step width for the functions that don't take an angle as parameter.

```
  pi 3 fm/ set-dphi
```

Now we start with six points in the middle. We first add the six points (the path is empty at the beginning), and in the next round we set them again, to set the normal vectors correctly (all beginning is difficult—since the normal vectors relate to the previous round, there are none in the first round).

```
  6 0 DO  add  LOOP next-round
  6 0 DO  set  LOOP next-round
```

Around them in the next round we draw the triangles that form the bottom hexagon. The size of the triangles is computed using the branch depth and multiplied by 0.03. Since OpenGL itself uses floating point numbers, the turtle graphics also works with such numbers.

```
  6 0 DO dup !.03 fm* set-r LOOP next-round
```

Now I use a small trick to create sharp edges—the 3D turtle graphics computes normal vectors on a point as sum of the cross products of the vectors left/behind and right/forward. Another slice at the same position causes that only one direction is considered for the normal vectors.

```
6 0 DO  dup !.03 fm* set-r  LOOP
```

Now we can procede with the real recursive part, the branches:

```
branches ;
: branches ( m n -- )  recursive
```

To avoid a double recursion, I use a loop for the end recursion.

```
  BEGIN  dup  WHILE
```

Even here we must start with a new round. To avoid that the tree is flat in a single plane, we roll it every branch by 54 degrees.

```
        next-round  pi !.3 f* roll-left
```

Next, we go forward corresponding to the branch depth to draw a new ring.

```
        dup !.1 fm* forward
        6 0 DO dup !.03 fm* set-r LOOP
```

For the other branches we need a loop—except for the last branch, that is done by the end recursion.

```
        over 1 ?DO
```

Each branch rotates around the turtle's eye axis—`I'` here is the end of the loop. The word `>turtle` saves the current state of the turtle on a turtle stack. `turtle>` takes it back again. I use a local variable, since the turtle needs some return stack, and therefore `I` and `I'` aren't accessible. The FP stack can be used only for intermediate computations, since the C library expects an empty stack.

After rotating we must turn right (with an angle of 18 degree here), and then turn the turtle back—so that the points of each slice fit together. The changed eye direction of the turtle remains with this operation, only the alignment in space is reconstructed.

```
            2pi I I' fm*/ { f: di |
            >turtle
               di roll-left pi 5 fm/ right
               di roll-right
               2dup 1- zweige
            turtle> }
```

Just finish the loop

```
            LOOP
```

and turn right for the end recursion (this time we roll by 0 degrees).

```
        pi 5 fm/ right
  1- REPEAT
```

Finally, we close the path and draw a leaf.

```
  close-path leaf 2drop ;
```

The leaf itself is a simple approximation to a sphere:

```
: 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 ;
```

These aren't all the sources, we need some overhead to change the view to the tree. The whole sources are in the file `tree.str` (3D graphics) and `tree.m` (user interface).

1.3. A More Complex Example: The Dragon

Since the dragon is really complex, I describe only the most important points. In typical Forth tradition the dragon is briddled by the tail.

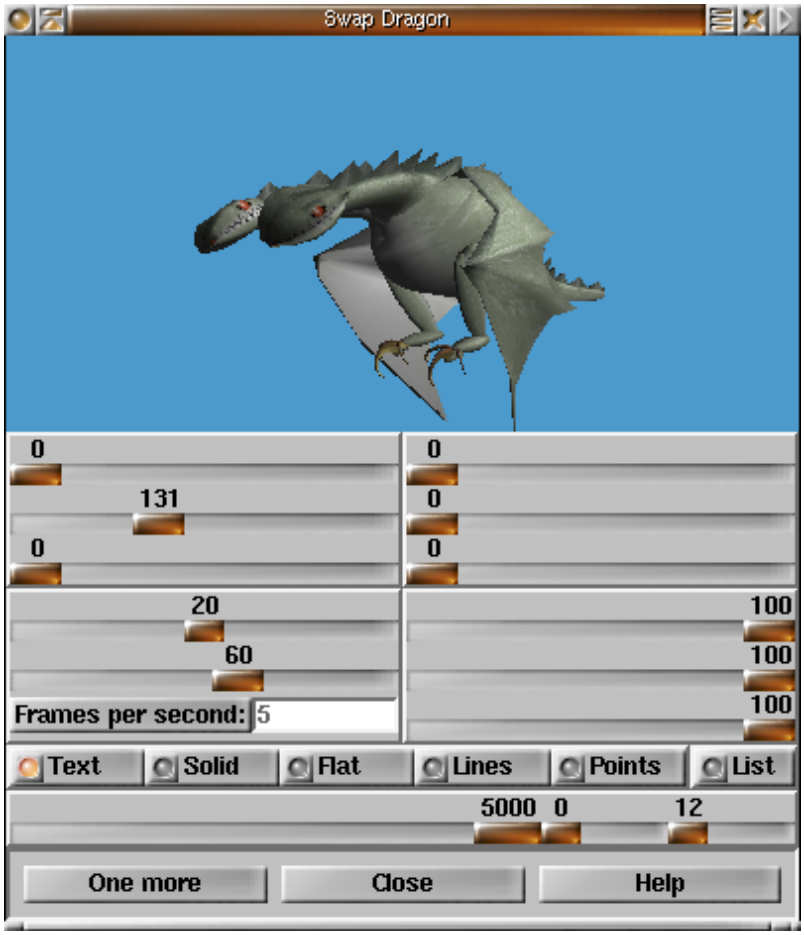


Figure 12.3: Swap Dragon

Tail

The dragon is composed of single segments, wich mainly are a circle with a point:

```
: 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! } ;
```

To wag the tail nicely, and to synchronize all the other movements, there's a timer that is turned into an angle $[0, 2\pi[$.

```
Variable tail-time
: time' ( -- 0..2pi )
  tail-time @ &24 &60 &30 * * um* drop
  0 d>f !$2'-8 pi f* f* ;
```

The real tail wagging now computes using segment number and time— the result is a translation left or right.

```
: tail-wag ( n -- f )
  >r pi r@ 1 + fm* !.2 f* time' f+
  fsin r> 2+ dup * 1+ fm/ !30 f* ;
```

The origin of the dragon is in the womb, not at the tail's point. The dragon however is drawn beginning with the tail's point—thus we first must compute a compensation, otherwise the tail wags with the dragon.[1]

```
: tail-compensate ( n -- f )  !0
  0 DO  I 2+ tail-wag f+ !1.1 f/  LOOP
  !1.1 !20 f** f* fnegate ;
```

The tail then is quite simple: first back to the tail's point, and set a point as initial polygon. Then, step by step forward, and draw a dragon segment. Each second segment has a point upwards, and scaling makes the tail thicker and thicker. The radius furthermore is enlarged, too. This scaling first has to be undertaken into the other direction. As texture mapping function, I use z, ϕ, thus the movement of the turtle as one texture coordinate and the angle against vertical for the other.

```
: 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 } ;
```

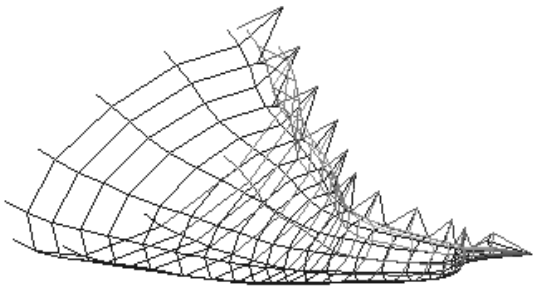


Figure 12.4: Tail

Body

The dragon's body is composed out of the same segments as the tail, but instead of growing further, the body must close again.

[1] Something like that sometimes happens in politics.

```
: 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- } ;
```

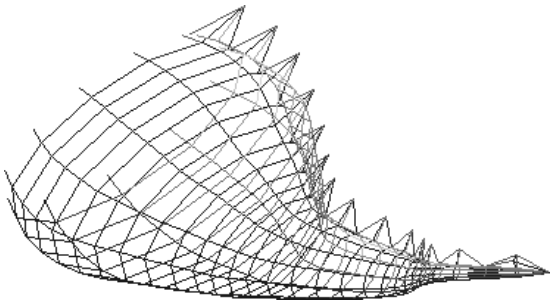


Figure 12.5: Body

Neck

The neck also consists of these segments, however, we have here two different growth functions, one for the shoulder (fast shrink), and one for the real neck (slow shrink). The shoulder turns left, the neck turns right again. Therefore the function `dragon-neck-part` is called two times.

```
: 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 } ;
```

Head

The head is composed using a rectangle with rounded edges and a slot for the teeth. This function isn't easy to generate, therefore I use an array for the coordinates, but just for the left half of the head; the right half is obtained by mirroring at the y axis. The

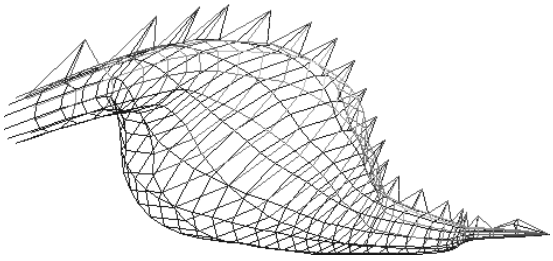


Figure 12.6: Neck

sizes of the slices is about the same as for the body. The head has a different texture, one with eyes, nose, and teeth.

```
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 ;
```

The second neck and head are drawn with corresponding negative angles. Like in the previous example, I save the state of the turtle to start from the same point again.

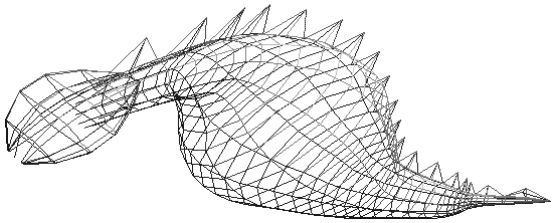


Figure 12.7: Head

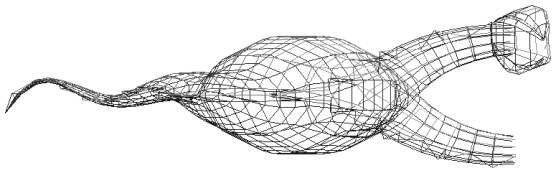


Figure 12.8: Second Neck

Wing

The wing has a simple, flat hexagon as slice. This hexagon provides the bending of the wing, and models the "fingers".

```
: 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 } ;
```

The folding function of the wing supplys a movement of arm/hand and finger dependent on time for a up/downward movement of the wing. f_2 is an additional term to the cosine, f_1 a multiplicative.

```
: wing-fold ( f1 f2 -- )
  time pi 5 fm/ f- fcos f+ f* down ;
```

The movements and the composing of the wing are complicated; therefore I don't explain all details. Here first I open a path, too. Then, step by step, shoulder, arm, and finally the fingers are drawn.

```
: wing ( -- )
  8 open-path !.9 scale
  6 0 DO  add  LOOP
  !.02 !1.2 wing-step !.3 forward          \ Shoulder
  pi &10 fm/ down  pi &8 fm/ roll-left
  time' fsin !1.3 f* !.2 f+ right
  !.02 !1 wing-step                        \ Upper arm
  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                          \ Lower arm
  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                            \ Closing step
  close-path ;
```

The wing is the same left and right. The symmetry is created by mirroring on the Y axis. Here, I must say another word about OpenGL: only the front sides of triangles really are drawn. The backfaces are culled. Such a mirror operation turns all fronts into "backs", since the turn changes. So I must tell OpenGL, and that's what `flip-clock` does.

```
: 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 ;
```

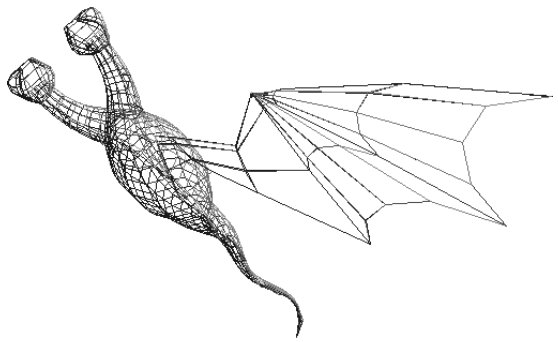


Figure 12.9: Wing

The Complete Dragon

I'll leave the legs out here, they aren't that interesting, since they consist of static parts (mostly long ellipsoids and bowed claws). Let's see the main program:

First of all, the dragon wags up and down with each wing fold. Then, for the dragon segments I must set the angle.

```
: 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
```

Then as said above, I draw the tail.

```
!.1 !.3 !.2 r@ dragon-tail
```

The return parameters of the tail are recycled in the body.

```
r> { f: ri f: r+ f: h n |
ri r+ h !.06 n dragon-wamp fdrop
```

I draw head and neck left and right starting at the same position, with negated angle parameters for the other side.

```
>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>
```

Then, the texture changes and I draw the two wings.

```
2dup !text
h !2 f* forward
>turtle  h right-wing   turtle>
>turtle  h left-wing   turtle>
```

I draw the legs with the same approach.

```
h !-6 f* forward
>turtle  right-leg   turtle>
>turtle  left-leg    turtle>
2drop 2drop } ;
```

1.4. Outlook

What can you do with that, and what's missing? A serious application is certainly the visualization of three dimensional data. Less "serious" applications would be computer games. They require collision detection, and quite likely a hierarchical model, to put spaces and moving/moveable objects in. Also different level of detail depending on the size of the object on the screen now must be programmed by hand. I'm not sure if there is another possibility for animated objects.

The usage of different textures is quite complex at the moment; they must be carried on the stack. Here, the 3D turtle object should provide better tools.

And again, Windows makes difficulties. Even though one can't say that the Mesa library under Linux is bug-free, it at least implements all features of OpenGL 1.2. The Windows 95 OpenGL library from SGI/Microsoft omits textures, and doesn't work very reliably. Since SGI opened up their GLX sources, the remaining Linux problems and the missing hardware support (only 3Dfx supported now) will be solved in the near future.

1.5. Instructions of the 3D Turtle Graphics

Class **3D–TURTLE (... –– ...)** ⟨***method***⟩**:** The 3D turtle.
Public:

Navigation

Early Method **LEFT (FS f ––):** turns the turtle's head left
Early Method **RIGHT (FS f ––):** turns the turtle's head right
Early Method **UP (FS f ––):** turns the turtle's head up
Early Method **DOWN (FS f ––):** turns the turtle's head down
Early Method **ROLL–LEFT (FS f ––):** rolls the turtle's head left
Early Method **ROLL–RIGHT (FS f ––):** rolls the turtle's head right
Early Method **X–LEFT (FS f ––):** rotate the turtle left around the x axis
Early Method **X–RIGHT (FS f ––):** rotate the turtle right around the x axis
Early Method **Y–LEFT (FS f ––):** rotate the turtle left around the y axis
Early Method **Y–RIGHT (FS f ––):** rotate the turtle right around the y axis
Early Method **Z–LEFT (FS f ––):** rotate the turtle left around the z axis
Early Method **Z–RIGHT (FS f ––):** rotate the turtle right around the z axis
Early Method **FORWARD (FS f ––):** move the turtle in z direction
Early Method **FORWARD–XYZ (FS fx fy fz ––):** move the turtle
Early Method **DEGREES (FS 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 (FS f ––):** scales the turtle's step width by the factor f
Early Method **SCALE–XYZ (FS 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 (MS –– m):** push turtle matrix on the matrix stack
Early Method **MATRIX> (MS m ––):** pop turtle matrix from the matrix stack
Early Method **MATRIX@ (MS m –– m):** copy turtle matrix from the stack
Early Method **1MATRIX (MS –– 1):** initialize turtle state with the identity matrix
Early Method **MATRIX∗ (MS m1 m2 –– m3):** 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 (FS x y z −−):** 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 (FS x y z −−):** 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 (FS r phi z −−):** set with cylinder coordinates

EARLY METHOD **SET–XY (FS x y −−):** set-xyz with $z = 0$

EARLY METHOD **SET–RP (FS r phi −−):** set with cylinder coordinates, $z = 0$

EARLY METHOD **SET–R (FS r −−):** 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 (FS r phi z −−):** add with cylinder coordinates

EARLY METHOD **ADD–XY (FS x y −−):** add-xyz with $z = 0$

EARLY METHOD **ADD–RP (FS r phi −−):** add with cylinder coordinates, $z = 0$

EARLY METHOD **ADD–R (FS r −−):** 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 (FS dphi −−):** 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

VARIABLE **SMOOTH (−− addr):** 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 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 MINOΣ 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

13 American National Standard FORTH

1. History

After the Forth–83–Standard has been approved in 1983, several members of the standard team started an intiative for an official standard. Since august 1987 a team, the X3J14 technical comittee, worked on a standard for the American National Standard Institute, short ANSI. This Standard has been officially approved (on March 24, 1994).

bigFORTH confirms to this standard. All words defined in the standard are available in bigFORTH or can be loaded into bigFORTH. For compatibility to older versions of bigFORTH, some words don't comply to the standard's behavior by default. By loading the file `ans.fs`, the behavior of these words is changed towards ANS Forth.

2. Wortsets

The ANSI standard groups words into socalled "wordsets". These sets allows developers to adjust system size to demand. Each wordset consists of two parts, whereas the one with "–EXT" contains extensions, which can be added individually, and thus haven't to be all defined.

bigFORTH however implements all wordsets of the standard, and all the extensions, because only a completely supported standard makes sure that all standard-conforming programs will run. In the following sections, all wordsets of the standard are listed, as well as all words that differ from bigFORTH 1.10.

2.1. The CORE Wordset

Only one wordset is mandatory: the CORE wordset. All other wordsets are optional, a missing implementation doesn't make a Forth non-conforming to the standard. Even though, bigFORTH implements all wordsets, or allows to load them from source.

! # #> #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.FB 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.FS nicht geladen wurde, die vom Standard, wenn ANS.FS 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.FB ohne ANS.FS 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 233).

(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.

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`

14 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.

15 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 "or any later version"applies 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.

16 Glossary

1. Index

2. Bibliography

[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