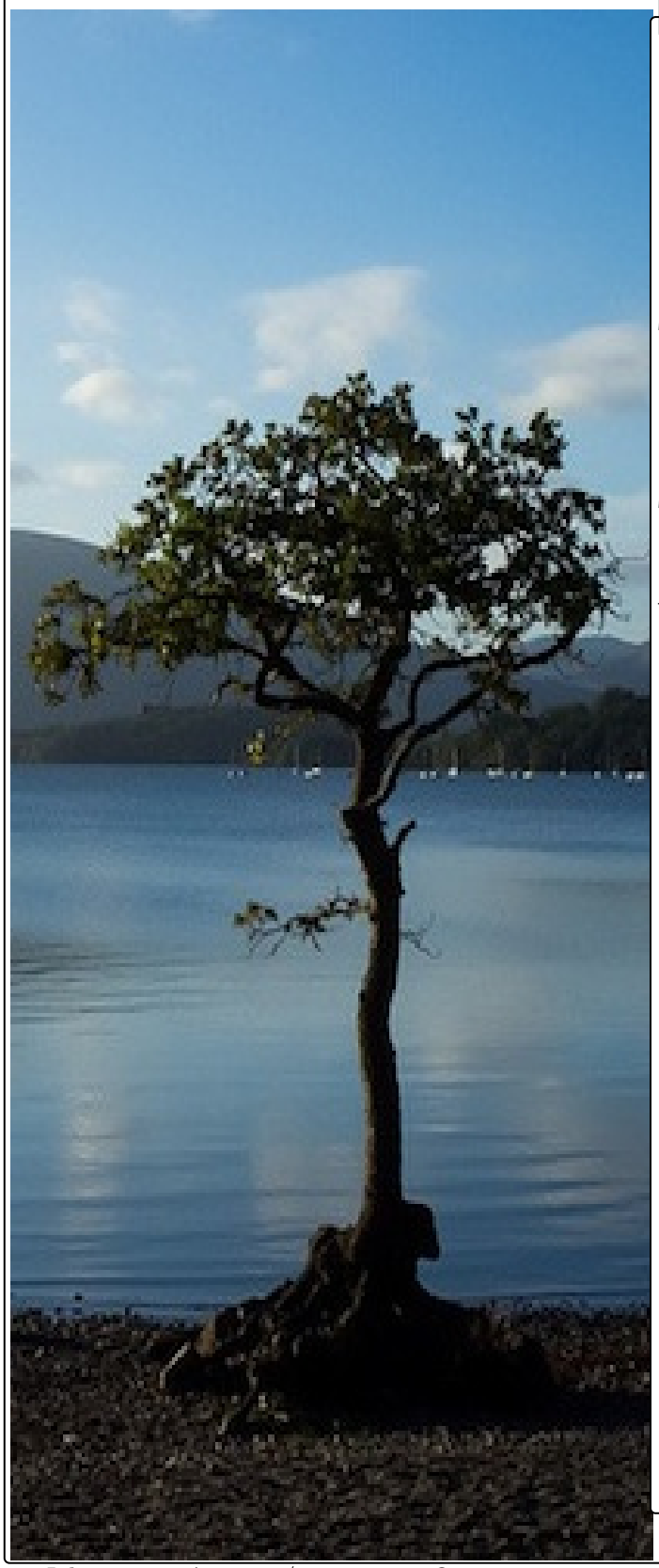
Vierte Dimension

4/2020

Das Forth–Magazin

für Wissenschaft und Technik, für kommerzielle EDV, für MSR–Technik, für den interessierten Hobbyisten



In dieser Ausgabe:

tematik GmbH
Technische
Informatik

Feldstraße 143
D–22880 Wedel
Fon 04103 – 808989 – 0
Fax 04103 – 808989 – 9
mail@tematik.de
`http://www.tematik.de`

Seit 2001 entwickeln und vertreiben wir unter dem Markennamen „Servonaut“ Baugruppen für den Funktionsmodellbau wie Fahrtregler, Lichtanlagen, Soundmodule und Funkmodule. Unsere Module werden vorwiegend in LKW–Modellen im Maßstab 1:14 bzw. 1:16 eingesetzt, aber auch in Baumaschinen wie Baggern, Radladern etc. Wir entwickeln mit eigenen Werkzeugen in Forth für die Freescale–Prozessoren 68HC08, S08, Coldfire sowie Atmel AVR.

Forth–Schulungen

Möchten Sie die Programmiersprache Forth erlernen oder sich in den neuen Forth–Entwicklungen weiterbilden? Haben Sie Produkte auf Basis von Forth und möchten Mitarbeiter in der Wartung und Weiterentwicklung dieser Produkte schulen?

Wir bieten Schulungen in Legacy–Forth–Systemen (FIG–Forth, Forth83), ANSI–Forth und nach den neusten Forth–200x–Standards. Unsere Trainer haben über 20 Jahre Erfahrung mit Forth–Programmierung auf Embedded–Systemen (ARM, MSP430, Atmel AVR, M68K, 6502, Z80 uvm.) und auf PC–Systemen (Linux, BSD, macOS und Windows).

Carsten Strotmann carsten@strotmann.de
`https://forth-schulung.de`

RetroForth
Linux · Windows · Native
Generic · L4Ka::Pistachio · Dex4u
Public Domain
`http://www.retroforth.org`
`http://retro.tunes.org`

Diese Anzeige wird gesponsort von:
EDV–Beratung Schmiedl, Am Bräuweiher 4,
93499 Zandt

Cornu GmbH
Ingenieurdienstleistungen
Elektrotechnik

Weitlstraße 140
80995 München
`sales@cornu.de`
`www.cornu.de`

Unser Themenschwerpunkt ist automotive SW unter AutoSAR. In Forth bieten wir u. a. Lösungen zur Verarbeitung großer Datenmengen, Modultests und modellgetriebene SW, z. B. auf Basis eCore/EMF.

KIMA Echtzeitsysteme GmbH

Güstener Straße 72 52428 Jülich
Tel.: 02463/9967–0 Fax: 02463/9967–99
`www.kimaE.de` info@kimaE.de

Automatisierungstechnik: Fortgeschrittene Steuerungen für die Verfahrenstechnik, Schaltanlagenbau, Projektierung, Sensorik, Maschinenüberwachungen. Echtzeitrechnersysteme: für Werkzeug– und Sondermaschinen, Fuzzy Logic.

FORTecH Software GmbH

Tannenweg 22 m D–18059 Rostock
`https://www.fortech.de/`

Wir entwickeln seit fast 20 Jahren kundenspezifische Software für industrielle Anwendungen. In dieser Zeit entstanden in Zusammenarbeit mit Kunden und Partnern Lösungen für verschiedenste Branchen, vor allem für die chemische Industrie, die Automobilindustrie und die Medizintechnik.

Ingenieurbüro
Klaus Kohl–Schöpe

Tel.: **(0 82 66)–36 09 862**
Prof.–Hamp–Str. 5
D–87745 Eppishausen

FORTH–Software (volksFORTH, KKFORTH und viele PD–Versionen). FORTH–Hardware (z. B. Super8) und Literaturservice. Professionelle Entwicklung für Steuerungs– und Messtechnik.

Mikrocontroller–Verleih
Forth–Gesellschaft e. V.

Wir stellen hochwertige Evaluation–Boards, auch FPGA, samt Forth–Systemen zur Verfügung: Cypress, RISC–V, TI, MicroCore, GA144, SeaForth, MiniMuck, Zilog, 68HC11, ATMEL, Motorola, Hitachi, Renesas, Lego ...
`https://wiki.forth-ev.de/doku.php/mcv:mcv2`

Titelbild: Baum am See Autor: M.Kalus (Bildbearbeitung mit GIMP)
Quelle: Bildausschnitt eines aus dem Netz gefischten freien Fotos.

Impressum

Name der Zeitschrift

Vierte Dimension

Herausgeberin

Forth–Gesellschaft e. V.
Postfach 32 01 24
68273 Mannheim
Tel: +49(0)6239 9201-85, Fax: -86
E–Mail: `Secretary@forth-ev.de`
`Direktorium@forth-ev.de`
Bankverbindung: Postbank Hamburg
BLZ 200 100 20
Kto 563 211 208
IBAN: DE60 2001 0020 0563 2112 08
BIC: PBNKDEFF

Redaktion & Layout

Bernd Paysan, Ulrich Hoffmann
E–Mail: `4d@forth-ev.de`

Anzeigenverwaltung

Büro der Herausgeberin

Redaktionsschluss

Januar, April, Juli, Oktober jeweils in der dritten Woche

Erscheinungsweise

1 Ausgabe / Quartal

Einzelpreis

4,00 € + Porto u. Verpackung

Manuskripte und Rechte

Berücksichtigt werden alle eingesandten Manuskripte. Leserbriefe können ohne Rücksprache wiedergegeben werden. Für die mit dem Namen des Verfassers gekennzeichneten Beiträge übernimmt die Redaktion lediglich die presserechtliche Verantwortung. Die in diesem Magazin veröffentlichten Beiträge sind urheberrechtlich geschützt. Übersetzung, Vervielfältigung, sowie Speicherung auf beliebigen Medien, ganz oder auszugsweise nur mit genauer Quellenangabe erlaubt. Die eingereichten Beiträge müssen frei von Ansprüchen Dritter sein. Veröffentlichte Programme gehen — soweit nichts anderes vermerkt ist — in die Public Domain über. Für Text, Schaltbilder oder Aufbauskizzen, die zum Nichtfunktionieren oder eventuellem Schadhaftwerden von Bauelementen führen, kann keine Haftung übernommen werden. Sämtliche Veröffentlichungen erfolgen ohne Berücksichtigung eines eventuellen Patentschutzes. Warennamen werden ohne Gewährleistung einer freien Verwendung benutzt.

Liebe Leser,

nun neigt sich das Jahr 2020 dem Ende zu, und ich kann sagen, das war mit das traurigste Jahr meines Lebens. Die Einschräkungen, die wir erleben im Lockdown, machen mir schwer zu schaffen. Ich habe Mühe, nicht selber depressiv zu werden, nicht zu resignieren. Meetings per Videokonferenz–Schaltung sind mir kein Ersatz. Ich kann das nicht. Ich meine, technisch ginge das, hab es ja durchaus probiert, anfangs. Aber es schmerzt mich sehr, die Beteiligten nicht persönlich anzutreffen. Wer das kann, soll es also machen. Immerhin sind da ja auch Menschen online zusammengekommen, die wegen der Entferungen voneinander zu einer Versammlung nicht hätten kommen können.

So hatte die Euroforth–2020 „Rome“ viele interessierte Teilnehmer. Was es da so alles gab an Beiträgen, stellt Gerald Wodni nun zusammen und läd es nach und nach hoch auf YouTube in die *4ther–Gruppe*. Vermutlich werden auch noch andere Beiträge dort landen — bin gespannt.

Neu in der Runde unserer Autoren sind Patrick Hedfeld und Wolfgang Strauss. Willkommen an Bord! Patrick erforschte das Leben im Spiel und Wolfgang das Forth im Propeller 2. „Neugierig sein“ ist ihr Lebensmotto, dem ich mich doch gerne anschließe, ist es doch existentiell für das menschliche Leben an sich.

Dann haben wir eine Kontroverse: Bernd Paysan verfasste eine Entgegnung auf Klaus Schleisiek aus *Heft 4d2020–03*. Beide sind Forth–Forscher von Gewicht. Man darf gespannt sein, wie es ausgeht.

In das Dunkel um das *grün–GRÜN–Problem* aus dem letzten Heft bringt Anton Ertl viel Licht — vielen Dank für diese tiefen Einblicke in die Materie — ganz besonders meinerseits. Und er weiht uns ein in das, was im Forth–Standardisierungs–Meeting in diesem Jahr alles geschafft worden ist. Auch Martin Bitter nimmt uns mit in Implementations–Tiefen: VOC — oder die Leichtigkeit der automatischen Produktion politischer Phrasen!

Ich selbst hab auch mal wieder Zeit gefunden, etwas zu erforschen, mit und im Forth, das „Conditional Compiling“, angeregt durch eine Mitteilung von Matthias Koch in seinem *Mecrisp*.

Das Titelbild „Baum am See“ bedeutete mir sowas wie Game of Live in „echt“ — ist es nicht erstaunlich, was, wann, wo und wie so alles keimt und wächst und wird?

Euer Michael

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
`http://fossil.forth-ev.de/vd-2020-04`

Die Forth–Gesellschaft e. V. wird durch ihr Direktorium vertreten:

Ulrich Hoffmann
Bernd Paysan
Carsten Strotmann

Kontakt: `Direktorium@Forth-ev.de`

Inspiration from a 1996 comp.lang.forth post of yours

Kürzlich erhielt Bernd Paysan folgende E–Mail:

> „Hi there!
>
> Way back in 1996, you wrote a post with the *minimum set of words* defined in assembly that is needed for a Forth, in this Usenet thread:
>
> `https://groups.google.com/g/comp.lang.forth/c/NS2icrCj1jQ`
>
> I wrote a very small x86 Forth that fits in a boot sector based on that: `https://github.com/cesarblum/sectorforth`
>
> Just wanted to shoot you a message to let you know about that :-) Writing sectorforth took me a month of evenings and weekends, and has been a super fun project.
>
> Cheers! Cesar Blum“

Cool! Wohl das kleinste Forth–System überhaupt! Und dort auf GitHub war zu lesen:

> „sectorforth is a 16bit x86 Forth that fits in a 512–byte boot sector . . .
>
> sectorforth contains only the eight primitives outlined in the Usenet post above, five variables for manipulating internal state, and two I/O primitives.
>
> With that minimal set of building blocks, words for branching, compiling, manipulating the return stack, etc. can all be written in Forth itself . . .“

Das Minimal–Wordset ist in unserem Forth–Wiki.

`https://wiki.forth-ev.de/doku.php/words:kernel_embedded:minimum_word_set`

FORTH, Inc.™

Archiv des Journal of Forth Application and Research (JFAR)

FORTH, Inc. hat neulich sein legendäres Journal allgemein zugänglich gemacht.

> „JFAR was a refereed journal and long served an important role in the early years of Forth programming. Its diverse content is a resource we help to preserve by making it available here, without charge. All code examples may not run unmodified on modern, standards–based Forth systems but the knowledge it nevertheless conveys may be invaluable.“

`https://www.forth.com/forth-books/jfar-archives/`

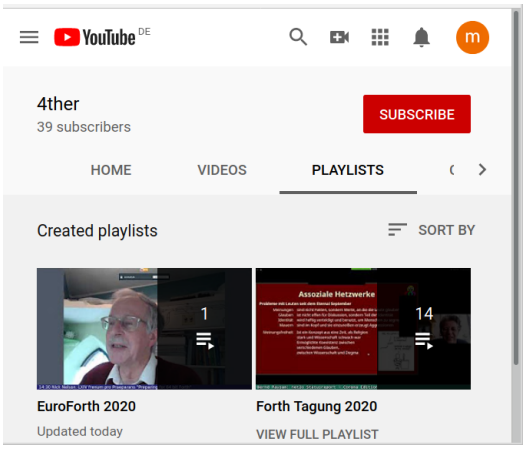
Conway

John Horton Conway war ein britischer Mathematiker. Er wurde am 26. Dezember 1937 in Liverpool geboren. Und ist in diesem Jahr am 11. April gestorben. Er lebte zuletzt in New Brunswick, New Jersey, USA. Er war dreimal verheiratet und hatte sieben Kinder — Game of Live, in echt.

`https://de.wikipedia.org/wiki/John_Horton_Conway`

Schaut euch mal die wunderbare Sammlung der Welten des *Game of Live* von Oingo Boingo an. Sein Video „epic conway's game of life“ auf YouTube hat mich sehr beeindruckt. mk

4ther



In der YouTube–Gruppe *4ther* werden seit diesem Jahr die Videos der Forth–Online–Konferenzen hinterlegt. Nach dem Stand von heute sind dort schon zwei Playlists vorhanden, die der Forth–Tagung 2020 mit 14 Beiträgen, und ein erster Beitrag von der EuroForth — Nick Nelson, „LXIV frenum pro Praeparans“. Willst du wissen, was das heißt? Na, dann schau da mal rein . . .

So, wie es die Zeit erlaubt, werden nun nach und nach die anderen Beiträge der EuroForth folgen. Ihr könnt die Gruppe abonnieren, um auf dem Laufenden zu bleiben. 39 Subscriber sind es schon. mk

Adventskalender

Die Qemu–Leute haben wieder einen Adventskalender. Und weil ich den lustig finde, hab ich mal den Kalender von 2018 aufgeklickt: `https://www.qemu-advent-calendar.org/2018/` Da — Day 4 — Fun with Forth! Ein Snake–Spiel, geschrieben in OpenFirmware für ppc64 :-)

Erich Wälde

Und noch mehr für euch zu Weihnachten

Spiel 2048

Dave Buckling hat das *Spiel 2048* in GNU/Forth übertragen. Wikipedia:

> „2048 ist ein freies Computerspiel, das von einem einzelnen Spieler im Webbrowser und auf Mobilgeräten gespielt werden kann. Es wurde im März 2014 von dem 19–jährigen italienischen Web-Entwickler Gabriele Cirulli erstellt. Ziel des Spiels ist das Erstellen einer Kachel mit der Zahl 2048 durch das Verschieben und Kombinieren anderer Kacheln.“[1]

`https://gitlab.com/davebucklin/2048/-/blob/master/2048.fs`

SixtyForth

SixtyForth ist ein in Assembler geschriebenes Forth–System für 64–Bit–Intel/AMD–Prozessoren. SixtyForth implementiert den Core–Wortschatz von ANSI–Forth.

`https://gitlab.com/drj11/sixtyforth`

m64th

m64th ist ein weiteres Forth für x86_64–Linux–Systeme von Massimiliano Ghilardi. Neben x86_64 ist m64th auch für AARCH64 (ARM) unter Linux und Android verfügbar. m64th ist unter aktiver Entwicklung und schon recht weit fortgeschritten.

`https://github.com/cosmos72/m64th`

nopforth

nopforth ist ein „nicht–standardisiertes“ Forth für Linux, NetBSD und Windows–Systeme mit Cygwin. Es ist unter aktueller Entwicklung.

`https://github.com/iru-/nopforth`

9P

9P ist ein sehr simples Protokoll für den Dateiaustausch (Remote–Filesystem) aus dem Unix–Nachfolger–Betriebssystem Plan 9.[2] Dieses Protokoll ist auf viele Betriebssysteme portiert worden (inkl. Linux) und wird auch in der Virtualisierungs–Lösung *QEMU* für den Dateiaustausch zwischen Host– und Gast–Systemen verwendet. Der Autor von nopforth hat das 9P–Protokoll unter GNU/Forth implementiert.

BoardForth

Boardforth ist das Projekt von Nick Pasucci aus Schottland. Ziel dieses ungewöhnlichen Forth–Systems ist es, *Gerber–Dateien* für die Erstellung von elektronischen Platinen direkt in Forth zu machen. Das Projekt ist noch am Anfang und noch nicht benutzbar. Eine fertige Platinen–Definition in Boardforth könnte nach der Idee des Autors aussehen wie im folgenden Listing gezeigt.

`https://github.com/nickpascucci/boardforth`

Listing:

```
1  include boardforth.fth
2  board MY_BOARD
3  my_board current_board !
4  \ Sets the number of layers on the board.
5  2 layers
6  \ Origin is the bottom left corner of whatever
7   \ shape the user specifies here.
8  \ mm: ( n -- n ) Converts millimeter coordinate
9   \ to internal representation
10 30 mm 30 mm rectangular
11 \ Set the draw location to (5mm, 5mm)
12  \ go: ( x y -- ) moves edit location
13 5 mm 5 mm go
14 \ Add a custom drawn object, in this case
15  \ a 3mm fiducial.
16 \ mark.fiducial: ( d -- )
17  \ creates mark and adds to board
18 3 mm mark.fiducial
19 8 mm 0 mm go
20 \ part.uc.2X3_HEADER
21 \ ( -- ) creates header, adds to board,
22  \ defs variable)
23 part.conn.2X3_HEADER H1
24 10 mm 10 mm go
25 \ Add an ATMEGA128 to the board named "IC1"
26  \ part.uc.ATMEGA128-16AU
27  \ ( -- ) creates IC component, adds to board,
28  \ defs variable)
29 part.uc.ATMEGA128-16AU IC1
30 \ Add a connection from IC1 pin 22 to H1 pin 1.
31  \ connect: ( addr n addr n -- addr )
32  \ leaves address of trace on stack)
33 IC1 22 pin H1 1 pin connect
34
35 \ Route the connection along the board
```

Carsten Strotmann

Fortsetzung auf Seite 23

[1] `https://de.wikipedia.org/wiki/2048` (Computerspiel)

[2] `https://de.wikipedia.org/wiki/Plan_9` (Betriebssystem).

Kopierwelt in Conways „Game of Life“

Patrick Hedfeld

Conways „Spiel des Lebens“ oder „Game of Life“ ist ein vom Mathematiker John Conway *erdachtes Spiel, welches auf einem sogenannten zellulären Automaten basiert.*[1] *Ein zellulärer Automat ist ein räumlich diskretes dynamisches System, dessen Entwicklung von der Nachbarschaft des vorherigen Schrittes abhängt.*

Einfacher ausgedrückt: Stellen wir uns ein riesengroßes Spielfeld vor, welches mit einzelnen Spielsteinen besetzt ist. Schwarze Steine bedeuten, dass das Feld „besetzt“ ist oder „lebt“ und weiße Steine wiederum drücken aus, dass das Feld „leer“ oder „unbelebt“ bzw. „tot“ ist.[2]

Ein Spielfeld könnte dabei so aussehen wie es Abb. 1 zeigt.

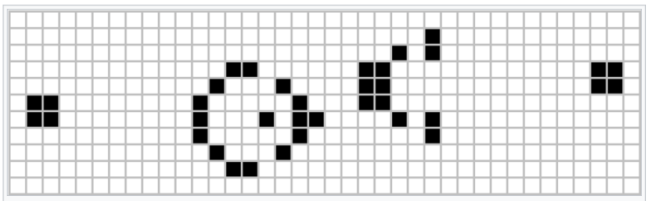


Abbildung 1: Spielfeld–Muster

Warum heißt das Spiel nun „Game of Life“ und was lebt denn hier? Das Spielfeld ist der „Input“ für die nächste Generation bzw. den nächsten Schritt, der ebenfalls ein Spielfeld darstellt. Man kann sich die Nachbarn eines jeden Feldes anschauen. Jeder Punkt im Spielfeld hat genau acht mögliche Nachbarn. Der Zustand der Zelle im nächsten Schritt hängt nun von ihrem eigenen Zustand und dem Zustand dieser acht Nachbarn ab. Man unterscheidet hier verschiedene „Welten“ basierend auf unterschiedlichen Regelsystemen.

Die klassische Welt

Ein paar Regeln:

- Eine „weiße“ oder „tote“ Zelle mit genau drei lebenden Nachbarn wird in der Folgegeneration neu „geboren“.
- Lebende Zellen mit weniger als zwei „lebenden“ Nachbarn sterben in der Folgegeneration an „Vereinsamung“ bzw. werden weiß.
- Eine lebende Zelle mit zwei oder drei lebenden Nachbarn bleibt in der Folgegeneration am Leben bzw. schwarz.
- Lebende Zellen mit mehr als drei lebenden Nachbarn „sterben“ in der Folgegeneration an Überbevölkerung und werden wieder „weiß“.

So eine Welt nennt man eine *23/3–Welt*, da die Zellen bei zwei oder drei Nachbarn „am Leben bleiben“ und bei drei Zellen „neu geboren“ werden. Somit beschreibt die Zahl vor dem Strich das „am Leben bleiben“ und die Zahl nach dem Strich die „Neugeburt“.

Jetzt gibt es verschiedene Objekte in einer 23/3 Welt, die von besonderem Interesse sind. Es gibt sogar ganze Kategorien aus Objekten, die man untersuchen kann.

Einerseits die *statischen Objekte*, also solche, die sich im Spiel selbst *nicht bewegen* oder über den Bildschirm wandern (Abb. 2).

Abbildung 2: Statische Objekte

Andererseits *oszillierende Objekte* wie den „Blinker“ (Abb. 3), oder sogar „Gleiter“, also Objekte, die über den Bildschirm „wandern“ (Abb. 4).

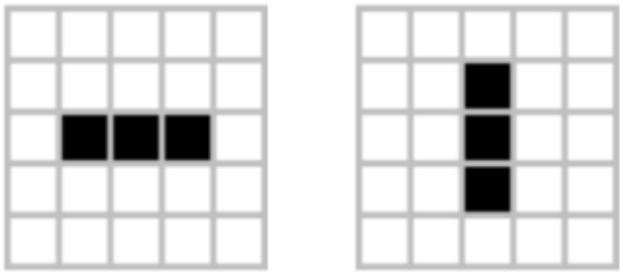


Abbildung 3: Blinker

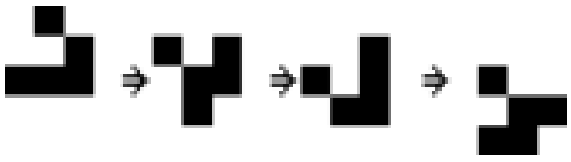


Abbildung 4: Gleiter

Für spannende Selbstversuche empfiehlt sich ein Simulator[3].

[1] Vergleiche zu dem Thema: Adamatzky, Andrew; „Game of Life Cellular Automata“, Springer Berlin 2010.

[2] Vergleiche dazu: Ilachinski, Andrew; „Cellular Automata: A Discrete Universe“, World Scientific, Singapur 2001

[3] `https://bitstorm.org/gameoflife/` (Stand 08.09.2020)

Jetzt existieren neben dieser *klassischen Welt* auch andere Welten, viele sind nicht weiter von Bedeutung, aber einige können eine erstaunliche Entwicklung entfalten. Beispielsweise die 236/3–Welt zeigt eine große Menge an „Explosionen“. Die 12345/3 erzeugt „Labyrinthmuster“. Für diesen Artikel interessant ist die 1357/1357–Welt, eine sogenannte „Kopierwelt“.

Basierend auf dem öffentlich zugänglichen Code in Forth für eine „klassische Conway–Welt“[4] wird im angefügten Listing der Code für die Kopierwelt wiedergegeben und im Folgenden erklärt.

Code für eine „Kopierwelt“

Der Code beginnt mit einem sogenannten „Wrappen“ (bis Zeile 6), welchem hier in der inhaltlichen Erklärung keine weitere Beachtung geschenkt werden soll. Im Folgenden wird die „Größe“ der Welt festgehalten und aufgebaut (Zeilen 8 bis 20). Gefolgt von ein paar wichtigen Definitionen, die den späteren Code leichter lesbar gestalten (Zeilen 31 bis 36). Die Abarbeitung einzelner Zeilen erfolgt über `foreachrow()` (Zeile 38, 39), die Anzeige über `showrow()` (Zeilen 46 bis 48). Zusammengefasst zum Befehl `show()` in Zeile 46 bis 48.

Die eigentliche Welt bzw. dessen Beschreibung erfolgen von Zeile 50 bis 58. Durch die vorher gemachten Definitionen passt das in nur acht Zeilen. Je eine Zeile, um die jeweiligen Nachbarn zu betrachten.

Die „Regeln“ für die entsprechende Welt entstehen dann im Abschnitt ab `gencell()` (Zeile 61 bis 72).

Die „klassische Welt“ (die 23/3–Welt) wird wie folgt abgebildet:

```
: gencell ( i -- )
  sum-neighbors over old + c@
  or 3 = 1 and swap new + c! ;
```

Hierbei gilt es zu beachten, dass es sich bei der Aufschlüsselung um einen zusätzlichen Trick in Forth handelt, der gleichermaßen mit den Zahlen 2 und 3 als „lebenserhaltende“ Parameter umgeht.[5] Die Information über die Nachbarn geht ein durch den Befehl `sum-neighbors`. Der Parameter `c` entspricht dabei der Information der jeweiligen Zelle. `old` beinhaltet die Informationen der „Altwelt“ und `new` die der neuen Welt nach dem Abgleich mit den jeweiligen Nachbarn.

Dieses klassische Regelwerk wird nun für die 1357/1357–Welt in `gencell()` umgewandelt zu:

```
\ Regeln der Kopierwelt
: gencell ( i -- )
  sum-neighbors
  dup 1 =
  swap dup 3 =
  swap dup 5 =
  swap 7 = or or or
  1 and swap new + c! ;
```

Man beachte den Abgleich gegen die Zahlen 1, 3, 5 und 7 sowie das Verschwinden der Information `old`. Denn in diesem Fall einer Kopierwelt ist es nicht weiter wichtig, wie der Zustand der Zelle vorher war — „tot“ oder „lebendig“ — da die gleichen Varianten vor und nach dem Schrägstrich (1357/1357) stehen und so im Regelwerk sind.

Die Definitionen `gen()` und `genrow()` (Zeilen 69 bis 72) dienen zur Erzeugung der einzelnen Zeilen im Programm. Die Definitionen `geniter()` (Zeile 74, 75) wurde noch zum Originalcode hinzugefügt, um die spätere Auswertung und den Start des Programmes zu erweitern. Die Definition für `life()` (Zeile 77) dient ebenfalls der Auswertung des Programmes.

Es folgt darüber hinaus noch eine Sammlung an Figuren, die in der entsprechenden Welt ausprobiert werden können. Die Zeilen 89 bis 99 enthalten unter anderem Blinker, Gleiter, Pulsare und vieles mehr.

In den Zeilen 106 und 109 erfolgt nun der eigentliche Start des Programmes, der sich wie folgt lesen lässt:

Erschaffe die Figur `blinker` an der Position `1000` der quadratischen 16 x 64–Welt und zeige diesen an — Befehl `show()` .

Laufe dann 16 Generationen[6] lang und zeige den jeweiligen Output am Bildschirm an.

Ergebnis der Kopierwelt

Die Ausgabe für diese Einstellung ist in den *Screenshots* gezeigt — 16 Generationen.[7]

Man beachte den „Blinker“ in der 0. Generation (Startgeneration). Er wird durch die Regel der 1357/1357–Welt (Kopierwelt) verteilt. In jeweiligen Schritten von einem Vielfachen von vier.[8]

Ebenfalls interessant ist der Vergleich der Startgeneration und der acht Schritte später. Hier erkennt man die Verdopplung des Blinkers an anderen Positionen. Ebenfalls verdoppelt sich die Figur der 3. Generation acht Schritte später.

[4] `https://rosettacode.org/wiki/Conway_27s_Game_of_Life/Forth` (Stand 08.09.2020)

[5] Im Wesentlichen dadurch, dass an dieser Stelle die „halben Zwischenzahlen“ gleich gewertet werden zu den „ganzen Zahlen“, also die 2 oder die 3 in der logischen Operation hier in Forth ein „ja“ erzeugt.

[6] Es wird ab der 0 gezählt und damit bis zu Generation 15.

[7] Generationsnummer = Bildunterschrift

[8] In komplexeren Strukturen bzw. Abbildungen auch in einem Vielfachen von 16 Schritten. Siehe dazu auch: Wójtowicz, Mirek, „Cellular Automation Rules Lexicon“

[9] Es sei noch der Ansatz in *APL* bzw. *lang5* erwähnt, der ebenfalls die Kopierwelt auf dem APL–Standard der GOL in lang5 aufbaut `http://lang5.sourceforge.net/tiki-view_forum_thread.php?comments_parentId=31` (Stand 08.09.2020).

Da sich die Welt in alle Richtungen ausbreitet, muss — will man weiter beobachten — die Welt entsprechend vergrößert werden, da diese sonst an den „Rand der Welt“ stößt.[9]

Man beobachtet dann weitere Kopien von den vorherigen Schritten. Es sei noch ergänzend erwähnt, dass auch andere Kombinationen von Figuren in anderen Welten ebenfalls Replikationen enthalten.[10] Die Kopierwelt ist eine der vielen faszinierenden Welten im *Spiel des Lebens* von John Conway und es gibt noch viel zu entdecken.

Listing

```
\ 1357 / 1357 Welt

\ Das schnelle Wrappen braucht quadratische
\ Dimensionen
1 6 lshift constant w \ 64
1 4 lshift constant h \ 16

: rows w * 2* ;
1 rows constant row
h rows constant size

create world   size allot
world    value old
old w +  value new

variable gens
: clear ( -- )
  world size erase 0 gens ! ;
: age ( -- )
  new old to new to old 1 gens +! ;

: col+ ( n1 -- n2 )
  1+ ;
: col- ( n1 -- n2 )
  1- dup w and + ; \ avoid borrow into row - Trick
: row+ ( n1 -- n2 )
  row + ;
: row- ( n1 -- n2 )
  row - ;

: wrap ( i -- i )
  [ size w - 1- ] literal and ;
: w@ ( i -- 0/1 )
  wrap old + c@ ;
: w! ( 0/1 i -- )
  wrap old + c! ;

: foreachrow ( xt -- )
size 0 do I over execute row +loop drop ;

: showrow ( i -- )
  cr
  old + w over + swap
  do I c@ if [char] * else bl then emit loop ;

: show ( i -- )
  [’] showrow foreachrow
  cr ." Generation " gens @ . ;

: sum-neighbors ( i -- i n )
  dup col- row- w@
  over row- w@ +
  over col+ row- w@ +
  over col- w@ +
  over col+ w@ +
  over col- row+ w@ +
  over row+ w@ +
  over col+ row+ w@ + ;

\ Regeln der Kopierwelt
: gencell ( i -- )
  sum-neighbors
  dup 1 =
  swap dup 3 =
  swap dup 5 =
  swap 7 = or or or
  1 and swap new + c! ;

: genrow ( i -- )
  w over + swap do I gencell loop ;

: gen [’] genrow foreachrow age ;

: geniter ( i -- )
  0 do gen show loop ;

: life begin gen 0 0 at-xy show key? until ;

\ Patterns 2D Umsetzung
  char | constant ’|’
: pat ( i addr len -- )
  rot dup 2swap over + swap do
  I c@ ’|’ = if drop row+ dup else
  I c@ bl = 1+ over w! col+ then
  loop 2drop ;



\ auswählbare Figuren

: blinker s" ***" pat ;
: toad s" ***| ***" pat ;
: pentomino s" **| **| *" pat ;
: pi s" **| **|**" pat ;
: glider s" *| *|***" pat ;
: pulsar s" *****|* *" pat ;
: ship s" ****|* *| *| *" pat ;
: pentadecathalon s" **********" pat ;
: clock s" *| **|**| *" pat ;



\ Start des Programmes

\ erschaffe einen Blinker an Position 1000:
clear 1000 blinker show

\ Laufe 16 Generationen lang und zeige den Output:
16 geniter

( Ende)
```

[10] Siehe die Übersicht hier: `https://www.ics.uci.edu/~eppstein/ca/replicators/`

Screenshots

```
$gforth main.fth -e bye
```

```
                    ***
```

```
Generation 0
```

```
                   * * *
                   ** **
                   * * *
```

```
Generation 1
```

```
                  ** * **

                  *** ***

                  ** * **
```

```
Generation 2
```

```
                 *   *   *
                 ***   ***
                   * * *
                 ** * * **
                   * * *
                 ***   ***
                 *   *   *
```

```
Generation 3
```

```
                ***  ***  ***
```

```
                  ***     ***


                  *** *** ***
```

```
Generation 4
```

```
                * *   *   * *
                ** * * * * **
                * *   *   * *

                * * *   * * *
                ** **   ** **
                * * *   * * *

                * *   *   * *
                ** * * * * **
                * *   *   * *
```

```
Generation 5
```

```
               ** ** *** ** **

               ***         ***

                   ** * **

               *** *** *** ***

                   ** * **

               ***         ***

               ** ** *** ** **
```

```
Generation 6
```

```
               *       *       *
               *** ** * * ** ***
                 * *   *   * *
               ** **       ** **
               * *     *     * *
                   ***   ***
               * * * * * * * * *
               ** * * * * * * * **
               * * * * * * * * *
                   ***   ***
               * *     *     * *
               ** **       ** **
                 * *   *   * *
               *** ** * * ** ***
               *       *       *
```

```
Generation 7
```

```
            ***             ***
```

```
Generation 8


            * * *           * * *
            ** **           ** **
            * * *           * * *


Generation 9


           ** * **         ** * **

           *** ***         *** ***

           ** * **         ** * **


Generation 10


          *   *   *       *   *   *
          ***   ***       ***   ***
            * * *           * * *
          ** * * **       ** * * **
            * * *           * * *
          ***   ***       ***   ***
          *   *   *       *   *   *


Generation 11


         *** *** ***     *** *** ***


         ***     ***     ***     ***


         *** *** ***     *** *** ***


Generation 12


        * *   *   * *   * *   *   * *
        ** * * * * **   ** * * * * **
        * *   *   * *   * *   *   * *

        * * *   * * *   * * *   * * *
        ** **   ** **   ** **   ** **
        * * *   * * *   * * *   * * *

        * *   *   * *   * *   *   * *
        ** * * * * **   ** * * * * **
        * *   *   * *   * *   *   * *


Generation 13

       ** ** *** ** ** ** ** *** ** **

       ***         *** ***         ***

           ** * **         ** * **

       *** *** *** *** *** *** *** ***

           ** * **         ** * **

       ***         *** ***         ***

       ** ** *** ** ** ** ** *** ** **


Generation 14
      *       *               *       *
      *** ** * * ** ** ** ** * * ** ***
        * *   *   * *   * *   *   * *
      ** **       ** * * **       ** **
      * *     *     *   *     *     * *
          ***   ***       ***   ***
      * * * * * * * *   * * * * * * * *
      ** * * * * * * * * * * * * * * **
      * * * * * * * *   * * * * * * * *
          ***   ***       ***   ***
      * *     *     *   *     *     * *
      ** **       ** * * **       ** **
        * *   *   * *   * *   *   * *
      *** ** * * ** ** ** ** * * ** ***
      *       *               *       *


Generation 15


     ***                             ***


Generation 16
```

TAQOZ — das eingebaute Forth des Propeller 2

Wolfgang Strauß

Nach über 13 Jahren Entwicklungszeit hat die Firma Parallax Inc. im November 2020 ihre zweite Mikrocontroller–Eigenentwicklung, genannt Propeller 2, offiziell vorgestellt und es gibt Produkte rund um den P2 im firmeneigenen Webshop zu kaufen. Dieser Chip hat erstaunliche Eigenschaften, die in diesem Artikel jedoch nur kurz behandelt werden sollen. Was mein Forth–Herz höher schlagen lässt, ist das Forth im ROM des Mikrocontrollers. Ja, es ist wahr: In jedem Propeller 2 ist ein Forth unlöschbar eingebrannt und steht für Experimente, Hardwaretests und Softwareentwicklung direkt zur Verfügung. Um die Vorstellung dieses Forths soll es im Folgenden gehen.

Unter der Haube

Der Propeller 2 hat dem Anwender eine Menge zu bieten. Die folgende Liste enthält nur einen Ausschnitt der Fähigkeiten des Chips als „Appetitanreger“. Für Details und weitere Funktionen sei auf das Datenblatt verwiesen.

- 8 Kerne (Cogs), 32–Bit, je 4 KBytes lokales RAM
- 180 MHz Takt (90 Mips/Kern), Übertaktung bis Faktor 2 möglich
- 512 KBytes Hauptspeicher (Hub–RAM) mit Zugriff für alle Cogs
- 16 KBytes ROM: Bootloader (seriell, SPI–Flash, SD–Karte), Monitor, TAQOZ–Forth
- CORDIC–Schaltkreis
  - Multiplikation
  - Division
  - Quadratwurzel
  - Rotation
  - Wandlung polar <–> kartesisch
  - Logarithmus
- 64 identische I/O–Pins (Smartpins). Jeder Pin ist mit autonomen Fähigkeiten ausgestattet:
  - Digital–Analog–Wandler (DAC)
  - Analog–Digital–Wandler (ADC)
  - USB Low–, Full–Speed
  - Digital gesteuerter Oszillator (NCO)
  - Pulsweitenmodulation (PWM)
  - Asynchrone Datenübertragung
  - Synchrone Datenübertragung
  - Quadraturdekoder

TAQOZ — wie es dazu kam

Die Fertigungsparameter des Chips sahen ein 16–KB–großes ROM vor. Der benötigte Bootloader und ein kleiner Monitor zur Ansicht und Manipulation des internen Speichers belegten aber nur 4 KB. Was also tun mit dem ungenutzten Speicher? Brach liegen lassen? Undenkbar! Der australische Hardwareentwickler Peter Jakacki hatte eine Idee. Sein Forth würde prima in den Bereich passen und durch die Interaktivität und speziell angepasste Worte schnelle Tests der Controller–Hardware sowie angeschlossener Peripherie erlauben. Das Konzept überzeugte den Chipentwickler Chip Gracey und so machte sich Peter ans Werk. Herausgekommen ist ein ungewöhnliches Forth, das der speziellen Architektur des Chips Rechnung trägt.

Der Name TAQOZ (gesprochen: Tackos) leitet sich ab aus Tachion (Name des Vorgängers) und Oz (umgangssprachliche Bezeichnung für Australien).

Verbindung herstellen

Für die Kommunikation zwischen dem PC und TAQOZ auf dem P2 braucht es nicht viel. Ein Terminalprogramm wie Minicom (Linux) oder TeraTerm (Windows) und ein USB–Seriell–Wandler reichen für eine interaktive Sitzung aus.

Nach einem Reset führt der Propeller 2 seine Bootsequenz aus. Je nach eingestellter Konfiguration, die er durch Abfrage von Widerständen ermittelt, lädt er Programme aus einem SPI–Flash, einer SD–Karte oder stellt ein Zeitfenster von 60 Sekunden für die Aufnahme einer seriellen Kommunikation zur Verfügung. Findet er kein Bootmedium bzw. treffen in dem Zeitfenster keine Zeichen über die serielle Schnittstelle ein, legt sich der Chip zur Ruhe und wartet mit minimaler Stromaufnahme auf den nächsten Reset. Also gilt es, dem Bootloader in angemessener Zeit zu geben, was er erwartet: die Zeichenfolge `"> "` ($3E, $20). Damit stellt sich der Bootloader auf die Baudrate des Senders (unser Terminalprogramm) ein. Die Geschwindigkeit muss zwischen 9600 Baud und 2 Megabaud liegen. Nun noch die Escape–Taste ($1B) gedrückt und TAQOZ begrüßt den neugierigen Abenteurer mit dem Start–Prompt:

```
Cold start
-------------------------------------------
 Parallax P2  .:.:--TAQOZ--:.:.  V1.1--v33h
```

```
-----------------------------------------
TAQOZ# --- ok
TAQOZ# --- ok
```

Erste Schritte, Stolperfalle

Was tut man, um sich einen Überblick über den enthaltenen Wortschatz seines Forths zu machen? Richtig. Man führt `words` aus. Das zeigt hier aber nicht den gewünschten Effekt. Es kommt aber auch keine Fehlermeldung. Die Erklärung: TAQOZ unterscheidet Groß– und Kleinschreibung und sowohl das Wort `words` als auch das Wort `WORDS` sind definiert.[1] Nach Eingabe von `WORDS` zeigt das Terminal dann endlich die gewünschte Liste.

```
 ---  ok
TAQOZ# WORDS ---
DUP OVER SWAP ROT -ROT DROP 3RD 4TH 2DROP 3DROP NIP 2SWAP 2DUP ?DUP AND
ANDN OR XOR ROL ROR >> << SAR 2/ 2* 4/ 4* 8<< 16>> 8>> 9<< 9>> REV |< >|
>N >B >9 BITS NOT = <> 0= 0<> 0< < U< > U> <= => WITHIN DUPC@ C@ W@ @ C+!
C! C@++ W+! W! +! ! BIT! SET CLR SET? 1+ 1- 2+ 2- 4+ + - UM* * W* / U/
U// // */ UM// C++ C-- W++ W-- ++ -- RND GETRND SQRT SETDACS ~ ~~ W~ W~~
C~ C~~ L>S >W L>W W>B W>L B>W B>L MINS MAXS MIN MAX ABS -NEGATE ?NEGATE
NEGATE ON TRUE -1 FALSE OFF GOTO IF ELSE THEN BEGIN UNTIL AGAIN WHILE REPEAT
SWITCH CASE@ CASE= CASE> BREAK CASE ADO DO LOOP +LOOP FOR NEXT ?NEXT I
J LEAVE IC@ I+ BOUNDS H L T F R HIGH LOW FLOAT PIN@ WRPIN WXPIN WYPIN RDPIN
RQPIN AKPIN WAITPIN WRACK PIN @PIN ns PW PULSE PULSES HILO DUTY NCO HZ
KHZ MHZ MUTE BLINK PWM SAW BIT BAUD TXD RXD TXDAT WAITX WAITCNT REBOOT
RESET 0EXIT EXIT NOP CALL JUMP >R R> >L L> !SP DEPTH COG@ COG! LUT@ LUT!
COGID COGINIT COGSTOP NEWCOG COGATN POLLATN SETEDG POLLEDG KEY WKEY KEY!
CON NONE COM CONKEY CONEMIT SEROUT EMIT EMITS CRLF CR CLS SPACE SPACES
RAM DUMP: DUMP DUMPW DUMPL DUMPA DUMPAW QD QW DEBUG lsio COG LUT KB MB
M . PRINT .AS .AS" .DECL .DEC4 HOLD #> <# # #S <D> U. .DEC .BIN .H .B .BYTE
.W .WORD .L .LONG .ADDR PRINT$ LEN$ " ." CTYPE ?EXIT DATA? ERASE FILL CMOVE
<CMOVE s ms us CNT@ LAP LAP@ .LAP .ms HEX DEC BIN .S WORDS @WORDS GET$
SEARCH $># @DATA HERE @HERE @CODES uemit ukey char delim names TASK REG
@WORD SPIN | || , [W] ["] NULL$ $! $= ASM FORGET CREATE$ CREATE VAR pub
pri pre : ; [ ] ' := ==! ALIGN DATCON ALLOT org bytes words longs byte
word long res [C] GRAB NFA' CPA CFA \ --- ( { } IFNDEF IFDEF TAQOZ TERM
AUTO SPIRD SPIRDL SPIWB SPICE SPIWC SPIWW SPIWM SPIWL SPIPINS SPIRX SPITXE
SPITX WAIT CLKDIV RCSLOW HUBSET WP WE CLKHZ ERROR SFPINS SF? SFWE SFINS
SFWD SFSID SFJID SFER4 SFER32 SFER64 SFERASE SFWRPG BACKUP RESTORE SFRDS
SFWRS SFC@ SFW@ SF@ SF .SF SDBUF sdpins MOUNT DIR !SD !SX SD? CMD ACMD
cid SDWR SDRDS SDWRS FLUSH FOPEN FLOAD FGET FREAD FWRITE SECTOR SDRD SDRDS
SDADR SD@ SD! SDC@ SDC! SDW@ SD @FAT @BOOT @ROOT fat END  432 ok
TAQOZ#
```

432 Worte fasst das Dictionary. Damit lässt sich eine Menge anstellen. Bei den meisten Worten funktioniert übrigens auch die Kleinschreibung, da das Forth bei erfolgloser Suche das Wörterbuch noch ein zweites Mal durchforstet, dann aber mit in Großbuchstaben umgewandeltem Token.

Besonderheiten

Es folgt eine Sammlung von Dingen, die mir besonders aufgefallen sind. Für den Einen oder Anderen mögen sie alltäglich sein, mich haben sie zum Nachdenken oder Schmunzeln gebracht.

Literale, Nummernrepräsentation

Mögliche Darstellung der Dezimalzahl 42 im Quelltext:

```
TAQOZ# 42 #42 42d        \ Dezimale Notation
TAQOZ# $2A 2Ah           \ Headezimale Notation
TAQOZ# %101010 101010b   \ Binäre Notation
TAQOZ# '*'               \ Zeichenliteral
```

Beliebige Zeichen können zur Erhöhung der Lesbarkeit eingefügt werden, wenn folgende Bedingungen erfüllt sind: nicht am Anfang, nicht am Ende, kein Zeichen aus dem Vorrat der gewählten Zahlenbasis.

```
TAQOZ# %1001_0110-1001_0100
```

Weitere außergewöhnliche Darstellungsformen:

```
TAQOZ# ^C       \ CTRL-C: legt 3 auf den Stack
TAQOZ# &192.168.0.3  \ IP-Adresse in 32 Bits
TAQOZ# 192:168:0:3   \ dito
TAQOZ# #P24     \ Pin 24: legt 24 auf den Stack
```

Kommentare

Vernünftig gewählte Kommentare gehören zu einem les– und verstehbaren Quelltext dazu. TAQOZ bietet eine gute Unterstützung. Erwähnenswert ist hier `---`. Es ist zwar nur ein Alias des bekannten \ und ich habe den Sinn erst nicht verstanden, aber bei einem Copy&Paste–Vorgang fiel der Groschen:

```
TAQOZ# 12 . --- 12 ok
```

Wie man sieht, leitet TAQOZ seine Ausgaben mit `---` ein. Ich kann also einfach die Zeile bis zum Ende kopieren und danach wieder einfügen, ohne dass es Fehler hagelt.

Hier die Liste der verfügbaren Kommentarworte:

`\` Kommentar bis Zeilenende

`---` Alias zu „\“. Nützlich für Copy & Paste

`(` Inline–Kommentar bis zur nächsten schließenden Klammer oder bis Zeilenende

`{` Mehrzeiliger Kommentar bis zur schließenden geschweiften Klammer

`[IFDEF] name` Wenn sich `name` nicht im Wörterbuch befindet, wird der folgende Bereich bis zur nächsten schließenden geschweiften Klammer auskommentiert

`[IFNDEF] name` Wie `[IFDEF]`, aber mit umgekehrter Logik

Namenswahl

Wer selbst ein Forth entwickelt, ist gut beraten, wenn er oder sie die Namen der Primitiven möglichst nach dem Standard wählt. Das hilft, Verwirrung zu vermeiden. Peter Jakacki hat einige Worte nach den entsprechenden Funktionen oder Operatoren der Programmiersprache SPIN[2] benannt, u. a., um sein Forth für Interessenten gefälliger aussehen zu lassen.

Hier eine Auswahl:

`>>`	wie `rshift`
`<<`	wie `lshift`
`byte`	definiert eine 8–Bit–Variable
`word`	definiert eine 16–Bit–Variable
`long`	definiert eine 32–Bit–Variable: wie `variable`
`:=`	definiert eine 32–Bit–Konstante: wie `constant`

[1] `words` (kleingeschrieben) definiert eine Array–Variable mit Wortbreite (16 Bit). Beispiel: `4 words BUFFER`

[2] Die Programmiersprache SPIN wurde speziell für die Propeller–Mikrocontroller entwickelt und hat eine treue Fangemeinde.

`==!`	Konstanten können geändert werden: `2 ' SPEED ==!`
`| || ,`	ein Byte, Word, Long compilieren

Tastenkürzel

Für oft benutzte Aktionen gibt es Tastenkürzel. Das ist praktisch und ich habe mich so daran gewöhnt, dass ich diese Funktionalität in anderen Forth–Systemen vermisse.

Eine Auswahl:

`^X`	wiederholt die letzte Zeile
`^W`	führt `WORDS` aus
`^S`	Datenstack leeren
`^Q`	Datenstack anzeigen
`^C`	Reset
`ESC`	eingegebene Zeile verwerfen

Stapelvielfalt

TAQOZ hat neben dem Daten– und Returnstack zwei weitere Stapel: den LOOP–Stack und den Branch–Stack. Dadurch wird der Return–Stack entlastet. Er enthält normalerweise nur noch die Rücksprungadressen der Wörter. Das vermindert die Fehleranfälligkeit des Programms. Die Worte `>R` und `R>` zur Zwischenspeicherung eines Wertes auf dem Return–Stack gibt es weiterhin, es sollten jedoch besser die Worte `>L` und `L>` verwendet werden. Sie nutzen den LOOP–Stack als Zwischenspeicher. Der LOOP–Stack hält den Index und das Limit einer `DO..LOOP`–Schleife. `DO` legt den Instruction–Pointer auf dem Branch–Stack ab. `LOOP` kann dann einfach dorthin zurückspringen. `DO` und `LOOP` sind ganz normale Worte und erledigen wie diese alle Arbeit zur Laufzeit.

Das Dictionary enthält keinen Code

Traditionell bestehen die Worte in einem Dictionary unter anderem aus einem Header und dem Codefeld. Im Header finden sich typischerweise der Name des Wortes, die Länge des Namens, Flags und der Verweis auf den Header des davor definierten Wortes.

In TAQOZ sind Code und Dictionary getrennt. Dadurch ergeben sich zwei in der Größe veränderliche Bereiche im Speicher. Code wächst in Richtung aufsteigender Adressen auf den Dictionarybereich zu und das Dictionary wächst in Richtung kleiner werdender Adressen. Treffen die beiden Bereiche aufeinander, ist der Speicher aufgebraucht und es können keine neuen Worte definiert werden.

Welchen Vorteil hat das Auslagern des Codes? Durch die Trennung von Header und Codefeld ist das Dictionary nun eine einfache Hintereinanderreihung von Headern. Nicht mehr benötigte Wortnamen können entfernt werden und durch Zusammenschieben der Lücken kann Speicher zurückgewonnen werden. Worte, welche den Code der entfernten Header verwenden, funtionieren natürlich weiterhin, denn der Code wird ja nicht entfernt.

Wie sieht das in der Praxis aus?

```
pri a  ( n1 -- n2 ) 10 * 1+  ;

pub b  ( n1 -- n2 ) DUP a  SWAP a  *  ;

WORDS

RECLAIM

WORDS
```

`pri` (private) startet eine Colon–Definition und setzt ein spezielles Flag. Damit ist der Header für eine eventuelle Entfernung vorgemerkt.

`pub` (public) ist ein Alias für die normale Colon–Definition `:` .

`RECLAIM` (zurückfordern) entfernt alle Header aus dem Dictionary, die das Flag gesetzt haben. `a` ist jetzt „privat“ und kann nicht mehr aufgerufen werden.

Interaktive Compilierung

Normalerweise läuft es doch so: Forth empfängt und sammelt eine Zeile mit Text. Nach dem Drücken der Enter–Taste wird die Zeile vom äußeren Interpreter in Worte unterteilt und diese dann ausgeführt. Das ist schön einfach, hat aber auch Nachteile:

```
PinHigh  5 us  PinLow
```

Hier soll ein Ausgangspin für 5 µs ein High–Potential annehmen. Wenn man sich das Signal auf einem Logikanalysator anschaut, wird man feststellen, dass der Pin wesentlich länger als 5 µs aktiv ist. Der Grund ist einfach. Gehen wir die Zeile aus der Sicht des Interpreters durch. Der Text „PinHigh“ wird im Dictionary gesucht, gefunden und der entsprechende Code ausgeführt. Der Pin ist High und es tickt die Zeit. Der Text „5“ wird im Dictionary gesucht, nicht gefunden und dann in die Zahl 5 gewandelt und auf den Stack gelegt. Das alles hat wahrscheinlich schon länger als 5 µs gedauert. Aber wir müssen noch die Worte `us` und `PinLow` finden und ausführen, bis der Pin in seinen Ruhezustand zurückwechselt.

```
: puls  ( -- ) PinHigh  5 us  PinLow ;  puls
```

Hier ist das Timing wesentlich exakter, da schon alles compiliert ist und der Code mit voller Geschwindigkeit laufen kann, wenn `puls` ausgeführt wird. Um dieses zu erreichen, mussten wir aber unseren Code in eine Definition einbauen. Das ist umständlich.

TAQOZ verwendet keinen Zeilenpuffer, sondern einen Wortpuffer. Wenn ein Wort komplett ist (erkennbar an einem Whitespace–Zeichen), wird es in einen speziellen Codebereich compiliert. Beim Empfang des Zeilenendes wird der Code einfach ausgeführt.

Die Vorteile:

- Die Prozessorlast ist während des Empfangs einer Zeile besser verteilt. Dadurch sind höhere Übertragungsgeschwindigkeiten möglich.
- Code läuft immer mit voller Geschwindigkeit
- „Compile–Only“–Worte wie `IF`, `ELSE`, `THEN`, `."` oder auch Schleifen können nun auch interaktiv benutzt werden.

Sonstige Besonderheiten

- Die maximale Länge von Wortnamen ist 16 Zeichen.
- Der Datenstack wird nach einem Fehler nicht geleert. Dafür gibt es das Wort `!SP`.

Spaß mit Smartpins

Ich musste es mehrfach lesen, bis ich es geglaubt habe: Der Propeller 2 hat an jedem seiner 64 I/O–Pins komplexe Schaltkreise verbaut, die selbstständig gängige Aufgaben erledigen können (siehe Liste am Anfang des Artikels). Alles ist also 64fach vorhanden. Mal eben 16 Analogsignale einlesen (parallel, nicht gemultiplext!) und die Ergebnisse über 16 UARTs mit bis zu 100 MBits/s „rausblasen“? Das Beispiel mag praxisfremd erscheinen; es soll auch nur die Möglichkeiten zeigen. Zum Warmwerden bieten sich die folgenden Beispiele an.

Smartpin–Blinkerei

Blinkende Leuchtdioden sind das „Hello world!“ des Hardwareentwicklers. Also folgen wir dem Brauch:

```
TAQOZ# 56 BLINK ---  ok
```

Die LED an PIN 56 blinkt mit 2 Hz. `BLINK` ist definiert als

```
: BLINK  ( pin# -- )
  PIN 2 Hz  ;
```

`PIN` nimmt eine Zahl vom Stack und speichert sie. Eine Reihe von Worten verwendet diesen gespeicherten Wert, deshalb braucht `PIN` nur ausgeführt zu werden, wenn sich der zu verwendende Pin ändert.

`Hz` nimmt eine Zahl vom Stack und benutzt sie zur Konfiguration des NCOs (Numerically–Controlled Oscillator) des gesetzten Pins. Ergebnis ist hier eine Ausgabefrequenz von 2 Hz.

Darf es auch ein wenig mehr sein?

```
TAQOZ# 10 HZ ---  ok
```

Die Frequenz erhöht sich auf 10 Hz. Der verwendete Pin ist weiterhin P56.

```
TAQOZ# MUTE ---  ok
```

`MUTE` beendet die Aktivität des momentan gewählten Smartpins.

Einen Pin blinken zu lassen ist ja ganz nett, aber wie wäre es mit 32 Pins?

```
TAQOZ# 32 0 DO I PIN I 4 / 2+ HZ LOOP ---  ok
```

Hier blinken oder flimmern nun 32 Ausgänge vor sich hin. Die CPU wurde nur für die Konfiguration der Smartpins benötigt. Die 32 NCOs der Pins erledigen die Ausgabe der Frequenzen autonom. Sie arbeiten auch nach einem Forth–Warmstart weiter.

So kehrt Ruhe ein:

```
TAQOZ# 32 0 DO  I PIN  MUTE  LOOP ---  ok
```

Smartpin–PWM

Die Ausgabe eines pulsweitenmodulierten Signals (PWM) ist ein weiterer Anwendungsfall für die Smartpins.

```
TAQOZ# 56 PIN  10 100 25 PWM ---  ok
```

Die Ausgabe geht an Pin 56. Die weiteren Parameter bestimmen die Pulsweite (10 Takte), die Rahmenlänge (100 Takte) und den Taktteiler (/25). Rahmenlänge und Pulsweite wurden so gewählt, dass sich ein Puls–Pause–Verhältnis von 10 % ergibt. Die einstellbare Auflösung ist 1 %. Der Systemtakt (25 MHz) wird durch 25 und dann noch einmal durch die Rahmenlänge (100) geteilt, sodass das PWM–Signal mit 10 kHz ausgegeben wird. Eine Vergrößerung der Rahmenlänge erhöht die Auflösung und verringert die Ausgabefrequenz.

```
TAQOZ# 10 100 10000 PWM ---  ok
```

Durch den höheren Vorteiler ist jetzt die Ausgabefrequenz auf 25 Hz heruntergesetzt. Die LED erscheint gleich hell, an dem Tastverhältnis von 10 % hat sich ja nichts geändert.

```
TAQOZ# 1 100 10000 PWM ---  ok
```

Das Tastverhältnis ist jetzt 1 %, die Leuchtdiode erscheint dunkler, die Ausgabefrequenz ist noch immer 25 Hz.

```
TAQOZ# 4 0 DO I PIN
              I 1+ 20 * 100 25 PWM LOOP --- ok
```

Die Schleife setzt 4 Pins (P0 bis P3) auf verschiedene Tastverhältnisse von 20 % bis 80 %.

Ein weiterer nützlicher Modus ist die Erzeugung einer bestimmten Anzahl Pulse mit definierter Breite und Abstand:

```
TAQOZ# 0 PIN  1000 3000 HILO  3 PULSES --- ok
```

Hier werden 3 Pulse mit einer Länge von jeweils 1000 Takten ausgegeben. Die Pause zwischen den Pulsen beträgt 3000 Takte.

Smartpin–UART

Jeder der 64 Smartpins hat u. a. einen seriellen Transmitter eingebaut. Einstellbar von 1 bis 32 Datenbits, die Baudrate kann bis zu 100 MBd betragen.

```
TAQOZ# 48 PIN  1 M TXD --- ok

TAQOZ# " Hello World!" DUP LEN$ TXDAT --- ok
```

Pin 48 wird als Transmitter mit 1 MBd konfiguriert. Die Standarddatenbreite von 8 Bit wird nicht verändert. In der zweiten Zeile erfolgt dann die Ausgabe der Daten.

Smartpin–Spannungsausgabe

Jeder Smartpin hat u. a. einen Digital–Analog–Wandler mit einer Auflösung von 12 Bit eingebaut. Es können Spannungen von 0 V bis 3,3 V mit einer Auflösung von 0,8 mV ausgegeben werden.

```
TAQOZ# 52 PIN 1250 mV
```

Mit dieser kurzen Zeile erscheint die gewählte Spannung an Pin 52.

Diagnose

TAQOZ hat hilfreiche Worte für die Programmentwicklung in der Werkzeugkiste:

`WORDS` Listet alle definierten Worte auf.

`DUMP` Zeigt den ausgewählten Speicherbereich in verschiedenen Formaten.

`.S` Listet den Inhalt des Datenstapels auf.

`DEBUG` Zeigt den Inhalt des Daten–, Return– und LOOP–Stacks, der Task–Register und einen Ausschnitt des aktuellen Codes und des Dictionarys.

`lsio` Zeigt den Pegel aller I/O–Pins an.

`LAP` , `.LAP` Stoppuhr zum Vermessen von Laufzeiten.

`RESET` Setzt den Propeller 2 zurück.

DUMP

Eines der nützlichsten Werkzeuge zur Programmentwicklung sind Worte zur Anzeige von Speicherbereichen. TAQOZ überschüttet den Entwickler förmlich mit Worten zur Speicherinhaltsanzeige in verschiedenen Formaten. Eine Besonderheit sind die Modifizierer, die die Datenquelle umschalten. So sind nicht nur Abbilder der verschiedenen internen Speicher darstellbar, sondern auch von SD–Karten und angeschlossenem SPI–Flash.

`DUMP ( addr bytes -- )` Anzeige 16 Bytes/Zeile plus ASCII

`DUMPW ( addr bytes -- )` Anzeige als 16–Bit–Words plus ASCII

`DUMPL ( addr bytes -- )` Anzeige als 32–Bit–Longs plus ASCII

`DUMPA ( addr bytes -- )` Anzeige ASCII — 64 Zeichen/Zeile

`DUMPAW ( addr bytes -- )` Anzeige ASCII — 128 Zeichen/Zeile

`QD ( addr -- )` Quick `DUMP` 32 Bytes in 2 Zeilen

`QW ( addr -- )` Quick `DUMPW` 32 Bytes in 2 Zeilen

Es folgen die Modifizierer. Sie werden nach jedem `DUMP` wieder auf den Standard (RAM) zurückgestellt:

`RAM` Wähle den Hauptspeicher (Hub–RAM) als `DUMP`–Speicher (default)

`COG` Wähle den Registerspeicher eines Kerns als `DUMP`–Speicher

`LUT` Wähle den Look–up–Speicher eines Kerns als `DUMP`–Speicher

`SF` Wähle das externe SPI–Flash als `DUMP`–Speicher

`SD` Wähle die SD–Karte als `DUMP`–Speicher

Beispiele:

```
TAQOZ# $FC000 $20 DUMP ---
000F_C000: 00 08 80 FF  3F 00 0C FC
           32 C8 06 F6  1F 04 DC FC
           '....?...2.......'
000F_C010: 40 7E 74 FD  01 CA A6 F0
           1F CA 26 F4  00 CA 62 FD
           '@~t.......&...b.' ok

TAQOZ# 0 8 LUT DUMP ---
 000: DEAD_BEE3 DEAD_BEE2 BD5B_7DD1 DEAD_BEEF
      DEAD_BEEF 0000_DEAD FFFF_FFFF 9A98_D6A1
  ok
```

Was ist an den I/O–Pins los?

Eine schnelle Übersicht über die Pegel an den I/O–Pins ist hilfreich für das Debugging externer Hardware. Das Wort `lsio` macht es sehr einfach:

```
TAQOZ# \ zeige IO-Pin-Pegel
TAQOZ# lsio ---
P:000000000011111111112222 .. 44555555555566
P:012345678901234567890123 .. 89012345678901
=:~h~~~~~ddd~~~~~~~~~~~~~~ .. ~~~~~~~~~~~~~~
ok
```

Hier ist an Pin 1 ein Pullup–Widerstand verbaut und die Pins 7 bis 9 werden durch Pulldowns nach Masse gezogen.

Setzen der Taktfrequenz

Wenn TAQOZ startet, kennt es die genaue Taktfrequenz nicht. Beim Start verwendet der Propeller 2 seinen internen RCFAST–Oszillator. TAQOZ hat eine Konstante `CLKHZ` , die den Wert 20.000.000 (20 MHz) enthält. Das ist der vom Chiphersteller garantierte minimale Takt. Der nominale Wert ist 25 MHz, welcher von den Serienchips auch einigermaßen genau eingehalten wird. Um beim Experimentieren ein genaues Timing zu haben, muß also die reale Frequenz gemessen und in die Konstante `CLKHZ` eingetragen werden. Das nachträgliche Ändern von Konstanten ist traditionell in Forth nicht vorgesehen, wird von TAQOZ aber unterstützt.

```
TAQOZ# 56 PIN  1 HZ  60 WXPIN --- ok
```

An Pin 56 ist eine LED angeschlossen. Sie soll jeweils 60 s an und 60 s dunkel sein. Das wird mit einer Stoppuhr nachgemessen.

Für dieses Beispiel wird davon ausgegangen, dass 48 s gemessen wurden.

```
TAQOZ# CLKHZ 60 48 */ . --- 25000000 ok
```

Die Konstante CLKHZ muss also auf den Wert 25.000.000 gestellt werden.

Das Wort ==! weist Konstanten einen Wert zu.

```
TAQOZ# 25,000,000  ‘ CLKHZ ==! ok

TAQOZ# CLKHZ . --- 25000000 ok
```

Messen der Ausführungsgeschwindigkeit

Wieviel Zeit verbrät denn nun meine neue verbesserte Routine? Diese Frage ist mit TAQOZ schnell beantwortet. Die Worte LAP und .LAP sind alles, was es braucht. Einfach ein LAP vor und hinter den zu testenden Code setzen und das Messergebnis mit .LAP ausgeben.

```
TAQOZ# LAP 1,000,000 FOR NEXT LAP .LAP ---
                     32,000,147 cycles ok

TAQOZ# 1,000,000 LAP FOR NEXT LAP .LAP ---
                     32,000,082 cycles ok

TAQOZ# LAP 1,000,000 LAP .LAP ---
                    58 cycles ok
```

Externes SPI–Flash testen

Der Propeller 2 hat keinen internen Flash–Speicher. Er verfügt über 512 Kilobytes RAM, welches gemeinsam von den 8 Kernen benutzt werden kann, sowie 4 KBytes lokalen Speicher für jeden Kern. Es ist also ein externer nichtflüchtiger Speicher nötig, von dem der P2 beim Start oder auch später Programmcode zur Ausführung in sein RAM laden kann. Deshalb hat jede P2–Platine, die etwas auf sich hält, einen flashbasierten Speicher aufgelötet. Gängige Größen sind 512 KB bis 16 MB.

Kein Wunder also, dass TAQOZ entsprechende Worte zum Testen und Bewirtschaften dieser Bausteine bereithält.

```
TAQOZ# \ zeige die SPI-Flash-ID
TAQOZ# .sf ---
       $EF70_1800 $0F5F_4423_$E468_5CF4 ok

TAQOZ# \ zeige einen SPI-Flash-Speicherauszug
TAQOZ# 0 $40 sf dump ---
00000: 44 63 93 \ \33 04 83 0A  ’Dc.\ \b)3...’
00010: 4A F9 97 B\ \F 65 76 01  ’J...\ \B.ev.’
00020: CE 44 46 42\ \ 04 56 BB  ’.DFB.\ \..V.’
00030: 41 1E 52 48 \ \C7 EC 66  ’A.RH.’\ \..f’
ok
```

SD–Karte testen

Die Alternative oder Ergänzung zum SPI–Flash ist die SD–Karte. Sie ist klein, preiswert und kann riesige Datenmengen aufnehmen. Viele P2–Boards sind mit einer Mikro–SD–Kartenhalterung bestückt. TAQOZ hat Worte für die Kommunikation mit SDHC–Karten[3] eingebaut.

```
TAQOZ# \ initialisiere FAT32-Filesystem
TAQOZ# MOUNT ---
       .SDSM32G 6130_3535 NO NAME 32k 30,432M
       ok
```

Acht Freunde

Der Propeller 2 ist ein 8–Kern–Prozessor. Die bisherigen Beispiele haben davon allerdings keinen Gebrauch gemacht; es wurde stillschweigend nur ein Kern genutzt, auf dem auch die Forth–Konsole lief. Hier soll nun gezeigt werden, wie weitere Kerne aktiviert und genutzt werden können.

TAQOZ läuft nach dem Start auf dem Kern #0. Die Kerne #1 bis #7 sind inaktiv.

```
TAQOZ# 1 NEWCOG
```

Kern #1 wurde aktiviert, mit einer weiteren Instanz von TAQOZ geladen und wartet nun darauf, einen Task zugewiesen zu bekommen.

Der Task:

```
TAQOZ# : BLINKER  ( -- )
TAQOZ#   BEGIN
TAQOZ#     56 HIGH  100 ms
TAQOZ#     56 LOW   100 ms
TAQOZ#   AGAIN  ;
```

Eine Endlosschleife also, die auf die altertümliche Art (ohne Smartpins) die LED an Pin 56 blinken lässt. Nicht besonders schön, erfüllt aber hier den Zweck.

Nun noch dem Kern #1 die Adresse des Tasks übermitteln und es geht los:

```
TAQOZ# ‘ BLINKER  1 TASK  W!
```

Und so wird man die Geister, die man rief, wieder los:

```
TAQOZ# 1 COGSTOP
```

Zum Schluss noch eine Definition, die das Starten und Ausführen in einem Rutsch erledigt. Zu beachten ist, dass das Aktivieren eines Kerns und Laden des Forth–Kernels etwa 5000 Systemtakte in Anspruch nimmt. Bei interaktiver zeilenweiser Eingabe (siehe oben) ist das kein Problem, da kein Mensch so schnell tippen kann. In einer Definition muss dem Kern die Zeit gegeben werden, bevor er einen Task zugewiesen bekommt:

```
TAQOZ# : RUN  ( task core -- )
TAQOZ#   \ ’task’ in Kern ’core’ ausführen
TAQOZ#   DUP NEWCOG
TAQOZ#   5000 WAITX  \ 5000 Takte warten
TAQOZ#   TASK W!  ;
```

Test, dieses Mal mit Kern #4:

```
TAQOZ# ‘ BLINKER  4 RUN

TAQOZ# 4 COGSTOP
```

[3] SDHC (Secure Digital High Capacity), Kapazität von 4 GBytes aufwärts, kleinere Karten (SD) werden nicht unterstützt.

Schnappschuss

TAQOZ wird aus dem ROM kopiert und läuft danach komplett im RAM. Ein Abschalten der Versorgungsspannung hat folglich den Verlust aller gemachten Änderungen zur Folge. Das Forth enthält deshalb Funktionen zum Sichern und Wiederherstellen des Systemzustandes.

`BACKUP` Speichert das Forth–System auf das externe SPI–Flash

`RESTORE` Stellt den zuvor gesicherten Zustand wieder her.

TAQOZ aufrüsten

Das residente Forth im Propeller 2 hat mit 432 Worten schon einen üppigen Funktionsumfang, mit dem es sich gut arbeiten lässt. Wünscht man mehr Unterstützung, wie z. B. Videoausgabe oder eine komplette SD–Karten–Bibliothek, gibt es eine stark erweiterte Version namens „TAQOZ Reloaded“ zum Nachladen. Dort bleiben dann kaum noch Wünsche offen. Die Version wird von Peter Jakacki aktiv weiterentwickelt.

TAQOZ Reloaded auf einen Blick

- Schnelle interaktive Entwicklung
- On–The–Fly–Kompilierung macht Einzeiler mit voller Ausführungsgeschwindigkeit möglich
- Trace–Level–Debug mit Dekompiler
- Multi–Core–Ausführung von Code
- FAT32– und SD–Formatierungswerkzeuge und Berichterstellung
- VGA–Bitmap–Texte und –Grafiken
- WIZnet W5500–Ethernet–Treiber inkl. Telnet, FTP, HTTP–Server

Fazit

Der Propeller 2 ist ein leistungsfähiger Chip mit einem ungewöhnlichen Konzept. Er hat eine aktive Fangemeinde, die sich in den Foren der Fa. Parallax Inc. austauscht. Um die umfangreichen Möglichkeiten des Mikrocontrollers zu erkunden, ist das TAQOZ–Forth bestens geeignet. Man merkt, dass es von einem Praktiker für die Praxis entwickelt wurde. Mein Dank geht an Peter Jakacki für ein gelungenes Forth und Bob Edwards für seine Einführung „The Bit Bashers Guide to the Parallax P2 — Using TAQOZ ROM Forth“, aus der ich mir das eine oder andere Beispiel „ausgeliehen“ habe.

Links

Propeller 2–Chips, Hardware, Datenblatt, Forum:
`https://www.parallax.com`

TAQOZ–Forth, Quelltext, Dokumentation:
`https://sourceforge.net/projects/tachyon-forth`

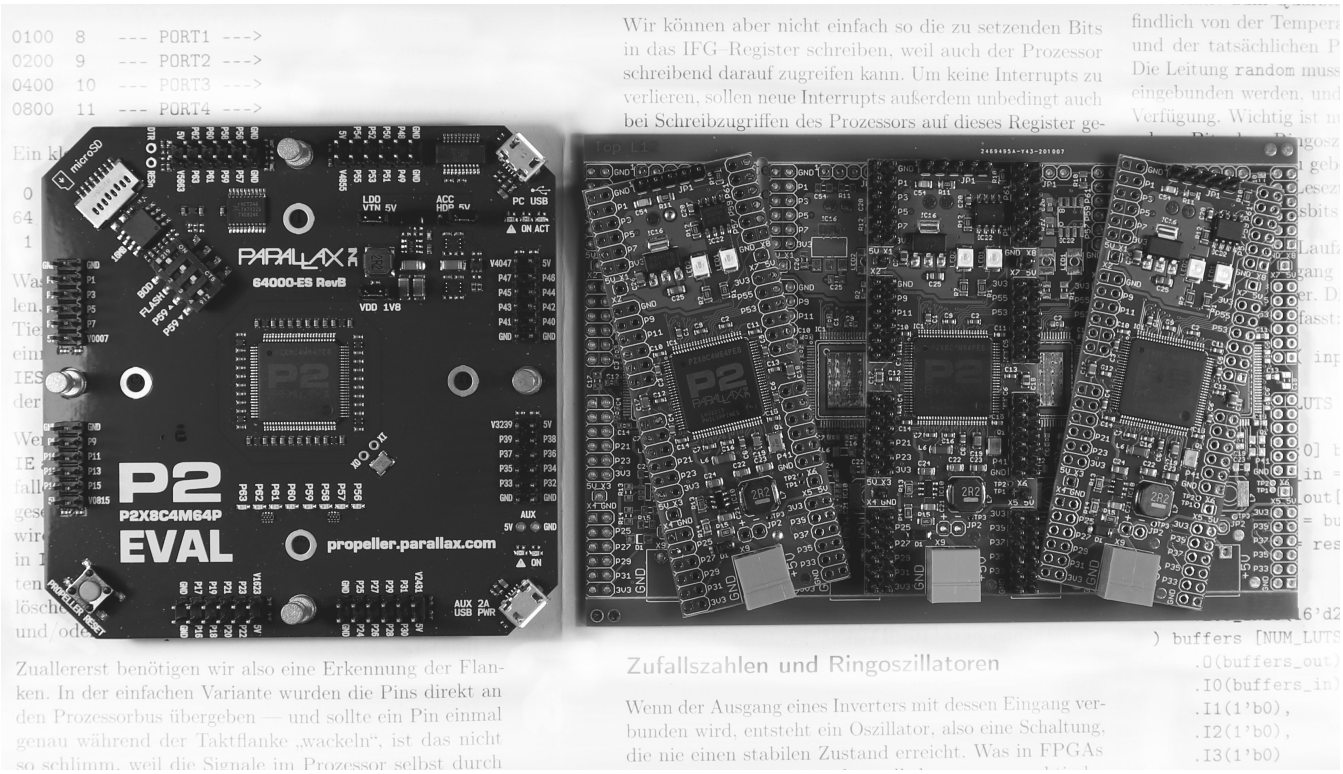


Abbildung 1: Links das Evaluierungsboard der Fa. Parallax Inc. für den Propeller 2, rechts KISS–EVAL–Boards des Users „ManAtWork“ aus dem Parallax–Forum.

Factoring, oder warum der Trute–Recognizer nicht zu viel macht

Bernd Paysan

Klaus Schleisiek *hat in der letzten VD seinen „Arme–Leute–Recognizer“ vorgestellt [1]. Der auf seinem bisher gepatchten Interpreter basiert. Er empfindet den* Trute*–Recognizer als zu komplex und zu schwer zu verstehen, wobei seine Kritik an den verschiedenen Versionen ins Leere läuft: Das ist nicht etwa dreimal erklärt, das sind drei Versionen innerhalb laufender Entwicklung. Die Entwicklung ist auch sicher noch nicht ganz abgeschlossen. Aber auch andere Debatten zeigen: Es ist Bedarf nach Erklärung, vor allem, wie man existierenden Code anpasst.*

Wie faktorisiert man einen Recognizer?

Ein Recognizer erkennt ein Lexem im Eingabestrom, und produziert ein Token, das im nächsten Schritt weiter verarbeitet werden kann. Das Weiterverarbeiten kann dann Interpretieren, Compilieren oder Postponen sein (abhängig vom Zustand des äußeren Interpreters), und wird vom äußeren Interpreter selbst erledigt. Oder von einem anderen Interpreter, der auch die Recognizer benutzt.

Die Struktur ist dabei so gewählt, dass der Programmierer jede dieser Teilaktionen als eigenes Wort implementiert, und daraus einen Rectype, einen Token–Deskriptor, generiert. Das Dispatchen macht eben der äußere Interpreter, da das für alle Rectypes die gleiche Aktion ist. Die Rectypes lassen sich beim normalen Erweitern des Recognizer–Stacks gut wiederverwenden. Auch die einzelnen Wörter sind öfter wiederverwendbar.

Dass Faktorisierung Programme „unübersichtlich“ macht, habe ich schon von einem Fortran–Programmierer gehört. Weil man dem Programm nicht linear folgen konnte, sondern in die einzelnen Unterprogramme musste. Von Forth–Programmierern ist das eine etwas ungewohnte Kritik.

Klaus' Arme–Leute–Recognizer macht das alles selbst, wobei jeweils ein Wort für Erkennen und Dispatchen eines bestimmten Tokens zuständig ist. Dabei ist das bereits Factoring, denn im ersten Listing war `target-compiler` noch ein Wort (ohne Factoring). Aber halt nicht weit genug. Tatsächlich zeigt etwa sein `target-number` eine klare Struktur:

```
: target-number ( addr len -- addr len | rdrop )
  2dup 2>r  snumber? ?dup
  0= IF  2r> EXIT  THEN  2rdrop rdrop
  comp? IF 0>  IF  d>target swap t_lit,  THEN
           t_lit,  EXIT  THEN
  dbg?  IF 0>  IF  d>target swap >t      THEN
           >t       EXIT  THEN
  drop
;
```

Wir sehen hier erst den Recognizer selbst, der im wesentlichen aus `snumber?` besteht (alles andere ist Glue–Code). Und wir sehen zwei Aktionen, eine für das Compilieren, und eine für das Debuggen, was im Target–Compiler in etwa dem Interpretieren entspricht (nur halt, dass der Code dann auf dem Target abläuft). Die gleiche Struktur finden wir auch bei `host-number`, nur dass hier ganz traditionell `STATE` abgefragt wird, und auf dem Host compiliert und ausgeführt wird.

In Trutes Recognizer haben wir den Recognizer und den passenden Glue–Code isoliert, ohne das Compilieren und Ausführen.

```
: rec-num ( addr len -- n rectype-num |
                        d rectype-dnum |
                        rectype-null )
  snumber? dup IF
      0> IF  rectype-dnum
      ELSE   rectype-num  THEN
  THEN
  drop rectype-null ;
```

Die Tokens, die dabei produziert werden, kann man auch mit den Aktionen oben befüllen.

```
' >t ' t_lit, ' lit,  rectype: t-rectype-num
:noname  d>target swap >t >t ;
:noname  d>target swap t_lit, t_lit, ;
' 2lit,  rectype: t-rectype-dnum
```

(Die Postpone–Varianten sind jetzt für das Compilieren von Macros im Host.) Jetzt müssen wir nur noch im `rec-num` die Rückgabewerte ersetzen, fertig ist der Recognizer für Numerale im Target–Compiler.

```
: rec-tnum ( addr len -- n t-rectype-num |
                         d t-rectype-dnum |
                         rectype-null )
  snumber? dup IF
      0> IF  t-rectype-dnum
      ELSE   t-rectype-num  THEN
  THEN
  drop rectype-null ;
```

Nun ja, das ist jetzt nicht ganz das Gleiche. Klaus' Target–Compiler hat ja die zwei Modes `comp?` und `dbg?`, nicht über `STATE`. Obwohl das einfach die Äquivalente im Target–Mode sind.

Generalisiert sieht alter Code, der den äußeren Interpreter ersetzt, in der Regel so aus:

```
: rec-something ( addr u -- ... )
  check-string? IF
```

```
    state @ IF  compile-something
    ELSE  interpret-something  THEN  EXIT
  THEN  failure-handling ;
```

Diese generelle Struktur kann man einfach in einen Recognizer konvertieren:

```
' interpret-something
' compile-something
' lit-something rectype: rectype-something
: rec-something ( addr u -- ... rt-something |
                            rectype-null )
  check-string?
  IF    rectype-something
  ELSE  rectype-null  THEN ;
```

Dabei wird tatsächlich die Reihenfolge von Interpreter und Compiler gegenüber dem `STATE @ IF` –Statement geändert. Das ist kein Zufall, das orientiert sich an den Zahlenbeträgen, die für diese Modes im `STATE` sind.

Was soll in einen Standard?

Erst mal: Was nicht in einen Standard soll, sind Tutorials. Das gehört eher hierher.

Die Recognizer–API soll eine Schnittstelle sein, mit der man den Interpreter umkonfigurieren kann. Nicht nur komplett austauschen, sondern erweitern.

Die verschiedenen Vorschläge für alternative, „einfachere“ Recognizer, die in letzter Zeit aufgetaucht sind, bringen meist eine spezifische Implementierung mit. Klaus nutzt den Return–Stack, um die Recognizer–Sequenz abzubrechen. Diese Sequenz ist bei ihm ausführbarer Code. Er trennt Erkennen des Tokens nicht vom Ausführen und Interpretieren, und macht den Dispatch mit `STATE`. Es gibt auch andere Vorschläge, bei denen zwar die Trennung von Erkennen und Ausführen als gute Idee akzeptiert wird, aber der Dispatch dann ebenfalls über `STATE` läuft.

Die Idee von Standards ist aber, dass man von solchen Implementierungs–Details wegabstrahiert. Da hat auch der Trute–Vorschlag noch eine Schwachstelle, denn dessen Tokens sind ja erklärtermaßen Tabellen, in die man halt hineingreift, und dann das xt ausführt, das dort steht. Da wäre auch besser, das sauber zu abstrahieren.

Ein Problem an der Stelle ist `POSTPONE`, das bei Trute eben über solche Details implementiert wird. Was genau `POSTPONE` macht, ist aber gerade in Target–Compilern nicht so einfach. Eigentlich gibt es dort zwei verschiedene `POSTPONE`s: eines, das für Makros im Host gebraucht wird, und eines für Makros im Target. Das ist mit dem vor allem für Literals stark vereinfachten `POSTPONE` aus dem Trute–Vorschlag nicht machbar. Dieser Vorschlag ist aber grundsätzlich gut, weil das mit den Literals häufig vorkommt. Das ist dann aber eine Abkürzung, und sinnvoll wäre es, wenn auch die explizite Langform zugänglich bleibt.

Was fehlt noch?

Macht Trutes Recognizer zu viel oder zu wenig? Ist er für solche Anwendungen over– oder underengineered? Könnte man es anders besser machen (für jeweils einen speziellen Fall optimiert)?

Wahrscheinlich kann man zu jeder dieser Fragen irgendwo „Ja“ antworten.

Beim Target–Compiler kann man noch einfach sagen, dass das auch Interpretieren und Compilieren ist, dass man also nur andere Tokens braucht, und sich auch nicht so einen abbrechen muss wie Klaus, der das parallele Konstrukt mit `comp?` und `dbg?` da mit rübergerettet hat. Ich habe aber auch andere Parser–Codes, für eine Art Config–Datei sowie für JSON und XML, und da findet man ein sehr typisches Pattern: Ein `CASE`–Statement, das Tokens nimmt und Aktionen ausführt. Nur ist ja genau die Idee von diesen Tokens, dass sie die Aktionen selbst mit sich bringen, aber halt nur für Interpret/Compile/-Postpone. Diese Design–Pattern werden vom aktuellen Recognizer nicht gut unterstützt.

Ich habe auch Code, der als Rückgabe immer ein xt plus `rectype-nt` liefert, weil es hier nur einen einzigen Zustand gibt (alles nur Interpreter). Auch das sieht so aus, als ob hier die vorhandene API etwas missbraucht wird. Es gibt auch Leute, die ihren vorhandenen Code (sieht auch so aus wie bei Klaus) einfach in den Recognizer packen, und dann `[’] noop rectype-nt` zurückliefern. Das ist natürlich auch geschummelt, aber damit kann man seinen Poor–Man’s–Recognizer–Code einfach hinschreiben.

An der Stelle wünsche ich mir dann ein Duck–Type–OOP, also eine Möglichkeit, neue Selektoren für existierende Tokens zu definieren, und dann den Tokens auch zu diesen Selektoren passende Aktionen zuzuweisen. Für eine Implementierung wie in Gforth ist das sicher akzeptabel, das so weiterzuentwickeln, für den Standard ist das erst mal zu viel. Da der Standard aber implementierungsneutral sein sollte, müsste eine Erweiterung in diese Richtung problemlos möglich sein.

An der Stelle ist der Trute–Vorschlag zur Zeit wohl noch nicht implementierungsneutral genug, denn die Tokens müssen zwingend Tabellen sein. Da wäre wohl sinnvoll, das Ausführen nicht zu verstecken, sondern die Wörter, die auf die Recognizer zugreifen, direkt in dem jeweiligen Mode (oder im generischen Fall, dem von `STATE` selektierten Mode) auszuführen.

Da es sich hierbei um Erweiterungen handelt, die sich auf alle Fälle erst mal außerhalb des Standards befinden, kann man hier natürlich relativ frei walten.

Referenzen

[1] Klaus Schlesiek, *Poor Man’s Recognizer,* VD 3/2020

[2] Matthias Trute, *Forth Recognizer — Request for Discussion,* `https://forth-standard.org/proposals/recognizer`

Von Groß– und Kleinbuchstaben

Anton Ertl

Michael Kalus wunderte sich in einem Leserbrief in VD 3/2020, S. 5 darüber, dass `blau` *und* `BLAU` *in Gforth zwar das gleiche Wort sind,* `grün` *und* `GRÜN` *aber nicht (übrigens auch in anderen populären Forth–Systemen). Hier die Erklärung, und wer sie bis zum Ende durchliest, findet wahrscheinlich auch, dass die von Gforth gewählte die beste Lösung ist.*

Ursprung

Seit der Spätantike (vielleicht schon früher) gibt es Groß– und Kleinbuchstaben. Welche man verwendet, ändert genauso wenig an der Bedeutung eines Wortes wie, ob es in Times oder in Helvetica gesetzt ist. Deswegen gibt es im Morse–Code auch nur Großbuchstaben. Auch spätere Codierungen, z. B. der 5–Bit–Baudot–Code (Fernschreiber) und 6–Bit–Codierungen (36–Bit–Maschinen) unterstützen nur Großbuchstaben.

1963 kamen mit ASCII endlich Kleinbuchstaben auf die Computer (obwohl auch noch der C128 (1985) nach dem Einschalten nur Großbuchstaben und Symbole zeigt). Damit stellte sich die Frage, ob z. B. X und x das Gleiche ist oder nicht. Pascal und das DOS–Filesystem beantworteten diese Frage mit „ja“ (case–insensitiv), C und Unix mit „nein“ (case–sensitiv).

Forth

In Forth gab es (natürlich) beide Antworten auf diese Frage. fig–Forth (und wahrscheinlich alle Systeme für Computer mit sehr wenig Speicher) antworteten „nein“, denn schließlich wollte man knappen Speicher nicht für die Antwort „ja“ verschwenden, auch wenn das nur ein paar Bytes kostet (wohl vermutlich ähnlich in C und Unix).

Die genaue Geschichte der Antwort „ja“ in Forth konnte ich für diesen Artikel nicht recherchieren, jedenfalls erlaubt Forth–94 (und Forth–2012) Forth–Systemen beide Antworten; mit anderen Worten, es wurde keine Antwort auf diese Frage standardisiert.

Und die Antwort ist „ja“ in Gforth, iForth, SwiftForth und VFX. Eine große Rolle spielt dabei wohl, dass die Forth–Standards alle Wörter in Großbuchstaben definieren, Programmierer aber NICHT DAUERND SCHREIEN WOLLEN, bzw. solches Geschrei nicht dauernd lesen wollen. Eine weitere Erklärung kam (wenn ich mich recht erinnere) von Elizabeth Rather: Bei (fern–)mündlicher Kundenbetreuung ist es einfacher, wenn das System keinen Unterschied zwischen Groß– und Kleinbuchstaben macht.

Dass der Standard den Systemen beides erlaubt, heißt aber auch, dass Standard–Programme für beides geschrieben sein müssen. Programmierer müssen also einerseits alle Wörter genauso schreiben, wie sie definiert sind (also Standard–Wörter in Großbuchstaben), andererseits dürfen sie sich auch nicht darauf verlassen, dass X und x zwei verschiedene Wörter sind. Allerdings halten sich viele Programmierer nicht daran, sondern schreiben Programme für case–insensitive Systeme.

Jenseits von ASCII

Um weitere Sprachen zu unterstützen, wurde zunächst das 8. Bit genutzt und die ISO–8859–Zeichensätze eingeführt (z. B. ISO–8859–1 (Latin–1) für westeuropäische Sprachen und ISO–8859–2 (Latin–2) für lateinschrift–basierte osteuropäische Sprachen).

Dabei kann man aber Wörter verschiedener Sprachen u. U. nicht im gleichen Text verwenden, und einige asiatische Sprachen mit großen Zeichensätzen passen bei weitem nicht in das 8–Bit–Korsett. Also wurde Unicode entwickelt, mit diversen Wirrungen, über die man einen weiteren Artikel schreiben könnte.

Jedenfalls setzt sich UTF–8 als Codierung von Unicode durch. UTF–8 hat die nette Eigenschaft, dass eine klassische ASCII–Datei (oder im Speicher ein ASCII–String) automatisch eine korrekte UTF–8–Datei (bzw. ein UTF–8–String) ist.

Und so funktionieren auch Forth–Systeme, die eigentlich nur für ASCII entwickelt wurden, mit ein paar Schwächen auch für UTF–8 (auch jenseits von ASCII), wenn sie das 8. Bit nicht für andere Zwecke verwenden.

Und Case–Insensitivität?

Während es für ASCII–Zeichen ganz einfach ist, das Zeichen case–insensitiv zu vergleichen, ist das jenseits von ASCII deutlich problematischer. Zunächst einmal müsste man die Codierung wissen; ist es Latin–1, Latin–2, UTF–8 oder eine andere ASCII–kompatible Codierung (z. B. KOI8–R).

Aber selbst, wenn man UTF–8 voraussetzt (was vielleicht noch etwas voreilig ist), muss man außerdem noch die Sprache wissen, in der das Wort geschrieben ist. So ist z. B. im Deutschen der Großbuchstabe zu i das I, während es im Türkischen İ ist. Umgekehrt ist im Türkischen der Kleinbuchstabe zu I das ı.

Und selbst, wenn man die Codierung und die Sprache kennt, bleibt noch die relativ aufwendige Implementierung von Case–Insensitivity für Nicht–ASCII–Zeichen. Das ist aus praktischer Sicht ein wichtiger Grund für die Entscheidung, Nicht–ASCII–Zeichen case–sensitiv zu behandeln.

Codierung und Sprache

Aber m. E. ist das Problem, dass man Codierung und Sprache kennen müsste, entscheidend. Wie könnte das funktionieren?

In Unix gibt es die Idee von Environment–Variablen wie `LANG`, über die der Benutzer den Programmen mitteilt, welche Sprache und welche Codierung er benutzt.[1]

Löst das unser Problem, dass wir die Codierung und die Sprache nicht kennen? Nein, denn die Codierung und die Sprache eines Programms hängen nicht vom Benutzer zur Zeit des Programmaufrufs ab, sondern vom Programmierer des konkreten Codes.

Auch eine Teilung von Compilezeit und Laufzeit, wie sie z. B. in Gforth über die Erzeugung eines Images vollbracht werden kann, ist nicht die Lösung: Man kann zur Compilezeit Code von verschiedenen Programmierern kombinieren, die verschiedene Codierungen und Sprachen verwendet haben.

Folgen wir dem Kaninchen[2]

Eine mögliche Lösung wäre, am Anfang jeder Quellcodedatei die Codierung und die Sprache zu deklarieren. Wenn die Programmierer das machen, wissen wir Codierung und Sprache für jedes Stück Code, aber was bedeutet das?

Die Codestücke sind ja keine abgeschlossenen Welten, sondern enthalten Namen von Wörtern aus anderen Dateien. Soll man in einem türkischen Stück Code das Standardwort `IF` jetzt `if` oder `ıf` schreiben, wenn man es klein schreiben will? Und wenn ein türkisches Wort mit einem `İ` vorkommt, wie schreibt man die kleingeschriebene Variante in einem deutschen Programmteil?

Wenn man tatsächlich in diese Richtung gehen will, wäre es meines Erachtens das sinnvollste, wenn mit jedem Namen die Codierung und Sprache mitgespeichert wird, in der der Name definiert wurde, und der Name wird dann entsprechend den Regeln dieser Codierung und Sprache verglichen. Auch in einem türkischen Programmteil wäre die Kleinschreibung des Standardwortes `IF`, das aus dem Englischen stammt, `if`, nicht `ıf`. Umgekehrt, wenn jemand in einem türkischen Programmteil `DİYARBAKIR` definiert hat, könnte man dieses Wort überall, auch in einem deutschen Programmteil, als `Diyarbakır`[3] aufrufen, aber nicht als `Diyarbakir`.

Auch mit dieser Variante gibt es noch potentielle Überraschungen: Wenn jemand in einem türkischen Programmteil ein Wort `ıf` definiert, würde ein späteres `IF` in einem deutschen Programmteil sich auf dieses Wort beziehen, nicht auf das Standard–Wort `IF`, dagegen würde sich `if` auf das Standard–Wort beziehen. Im Folgenden gehe ich davon aus, dass wir das in Kauf nehmen.

Implementierung des Kaninchenbaus

Auf der Implementierungsseite wird die Codierung wohl am besten beim Einlesen der Datei in UTF–8 umgewandelt, dadurch sind alle Namen im Forth–System in UTF–8, und man muss sich die Codierung nicht beim Namen speichern, und Ausgabe und Vergleich von Namen werden deutlich einfacher.

Bezüglich der Sprache wird bei jedem Wort ein Index für die Sprache abgespeichert. Da vermutlich nicht mehr als 256 Sprachen gleichzeitig im System vorkommen, reicht dafür ein Byte. Beim Vergleich der Namen verwendet man dann den case–insensitiven Vergleich der im Wort–Kopf abgespeicherten Sprache.

Größere Forth–Systeme verwenden Hashing, um das Suchen von Namen zu beschleunigen. Bei case–insensitiven Forth–Systemen muss jede mögliche Schreibweise eines Namens den gleichen Hashwert ergeben, sonst wird der Name nicht sicher gefunden.

Bei unserer sprachabhängigen Case–Insensitivität haben wir das Problem, dass wir beim Berechnen des Hashwerts des Wortes, nach dem wir suchen, nicht wissen, nach den Regeln welcher Sprache wir das Wort vergleichen werden; es können sogar mehrere Sprachen sein. Daher müssen wir einen Hashwert bilden, der für die möglichen Schreibweisen eines Namens in allen möglichen Sprachen immer denselben Hashwert bildet. Das kann z. B. erreicht werden, indem vor der Berechnung des Hashwerts alle vier Schreibweisen I, i, İ, und ı (und vielleicht noch mehr) auf eine (z. B. I) abgebildet werden.

Zahlt es sich aus?

Ein vollkommen case–insensitives System ist also möglich, kostet aber einiges an Implementierungsaufwand (vermutlich viele Personenwochen), Speicherplatz (auf größeren Systemen leicht leistbar), Compilationsgeschwindigkeit und auch Deklarationsaufwand auf der Programmiererseite. Wollen wir uns das wirklich leisten? Ist es uns wirklich so wichtig, dass `grün` als `GRÜN` erkannt wird?

Und was passiert, wenn jemand die Codierung und Sprache nicht deklariert? Eine naheliegende Möglichkeit in diesem Fall ist, nur ASCII–Zeichen case–insensitiv zu behandeln, und alles andere case–sensitiv, denn dann ist sowohl die Codierung[4] als auch die Sprache egal.

In voller Absicht implementieren wir das in Gforth, ohne die Möglichkeit anzubieten, Codierung und Sprache zu deklarieren. Auch andere case–insensitive Forth–Systeme

[1] Es gibt weitere Variablen, mit denen man diverse Feinheiten genauer einstellen kann; man kann sie mit dem Kommando `locale` ausgeben.

[2] Am Anfang von *Alice im Wunderland* folgt Alice einem Kaninchen in seinen Bau.

[3] So wird dieser Name auch in der deutschen Wikipedia geschrieben.

[4] Wenn verschiedene Codierungen gemischt werden, kann das zu einem Problem werden, weil `grün` in Latin–1 dann nicht mit `grün` in UTF–8 zusammenpasst, aber das kann der Programmierer, der Dateien in verschiedenen Codierungen vereint, lösen, indem er alle Programmdateien in UTF–8 konvertiert. Und auf die Dauer wird das Problem immer geringer, weil sich UTF–8 immer mehr durchsetzt.

implementieren das, auch wenn ich den Eindruck habe, dass das nicht aus ausgefeilten Überlegungen erfolgt, sondern weil es sich automatisch ergibt, wenn ein Forth–System für ASCII mit ASCII–kompatiblen Codierungen in Kontakt kommt. Warum implementiert Gforth Case–Insensititität nur für ASCII–Zeichen?

Der Hauptvorteil von Case–Insensitivität ist, dass wir Standard–Wörter wie `DUP` klein schreiben können, der ist auch mit der ASCII–Beschränkung erfüllt. Wie oben angeführt, würde eine Erweiterung auf Nicht–ASCII–Zeichen einen beträchtlichen Implementierungs–Aufwand erfordern. Weiters müssten die Programmierer in ihren Programmdateien auch noch die Sprache deklarieren; wieviele würden das tatsächlich tun, oder würden sie nicht lieber die Case–Insensitivität von `grün` in Kauf nehmen?

Bezüglich der Codierung wäre der Aufwand geringer, aber auch da stellt sich die Frage: Wer würde ein existierendes z. B. Latin–1–Programm um die Deklaration einer Codierung erweitern? Mit demselben oder geringerem Aufwand könnte das Programm einfach in UTF–8 umgewandelt werden. UTF–8 sollte m. E. die Default–Codierung sein, sodass man sich damit die Deklaration sparen kann.

Diese Gründe finde ich so überzeugend, dass ich diese Vorgangsweise zur Standardisierung vorgeschlagen habe.[5]

Chuck Moore propagiert das Eliminieren von Anforderungen als eine wichtige Möglichkeit, um Komplexität zu eliminieren. Im Allgemeinen sehe ich diese Strategie eher skeptisch, da die Benutzer oft ein komplexeres Programm, das ihre Anforderungen erfüllt, einem einfacheren, das sie nicht erfüllt, vorziehen. In diesem Fall aber scheint mir ein Fall vorzuliegen, in dem man Chuck Moore's Rat folgen sollte.

Fortsetzung von Seite 6

Der Baudot–Code

Jean–Maurice–Émile Baudot, genannt Émile, (* 11. September 1845 in Magneux, Département Haute–Marne; † 28. März 1903 in Sceaux bei Paris) war ein französischer Ingenieur und Erfinder.

Geboren als Sohn eines Bauern, besuchte Baudot nur die Grundschule. In seiner Jugend arbeitete er auf dem Bauernhof seines Vaters. Bis zu seinem Eintritt in die französische Verwaltung für Post und Telegrafie am 16. Juli 1870 lebte er somit ein ländliches Leben, das ihn keineswegs zu seinen späteren Erfindungen prädestinierte.

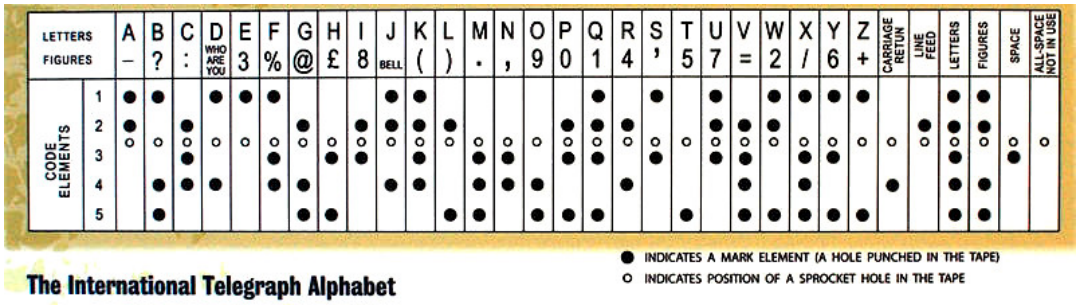
Der ursprüngliche Baudot–Code (später International Telegraph Alphabet No. 1 (ITA1), CCITT–1) wurde von Émile 1870 für ein von ihm entwickeltes Telegrafengerät entworfen. Der Code wurde direkt über eine klavierähnliche Tastatur mit fünf Tasten eingegeben. Dazu wurde die Tastatur mit dem Zeige– und Mittelfinger der linken und mit dem Zeige–, Mittel– und Ringfinger der rechten Hand bedient. Die dem zu sendenden Zeichen entsprechenden Tasten mussten gleichzeitig gedrückt werden und rasteten für einen Moment ein, bis die Kombination vom Gerät als eine Folge von Stromimpulsen gesendet und die Tastatur für das nächste Zeichen wieder freigegeben wurde. Auf diese Weise wurden Geschwindigkeiten von 180 Zeichen pro Minute erzielt.

Da es mit fünf Tasten nicht genug verschiedene Tastenkombinationen gibt, hätten nicht einmal alle 26 Buchstaben plus 10 Ziffern codiert werden können, wenn Baudot nicht einen Umschaltcode eingeführt hätte, der die doppelte Belegung fast aller Kombinationen erlaubte: Er definierte zwei Leerzeichen. Wenn das eine gesendet wurde, sollten die nachfolgenden Zeichen nach einer Tabelle mit Buchstaben interpretiert werden, nach dem anderen sollte eine Tabelle mit Ziffern und Zeichen benutzt werden.

Für seine Leistungen wurde Émile Baudot 1879 das Kreuz der französischen Ehrenlegion verliehen. Die höchste zu Lebzeiten erhaltene Auszeichnung seines Schaffens erhielt Baudot, als er 1898 zum Offizier der französischen Ehrenlegion ernannt wurde. Am 28. März 1903 starb Émile Baudot nach langer Krankheit im Alter von 57 Jahren.

Im Jahre 1926 wurde ihm zu Ehren die Einheit für die telegrafische Schrittgeschwindigkeit eingeführt und *Baud* genannt. Die Maßeinheit Baud bezeichnet die Anzahl der übertragenen Symbole pro Sekunde. Der Asteroid (14400) Baudot wurde im Jahre 2000 nach ihm benannt. mk

(Quelle: `http://informatik.rostfrank.de/rt/lex06/baudot_code.html`



The International Telegraph Alphabet

[5] `https://forth-standard.org/proposals/case-insensitivity`

Einführung in das VIS–System II

Martin Bitter

Noch ist es nicht endgültig, aber es gibt Bestrebungen, VIS in das Feuersteinprojekt aufzunehmen. VIS wurde entworfen und realisiert von Manfred Mahlow. *Das VIS–System ist eine Abwandlung und Erweiterung des VOCABULARY–Konzepts, wie man es von klassischen Forth–Systemen kennt. VIS steht für VOC[1], ITEM und STICKY. Die beiden letzteren Begriffe sind Neuschöpfungen.*

Wie in Heft 03/2020, S. 31, versprochen, nun etwas über den Gebrauch von `item` *und verschachtelte VOCs. Ein ernsthaftes Beispiel geriet mir zu lang und abschweifend, deshalb hier das Nonsens–Paket* `politics`*. Es ist kurz, zeigt aber die Dinge, auf die es mir jetzt ankommt. Ich entschuldige mich bei allen hart und ehrlich arbeitenden PolitikerInnen und PolitikerAußen.*

Das `politics`–VOC

In Zeile 1 definiere ich ein `VOC` mit Namen `politics`. Die Phrase `politics DEFINTIONS` legt fest, dass ab jetzt alle neuen Worte in diesem VOC landen. Und `politics also` erreicht, dass der Interpreter/Compiler zuerst im `politics`–VOC nachschaut, danach in den anderen VOCs, die in der Suchreihenfolge sind (vgl. `order`). [Im Folgenden verwende ich hin und wieder auch die Abkürzung *[Zn]* als Hinweis auf die *Zeile n* des Listings.]

Ab der Zeile 5 geschieht Kriminelles: Ziffern werden als Worte definiert. Das erste Wort `1` wird im `politics`–VOC erzeugt. Bei der Definition von `2` schaut der Compiler zuerst im `politics`–VOC nach — dafür habe ich mit dem `also` gesorgt — findet dort das Wort `1` und führt es aus.[2] Das Gleiche gilt für die Definitionen von `3`, `4` und `5`.

Es lohnt sich eine Überlegung zur Definition der `5`. Darin wird tatsächlich `die Zahl „5“` aus dem `forth`–VOC in die `DO ... LOOP` eingebaut. (Was würde bei einer Überdefinition geschehen?)

Sodann wird ein *definierendes* Wort `phrase:` definiert. Beim Kompilieren legt `<builds` den Wortkopf an, `0 parse` liest aus dem Eingabestrom die restliche Zeile ein und `string,` legt sie als *Counted–String* im Speicher ab. Wird später das neu erzeugte Wort ausgeführt, so sorgt `does>` dafür, dass die Datenadresse des Wortes auf den Stack kommt. Hier ist das die Adresse des Counted–Strings, die `count` für weitere Aufgaben aufbereitet. Das Wort `phrase` *[Z12]* ist hier aus didaktischen Gründen gezeigt, im Ernstfall fiele es weg.

```
1   VOC politics
2   politics DEFINiTIONS
3   politics also
4
5   : 1 ( caddr u -- )  type space ;
6   : 2 ( caddr u -- )  2dup 1 1 ;
7   : 3 ( caddr u -- )  2dup 2 1 ;
8   : 4 ( caddr u -- )  2dup 3 1   ;
9   : 5 ( -- )
10    5 0 DO 2dup politics 1 LOOP  2drop ;
11
12  : phrase: ( "name" "string" -- )
13    <builds 0 parse string, does>
14    count
15  ;
16
17  : iphrase: ( "name" "string" -- )
18    item <builds 0 parse string, does>
19    count
20  ;
21
22  sticky : äh ." Äh! " ;
23
24  VOC whatdoes
25  whatdoes DEFINITIONS
26  whatdoes also
27
28  : announce ( caddr u n -- )
29    -rot
30    ." Prints string '" type ." ' "
31    . ." times!"
32  ;
33
34  : 1? ( caddr u -- ) forth 1 announce ;
35  : 2? ( caddr u -- ) forth 2 announce ;
36  : 3? ( caddr u -- ) forth 3 announce ;
37  : 4? ( caddr u -- ) forth 4 announce ;
38  : 5? ( caddr u -- ) forth 5 announce ;
39
40  only
41  forth DEFINITIONS
42
43  politics phrase: Hugo Glückauf!
44  item politics phrase: Strauss Sympathisant!
45  politics iphrase: Kohl Ich schreibe Gechichte!
46
47  : phrase: ( -- ) politics phrase: ;
48  : iphrase: ( -- ) politics iphrase: ;
49
50  iphrase: Wehner Sie Pinscher!
51  iphrase: Blühm Die Rente ist sicher!
52  iphrase: Merkel Der Lockdown ist alternativlos!
53  iphrase: Münchner A Maß wüll i habn!
54  iphrase: Engel Hallelujah!
```

Nun folgt das ernst gemeinte `iphrase:` — mit einem Unterschied: es kommt das Wort `item` *[Z18]* hinzu! `item` „merkt“, dass es sich im `politcs`–VOC befindet und sorgt dafür, dass das nächste Wort, das mit seiner Hilfe definiert wird, beim Aufruf eben dieses `politics`–VOC einschaltet. Hört sich unspektulär an, bietet aber beim weiteren Arbeiten große Erleichterungen (s. Beispiel „Parlamentsdebatte“).

Ähnlich ist es mit dem Wort `äh`, das mit dem Präfixwort `sticky` *[Z22]* definiert wird.

[1] *VOC ist eine Ersetzung für VOCABULARY.*

[2] Statt der Zahl „1“. Das normale `number` aus dem `forth`–VOC kommt nicht zum Tragen.

Noch einmal zur Klarheit: Manfred Mahlow nennt die beiden Worte `item` und `sticky` „prefix words“, weil sie beide Einfluss darauf haben, wie das nächste Wort, das definiert wird, sich verhält. Auch dazu mehr weiter unten in der „Parlamentsdebatte“. Doch ihr ahnt vermutlich schon, was es mit den beiden auf sich hat.

Wenn ich schon mal dabei bin, Sinnloses zu tun, kann ich das auch steigern.

Noch bin ich ja im `politics`–VOC. Wenn ich jetzt ein neues VOC erzeuge, `VOC whatdoes` *[Z24]*, ist das ein ganz normales Wort im `politics`–VOC. Das bedeutet aber auch, es ist vom `forth`–VOC aus unsichtbar. In den beiden folgenden Zeilen sorge ich dann dafür, dass alle weiteren Definitionen zum einen im `whatdoes`–VOC landen und zum anderen auch dort gesucht werden.

Mit `announce` definiere ich in diesem `whatdoes`–VOC nun ein Wort, das eine übergebene Zeichenkette und eine Zahl in eine Auskunftsmeldung einbaut. `announce` wird dazu benutzt, „Frageworte“ zu definieren, die mit den Zahlworten `1`, `2`, `3`, `4` und `5` im `politics`–VOC korrespondieren.

Für heute bin ich fertig. Mein Politikpaket kann verschnürt werden. Eine Kleinigkeit fehlt noch, doch auch da vertröste ich auf die folgende Parlamentsdebatte. Mit `only` (Zeile 40) und `forth DEFINITIONS` *[Z40, 41]* räume ich noch die Suchordnung auf.

Parlamentsdebatte

Die Protagonisten (Anwendungsbeispiel)

Die Protagonisten mit ihren Standardphrasen werden in *[Z43 bis 54]* eingeführt. Dabei stört es mich, dass vor jedem `phrase:` bzw. `iphrase:` ein `politics` eingegeben werden müsste. Ich definiere deshalb diese Worte aus dem `politics`–VOC hier noch einmal ins aktuelle `forth`–VOC, um mir die Arbeit zu erleichtern (Ihr durchschaut diesen Trick nun schon!).

Ich möchte auf die Unterscheide in *[Z43, 44, 45]* aufmerksam machen. Nur scheinbar passiert in den drei Zeilen das Gleiche: Es wird durch `phrase:` bzw. `iphrase:` ein Wort im `politics`–VOC erzeugt — `Hugo`, `Strauss`, `Kohl` — das beim Aufruf eine Zeichenkette auf den Stack legt — `Glückauf`, `Sympathisant`, `Ich schreibe Gechichte`. Die Unterschiede sind: Im ersten Fall taucht `item` gar nicht auf, dann explizit und schließlich implizit innerhalb von `iphrase:`.

Die Konsequenzen zeigen sich in der weiteren Verwendung der erzeugten Worte:

```
hugo <ret>
```

legt zwei Zahlen `( -- caddr u )` auf den Stack. Das ist ok.

```
hugo 1 <ret>
```

legt jetzt drei Zahlen auf den Stack `( -- caddr u 1 )`. Aber ich will, dass das *Wort* `1` aus dem `politics`–VOC genommen wird, nicht die Zahl „1“. Erst die Phrase:

```
hugo politics 1 <ret>
```

macht endlich das, was ich will. Es erscheint die Ausgabe: „Tachchen!“, weil ich nach `hugo` explizit das `politics`–VOC eingeschaltet habe.

Dieses explizite Einschalten des VOCs kann das mit `phrase:` definierte Wort selbst erledigen, wenn ich vor seiner Definition das Präfixwort `item` aufrufe. Genau das habe ich im zweiten Fall gemacht. Folge:

```
Strauss 1 <ret>
```

gibt sofort „Sympathisant!“ aus, weil das Wort `Strauss` das `politics`–VOC aktiv hinterlässt, sodass der Interpreter/Kompiler nun dort nach dem Wort `1` sucht, es findet und ausführt. Und erst *danach* sind wir wieder im `forth`–VOC.

Im dritten Fall verwende ich `iphrase: ,` wodurch ja schon bei der Definition von `Kohl` das Wort `item` verwendet worden ist. Der Effekt ist dann der Gleiche:

```
Kohl 1 <ret>
```

ergibt die Phrase „Ich schreibe Gechichte!“

Jetzt können wir mal Politiker spielen!

Ach so! Wie komm’ ich auf die Namen der Politiker? Sie sind mir im Gedächtnis geblieben und in den Geist gekommen ;-) Der Münchner gehört eigentlich frei nach Thoma in den Himmel. Ihn habe ich als Gegengewicht zu FJS eingefügt. Und der Hugo? Der leitet sich von Kohl ab, wegen dieser Anekdote: Ich glaube es war in den 1990ern, da schrieb der Personalrat der „Zeche Hugo“, hochdeutsch „Bergwerk Hugo“ in Gelsenkirchen–Buer einen Brief an den damaligen Kanzler Helmut Kohl. Das Antwortschreiben begann mit den Worten: „Sehr geehrter Herr Bergwerk!“ und war adressiert an Herrn Hugo Bergwerk, Gelsenkirchen.

Das Spiel

Zuerst eine Übersicht[3]:

```
politics words <ret>
---------------
<< FLASH: politics
>> RAM: politics
wtag: ..36 lfa: ..20 xt: ..32 name: whatdoes
ctag: ..1
wtag: ..35 lfa: ..F4 xt: ..00 name: äh
wtag: ..34 lfa: ..94 xt: ..A6 name: iphrase:
wtag: ..34 lfa: ..44 xt: ..54 name: phrase:
wtag: ..34 lfa: ..F4 xt: ..FE name: 5
wtag: ..34 lfa: ..D0 xt: ..DA name: 4
wtag: ..34 lfa: ..AC xt: ..B6 name: 3
wtag: ..34 lfa: ..88 xt: ..92 name: 2
wtag: ..34 lfa: ..60 xt: ..6A name: 1
```

Falls ich vergessen habe, welche Worte `whatdoes` enthält:

[3] Die Liste ist sehr breit. Damit sie in die Spalte passt, hier eine verkürzte Darstellung.

```
politics whatdoes words <ret>
---------------
<< FLASH: politics whatdoes
>> RAM: politics whatdoes
wtag: ..20 lfa: ..48 xt: ..54 name: 5?
wtag: ..20 lfa: ..24 xt: ..30 name: 4?
wtag: ..20 lfa: ..00 xt: ..0C name: 3?
wtag: ..20 lfa: ..DC xt: ..E8 name: 2?
wtag: ..20 lfa: ..B8 xt: ..C4 name: 1?
wtag: ..20 lfa: ..4C xt: ..5E name: announce
```

Was machen `Merkel`, `Hugo`, `Kohl`, ... mit `1?` ?

```
Merkel whatdoes 1? <ret>
```

gibt den String „Der Lockdown ist alternativlos!“ einmal aus!

Let's Play! (Ein Mitschnitt)

```
Hugo politics 1 Glückauf!
Blühm 2
Die Rente ist sicher!
Die Rente ist sicher!

Wehner 3
Sie Pinscher!
Sie Pinscher!
Sie Pinscher!

Münchner 2
A Maß wüll i habn!
A Maß wüll i habn!
```

Noch eine Besonderheit: Das Wort `äh` wurde ja mittels Präfix `sticky` definiert. Das ermöglicht nun die Sequenz `Engel äh 4` . Normalerweise würde nach `Engel` *nur für die Ausführung des nächsten Wortes* in das `politics`–VOC geschaltet und die `4` würde vom `forth`–VOC ausgeführt. Doch das „klebrige“ Wort `äh` verhindert das nun. Man bleibt praktisch am `politics`–VOC kleben und reicht das durch an das Wort `4` :

```
Engel äh 4
Äh!
Hallelujah! Hallelujah!
Hallelujah! Hallelujah!
```

Jetzt weiß ich auch, wieso `words`, `order` und der Dictionarybrowser `??`[4] funktionieren, ohne den VOC–Mechanismus zu stören.

Tatsächlich machte mir das Spielen mit den Worten aus dem `politics`–VOC mehr Spaß, als ich zu Anfang dachte. Wer weiß? Vielleicht bekommt ja der eine oder andere auch Lust, Ähnliches zu spielen? Vielleicht mit anderen sozialen Rollen? Lehrern? Etwa so:

```
iphrase: Mutmacher: Halb so schlimm!
iphrase: Selbstmitleid: Wieso immer ich?
```

Oder doch lieber Programmiererpersönlichkeiten? Ich warte auf Einsendungen ;-)

Finis!

Wir sehen uns vielleicht wieder im nächsten Heft bei einer dritten Folge: Über die Fallstricke und warum ich unbedingt eine leere Menge in einem VOC brauche.

Nützliches

```
https://wiki.forth-ev.de/lib/exe/fetch.php/
events:ft2019:vocs_items_und_sticky_words.pdf
```

Quellenangabe

VIS für 430eForth und Mecrisp ist zu bekommen unter:

```
https://forth-ev.de/wiki/projects:forth-
namespaces:start
```

Visa–Informations–System

Dieses VIS gibt es schon ein Weilchen, hat aber mit Forth nichts zu tun. Steht hier nur, damit ihr euch nicht wundert, wenn ihr nach VIS googelt.

> „While ideas and proposals on how the Schengen agreement should be updated and that the Schengen area needs rearrangement have been constantly floating in the air, the European Union keeps working to enhance security in the EU countries, and the non–EU Schengen members.
>
> From Salvini pushing for Italy to leave, to the former Belgian PM suggesting expelling of the Visegrad countries and Macron's proposal for rearrangement of the area, all kind of ideas have been presented by the highest officials of the Member States.
>
> However, aside from the United Kingdom still in the process of leaving even over three years after its referendum to exit the EU, no other serious moves to withdraw or expel any country have been in sight in the last years. Contrarily, the European Union has been continuously working towards creating new means to ensure the block continues to work efficiently.“

```
https://edps.
europa.eu/data-
protection/european-
it-systems/visa-
information-system_de
```

[4] Ja, tatsächlich sind es zwei Fragezeichen, die das Wort bilden. Siehe Heft 03/2020 S.31

Conditional Compiling

Michael Kalus

Forth zeichnet sich durch seinen simplen Quellcode–Parser[1] aus. Das ist mit ein Grund dafür, dass es so kompakt ist auf MCUs. Und der Parser kann so einfach sein, weil in Forth sehr wenig syntaktische Regeln vorgegeben werden. Die Hauptregel ist: Jede Funktion holt sich ihre Parameter vom Stack[2] und hinterlässt das Ergebnis auch wieder dort. Keine Klammern, keine Kommata als Separatoren von Ausdrücken, eingerückte Zeilen haben keine Bedeutung — nichts dergleichen. Einfach nur Worte, getrennt von „white space“, und dahinter der Stack. Doch es gibt Ausnahmen davon. Ähnlich wie in anderen Sprachen will man in Forth auch Kommentare haben. Also Zeichenfolgen im Quellcode, die dem menschlichen Leser etwas bedeuten, den Code–Compiler jedoch nicht interessieren sollen.

Kommentare

Alles, was im Kommentar steht, soll der Compiler übergehen. Das ist, wenn man so will, schon eine Form von bedingter Compilierung. Der irgendwie als Kommentar gekennzeichnete Textbereich wird weder compiliert noch von der Maschine interpretiert. Oder? Auch davon gibt es Ausnahmen. Manche Autoren benutzen die Kommentare gleich als Dokumentation für die Funktionen. Daraus werden die Handbücher gemacht, die Forth–Enzyklopädien oder Quick–Reference–Cards, mit Hyperlinks zum Quellcode gleich in der Forthdefinition.

Doch zurück zum Einfachen. In einem Assembler wird für gewöhnlich nichts von dem assembliert, was nach einem *Semikolon* bis zum Zeilenende steht. Das ist Kommentar. Wie wir wissen, ist das Semikolon in Forth anders besetzt. Es schließt eine Definition ab und kommt zusammen mit dem Doppelpunkt daher, der bekanntlich eine neue Definition einleitet. Das `:` und das `;` bilden eine Art Klammer um die zu schaffende Funktion. Wobei das *erste Wort* nach dem Doppelpunkt der *Name* der neuen Funktion wird.

```
: test ( -- ) 1 2 3 . . . ;
```

Und da geht es schon los. Wir sahen soeben eine beliebte Konvention, in der so ein Stückchen Quellcode aufgeschrieben werden könnte. Eine einfache Demo–Definition, nicht wahr? Aber das Schriftbild zeigt *keine* festgelegte Syntax, nur ein gern benutztes „pretty printing“. Zum Beispiel sind die folgenden Schreibweisen absolut gleichwertig und ergeben dasselbe Kompilat.

```
: test
( -- )
1 2 3
. . . ;

: test
1
 2
  3 ( n1 n2 n3 -- ) . ( n1 n2 -- )
. . \ now stack is empty again
;
```

Es gibt bestimmt noch viele andere Formen, die euch einfallen könnten. Einzige syntaktische Bedingung für die Funktion ist, dass ihr Name nach dem Doppelpunkt zu kommen hat, dem dann der Forth–Körper folgen muss, dessen Ende vom Semikolon angezeigt wird. Was danach noch kommen sollte, gehört dann nicht mehr dazu. Und irgendwo dazwischengestreute Kommentare dürfen sein, müssen es aber nicht.

Bedingtes Compilieren

Eigentlich sind Kommentare schon eine Art bedingte Kompilierung, denn es macht sich ja kein irgendwie gearteter Syntaxprüfer darüber her, sondern es handelt sich um einen glatten Durchfluss von Forth–Instruktionen in der gesamten Definition. Der Doppelpunkt ist ein Forthwort und das Semikolon auch. Und alles dazwischen ebenso. Sogar die öffnende Klammer ist ein Forthwort und kein syntaktisches Zeichen! Forth trifft einfach auf dieses Wort `(` und macht damit, was es mit allen Forthworten macht: Es wird ausgeführt, `execute` .

Das Besondere an `(` ist nun, dass es den Kompiliervorgang anhält, sich selbst auf die Suche nach einer schließenden Klammer macht und alles, was bis dorthin an Zeichen steht, geflissentlich ignoriert. Man könnte fast sagen, `(` ist eine „Condition“.

```
( IF skip_to_)_or_end_of_line
  THEN proceed_compiling
```

Das Kommentarzeichen \ (backslash) funktioniert entsprechend; es ist auch ein Forthwort.

```
\ IF skip_to_end_of_line
  THEN proceed_compiling
```

Anders als bei der Klammer wird einfach immer der ganze Rest der Zeile verworfen.

So kann man sich also in Forth alle möglichen Schalter–Worte schaffen, die ausgeführt werden sollen, sobald

[1] Ein Parser ist ein Computerprogramm, das für die Zerlegung und Umwandlung einer Eingabe in ein für die Weiterverarbeitung geeigneteres Format zuständig ist.

[2] Datastack — last in, first out memory area.

[3] Die Schreibweise, das Wort in Klammern zu setzen, ist pure Konvention für den menschlichen Leser und macht im Compiler nichts. Wenn die Schalter `fix`, `fax`, `fox`, `fux` oder `knax` geschrieben würden, funktionierte es trotzdem. Es hat sich aber so eingebürgert, die eckigen Klammern zu benutzen, um anzudeuten, dass das Wort den Compiler manipuliert.

der Compiler im Quellcode auf sie trifft. `[else]` , `[then]` , `[if]` , `[ifdef]` und `[ifndef]` sind solche Schalter.[3] Mit ihrer Hilfe gelingt dann das, was *Conditional Compiling* eigentlich meint. Es werden je nach Bedingung unterschiedliche Abschnitte aus dem Quellcode kompiliert. Damit kann man z. B. schön steuern, für welche Hardware der Code sein soll. Der hardware–spezifische Teil wird geladen, die Teile für eine andere Hardware jedoch nicht. Und das, was hardwareunabhängig ist, wird danach immer kompiliert.

Listing 1 zeigt, wie man so etwas in Forth realisieren kann. Der Quellcode stammt übrigens aus dem Mecrisp von Matthias Koch. Hab Dank, Matthias, für die freundliche Genehmigung!

Lasst uns mal tiefer einsteigen in das Beispiel von Matthias — trivial ist das ja nicht gerade. :-) Wie man sieht, läuft das Ganze auf die drei konditionalen Worte `[if]` , `[ifdef]` und `[ifndef]` hinaus. Was machen diese Schalter? `[if]` reagiert auf ein Flag im *Top of Stack* — Matthias hat das angedeutet in seinem Stack–Kommentar:

```
( ? -- )
```

Falls das `flag = false` ist, wird der Code dahinter übergangen. Bis wohin? Das wird dann vom `[else]` ermittelt.

Für die beiden anderen — `[ifdef]` und `[ifndef]` — wird ausnahmsweise mal nichts auf dem Stack übergeben. Sie sind als *Parsing Words* gestaltet und suchen sich das nächste *Token* im Quellcode — also das unmittelbar auf sie folgende Forthwort.[4] Im `[ifdef]`–Fall wird kompiliert, falls es das Token im Forth gibt. Im `[ifndef]`–Fall nur dann, wenn das Token noch nicht existiert. Jeder Schalter kann übrigens an jeder beliebigen Stelle einer Zeile stehen.

Interessant finde ich, dass `[then]` gar nichts tut. Es wird zwar unmittelbar (`immediate`) ausgeführt, aber so, wie eine leere Definition, also ein NOP. Und es ist nur dazu da, damit `[else]` weiß, wann es fertig ist. Es dient dem `[else]` also als schlichte Textmarke.

Ist nun `[else]` das eigentliche Arbeitstier dieser ganzen Konstruktion? Hm, bei näherer Betrachtung werden da zwar viele Zeichenketten–Vergleiche angestellt, aber *kompiliert* wird genau — nichts. Es ist also eine bloße Suchroutine, ein Parser sozusagen. Damit bewegt sich Forth von der Stelle an durch den Quellcode, wo es in einen der drei Schalter hineingelaufen ist.

Die eigentliche Kompilier–Arbeit findet also woanders statt. Und zwar ganz forthig immer *hinter* einem der drei Schalter. Wenn `[if]` zutrifft, wird alles kompiliert, was im Input–Stream des Quellcodes folgt. Bis zum `[then]` — was ja nur ein NOP ist — und danach einfach weiter, als sei nichts geschehen. Da kommt das `[else]` gar nicht zum Einsatz. Im anderen Falle schon. Dann wird `[else]` aktiviert und sucht, wie wir gleich sehen werden, den nächsten Schalter im Inputstream, damit Forth dort weitermachen kann. Das geschieht einfach dadurch, dass der Lesezeiger für den Inputbuffer genau dorthin gesetzt wird.

So, wie hangelt sich `[else]` denn nun durch den Quellcode? In der Definition `nexttoken` wird — wirklich, wie sein Name es sagt — nur das nächste Token aus dem Eingabestrom geholt, wobei, falls die aktuelle Zeile zu Ende sein sollte, die nächste Zeile mit `query` (was wiederum `accept` aufruft) angefordert wird. Kompiliert wird derweil gar nichts. Aber der Lesezeiger wird fortgeschrieben, sodass Forth dann dort weitermachen kann.

Die drei Schalter `[if]` , `[ifdef]` und `[ifndef]` benehmen sich also eigentlich wie Kommentare mit Zusatzaufgaben. Wer hätte das gedacht?

Eine Besonderheit im Mecrisp sind die *Faltbarkeits–Flags*: Sie sorgen dafür, dass die Konstantenfaltung auch über diese Schalter hinweg funktioniert.

Lange Kommentare

Da wir nun schon wissen, wie man den Inputstream manipulieren kann, schaffen wir auch einen *Long Comment*. Also einen Kommentar, der über mehrere Zeilen gehen kann.

```
... some-forth-code
(* Long comment ...
Kommentartext1
Kommentartext2
Kommentartext3
...
*)
more-forth-code ...
```

Schaut euch das Listing 2 gut an. Es stammt auch aus Matthias' Feder. Dort wird der Schalter `(*` definiert. Und ja, natürlich könnte der auch anders heißen als „bracket–star“. Es ist ja nur ein Forthwort.

Viel Spaß mit Forth wünsch ich euch.

Listing–1

```
1
2  \ Conditional compilation
3
4  \ Idea similar to http://lars.nocrew.org/dpans/dpansa15.htm#A.15.6.2.2532
5
6  : nexttoken ( -- addr len )
7    begin
8      token          \ Fetch new token.
```

[4] Forth liest den Quellcode zeilenweise in einen *Input–Buffer*. Die Abfolge davon wird *Input–Stream* genannt.

```
9     dup 0= while      \ If length of token is zero, end of line is reached.
10      2drop cr query   \ Fetch new line.
11    repeat
12  ;
13
14  : [else] ( -- )
15    1 \ Initial level of nesting
16    begin
17      nexttoken ( level addr len )
18
19      2dup s" [if]"     compare
20   >r 2dup s" [ifdef]"  compare r> or
21   >r 2dup s" [ifndef]" compare r> or
22
23      if
24        2drop 1+  \ One more level of nesting
25      else
26        2dup s" [else]" compare
27        if
28          2drop 1- dup if 1+ then  \ Finished if [else] is reached in level 1. Skip [else] branch otherwise.
29        else
30          s" [then]" compare if 1- then  \ Level completed.
31        then
32      then
33
34      ?dup 0=
35    until
36
37    immediate 0-foldable
38  ;
39
40  : [then] ( -- ) immediate 0-foldable ;
41
42  : [if]   ( ? -- )                0=  if postpone [else] then immediate 1-foldable ;
43  : [ifdef]  ( -- ) token find drop 0=  if postpone [else] then immediate 0-foldable ;
44  : [ifndef] ( -- ) token find drop 0<> if postpone [else] then immediate 0-foldable ;
45
```

Listing–2

```
1
2   : (* ( -- ) \ Long comment
3
4   begin
5     token  \ Get next token
6     dup 0= if 2drop cr query token then  \ If length of token is zero, end of line is reached. Fetch new line. Fetch new token.
7     s" *)" compare  \ Search for *)
8   until
9
10  immediate 0-foldable ;
```

Weitere Leserbriefe

Wie `if else then` gefunden wurden

Carsten fand neulich Eric Fischer's Geschichtsforschung zu der Frage, wann und wie eigentlich die heute geläufigen „conditional forms“ in die Programmiersprachen Einzug gehalten haben, speziell das `else`.

Erstaunlicherweise scheint das `else` im *deutschen Sprachraum* entstanden zu sein, als ungeschickte Übersetzung der Phrase

`falls wahr, dann A, sonst B.`

Im englischen Sprachgefühl ist `else` wohl mehr wie unser `noch`. Also so wie „what else?“ unserem „was noch?“ entspräche. Oder das „somewhere else“ meint „woanders“ zu sein. Jedenfalls assoziiert ein Brite oder Amerikaner zunächst kein Konditional mit dem Wort `else`. Und auch das Konditional `falls` machen die da drüben lieber zum `in case` und ungern zu einem `if`. Es war wirklich ein langer Weg bis zur Schreibweise von Abb. 1.

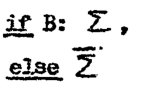


Abbildung 1: Die Erfindung der heutigen Konditionals.

Dass wir das im Forth noch anders sehen, nämlich umgekehrt polnisch, würde Eric zusätzlich verwirren, fürchte ich. Denn wir schreiben ja erst den Wert hin, dann das auswertende Konditional `if` und dahinter das, was kommt, falls es zutrifft, sonst — `else` — eben das andere. Danach — `then` — geht's weiter im Programm. mk

`https://github.com/ericfischer/if-then-else/blob/master/if-then-else.md`

Forth–200X–Treffen bei der EuroForth 2020

Anton Ertl

Technik

Wie auch die EuroForth selbst fand das Standardisierungstreffen diesmal online statt. Wir verwendeten dabei eine Reihe von Werkzeugen, von denen wir viele vermutlich auch in kommenden Treffen benutzen werden, auch wenn wir uns wieder persönlich treffen. Die Grundlage unserer Kommunikation bildete neben dem Web der Chat–Service Mattermost. Für die Videokonferenz verwendeten wir BigBlueButton auf dem Hoster `senfcall.de`. Beide bieten Chat–Funktionalität, dabei setzte sich das persistente Mattermost durch.

Für die Ausarbeitung von Texten setzte sich Mattermost gegenüber Etherpad durch, weil es da klarer ist, welche Variante von wem stammt, und es sich da auch natürlich ergibt, dass mehrere Vorschläge gepostet werden, und man sie vergleichen kann.

Für Abstimmungen innerhalb des Komitees wurden die Mechanismen von `forth-standard.org` verwendet. Änderung posteten wir dann als Proposal auf `forth-standard.org`. Über den komitee–internen Abstimmungsmechanismus wurden sie `forth-standard.org` zur Abstimmung gegeben. Im Gegensatz zu früher kann man damit klar nachvollziehen, wie der exakte Wortlaut des Vorschlags war, über den abgestimmt wurde, und was das exakte Ergebnis war.

`Forth-standard.org`

Bis jetzt ist `forth-standard.org` zwar eine gute Platform, um Vorschläge (*proposals*) zu präsentieren und zu diskutieren (entspricht der RfD–Phase des alten Systems), unterstützt aber noch keinen Umfrage–Mechanismus entsprechend dem CfV–Mechanismus des alten Systems. Immerhin können Vorschläge jetzt explizit mit einem Zustand (z. B. informal, formal etc.) gekennzeichnet werden.

Weil die Umfragen noch fehlten, sind eine Reihe von Vorschlägen weiterhin in der Warteschleife, bis es so einen Mechanismus gibt. Einige Vorschläge erschienen dem Komitee jedoch so unkontroversiell, dass das Komitee direkt darüber abgestimmt hat, ohne auf die Ergebnisse so einer Umfrage zu warten.

`Forth-standard.org` hat jetzt auch einen Mechanismus, um Fragen als beantwortet bzw. andere Beiträge als bearbeitet zu markieren („closed“); damit weiß das Komitee, dass es sich mit der Frage nicht mehr befassen muss. Wenn dann doch noch Unklarheiten bestehen, kann die Frage auch wieder geöffnet werden.

Inhaltliche Änderungen

`vocabulary` erzeugt eine benannte Wordlist *wid*. Wenn der Name des Vocabulary ausgeführt wird, wird die erste Wordlist in der Search–Order durch *wid* ersetzt. Eine typische Verwendung ist z. B.

```
vocabulary libwords
get-current
also libwords definitions
... \ Definitionen
previous set-current

\ in einer anderen Datei
also libwords
... \ Wörter aus libwords verwenden
previous
```

`vocabulary` wurde in Forth–94 (und Forth–2012) nicht standardisiert, ist aber weit verbreitet, sowohl in Systemen als auch in Programmen. Da auch keine Unterschiede zwischen den Systemen bekannt sind, wurde `vocabulary` direkt standardisiert.

`cs-drop` ergänzt die Kontrollfluss–Wörter (`begin again until if ahead then cs-pick cs-roll`) um die Möglichkeit, ein *orig* oder *dest* wegzuwerfen; dabei muss der Programmierer sicherstellen, dass jedes *orig* (das von einem Vorwärtssprung wie `if` oder `ahead` stammt) genau einmal aufgelöst wird.

`[If]` verwendet jetzt 0/nicht–0 wie `if`.

In der Warteschleife

Unter den auf den Umfrage–Mechanismus wartenden Vorschlägen ist auch der für *case insensitivity* nur für ASCII–Zeichen, siehe mein Artikel *Von Groß– und Kleinbuchstaben* in diesem Heft.

Weiters gibt es einen Vorschlag für eine neue Definition von *execution token*, der besser mit den Verwendungen dieses Terms zusammenpasst.

Sonstiges

Wir stellten klar, dass Referenzimplementierungen frei verwendet werden dürfen. Die Referenzimplementierung von `synonym` wurde entfernt, weil sie inkorrekt war, und eine korrekte zuviel systemspezifische Wörter verwendet. Die Referenzimplementierung von `[IF]` etc. wurde durch eine ersetzt, die auf case–insensitiven Systemen auch kleingeschriebene `[then]` etc. erkennt.

Wir beschlossen auch eine Reihe von Änderungen am Text des Dokuments, die für mehr Klarheit sorgen sollen, auf die ich aber hier nicht näher eingehe. Wer's genau wissen will, findet die Links auf `http://www.forth200x.org/meetings/2020-notes.html` bzw. im Allgemeinen auf `https://forth-standard.org/proposals`.

Das Komitee bearbeitete viele Fragen, die im letzten Jahr auf `forth-standard.org` gestellt worden waren, und beantwortete die meisten bzw. bewertete eine existierende Antwort als richtig.

Forth–Gruppen regional

Mannheim **Thomas Prinz**
Tel.: (0 62 71) – 28 30$_p$
Ewald Rieger
Tel.: (0 62 39) – 92 01 85$_p$
Treffen: jeden 1. Dienstag im Monat
Vereinslokal Segelverein Mannheim e.V. Flugplatz Mannheim–Neuostheim

München **Bernd Paysan**
Tel.: (0 89) – 41 15 46 53
bernd@net2o.de
Treffen: Jeden 4. Donnerstag im Monat um 19:00 in der Pizzeria La Capannina, Weitlstr. 142, 80995 München (Feldmochinger Anger).

Hamburg **Ulrich Hoffmann**
Tel.: (04103) – 80 48 41
uho@forth-ev.de
Treffen alle 1–2 Monate in loser Folge
Termine unter: http://forth-ev.de

Ruhrgebiet **Carsten Strotmann**
ruhrpott-forth@strotmann.de
Treffen alle 1–2 Monate im Unperfekthaus Essen
http://unperfekthaus.de.
Termine unter: https://www.meetup.com/Essen-Forth-Meetup/

Dienste der Forth–Gesellschaft

Nextcloud https://cloud.forth-ev.de

GitHub https://github.com/forth-ev

Twitch https://www.twitch.tv/4ther

μP–Controller Verleih **Carsten Strotmann**
microcontrollerverleih@forth-ev.de
mcv@forth-ev.de

Spezielle Fachgebiete

Forth–Hardware in VHDL microcore (uCore) — **Klaus Schleisiek**
Tel.: **(0 58 46) – 98 04 00 8**$_p$
kschleisiek@freenet.de

KI, Object Oriented Forth, Sicherheitskritische Systeme — **Ulrich Hoffmann**
Tel.: **(0 41 03) – 80 48 41**
uho@forth-ev.de

Forth–Vertrieb
volksFORTH
ultraFORTH
RTX / FG / Super8
KK–FORTH
Ingenieurbüro
Klaus Kohl–Schöpe
Tel.: **(0 82 66)–36 09 862**$_p$

Termine

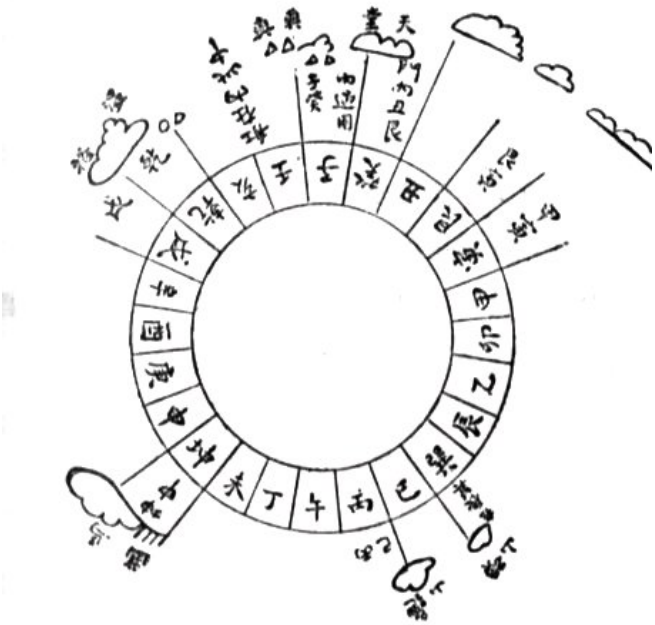
Donnerstags ab 20:00 Uhr
Forth–Chat net2o forth@bernd mit dem Key keysearch kQusJ, voller Key:
kQusJzA;7*?t=uy@X}1GWr!+0qqp_Cn176t4(dQ*

Montags ab 20:30 Uhr
Online Forthtreffen NRW
(Interessenten bitte beim Verein nachfragen)

27.–30.12.2020
37c3 wurde zu rC3 – remote Chaos Experience
http://https://events.ccc.de/

Leibhaftige Treffen sind keine bekannt geworden.

Details zu den Terminen unter http://forth-ev.de



Möchten Sie gerne in Ihrer Umgebung eine lokale Forthgruppe gründen, oder einfach nur regelmäßige Treffen initiieren? Oder können Sie sich vorstellen, ratsuchenden Forthern zu Forth (oder anderen Themen) Hilfestellung zu leisten? Möchten Sie gerne Kontakte knüpfen, die über die VD und das jährliche Mitgliedertreffen hinausgehen? Schreiben Sie einfach der VD — oder rufen Sie an — oder schicken Sie uns eine E–Mail!

Hinweise zu den Angaben nach den Telefonnummern:
Q = Anrufbeantworter
p = privat, außerhalb typischer Arbeitszeiten
g = geschäftlich
Die Adressen des Büros der Forth–Gesellschaft e.V. und der VD finden Sie im Impressum des Heftes.

Ob, und wenn, wo, die Tagung 2021 stattfinden wird,
ist noch immer völlig offen.
Wünscht euch was!

„Was ist wichtiger“, fragte der große Panda,
„die Reise oder das Ziel?“

„Die Gesellschaft!“, sagte der kleine Drachen.