Vierte Dimension

3/2020

Das Forth–Magazin

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

In dieser Ausgabe:

Poor Man's Recognizer

Forth–Prozessor K1 und SmallForth auf MAX1000

Peripherie für Mecrisp–Ice oder: Ein tiefer Tauchgang in die Welt der FPGAs

Natürliche Sprachen und Forth

Eine Einführung in das VIS–System

VolksForth–Update

Mitgliederversammlung und Tagung

36. Jahrgang — **Ausg. 3/2020** — 4,–€ *Organ der Forth–Gesellschaft e.V.*

Fahrtregler - Lichtanlagen - Soundmodule - Modellfunk

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

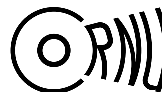
Möchten Sie die Programmiersprache Forth erlernen oder sich in den neuen Forth–Entwicklungen weiterbilden? Haben Sie Produkte auf Basis von Forth und möchten Mitarbeiter in der Wartung und Weiterentwicklung dieser Produkte schulen?

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
Entwicklungsbüro Dr.–Ing. Egmont Woitzel

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`

Leserbriefe und Meldungen 5

Poor Man's Recognizer 7
Klaus Schleisiek

Forth–Prozessor K1 und SmallForth auf MAX1000 11
Klaus Kohl–Schöpe

Peripherie für Mecrisp–Ice oder: Ein tiefer Tauchgang in die Welt der FPGAs 15
Matthias Koch

Natürliche Sprachen und Forth 23
Jens Storjohann

Eine Einführung in das VIS–System 29
Martin Bitter

VolksForth–Update 32
Carsten Strotmann

Mitgliederversammlung und Tagung 36
Carsten Strotmann

Titelbild [recognizer square brackets]   Autor M. Kalus

Quelle: MS Paint 3D des Autors; für 4d2020–03, 8/2020

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,

wie erhofft, ist zum Herbstanfang wieder ein Heft fertig geworden.

Sogar 36 Seiten stark diesmal! Liebe Leser, liebe Forthfreunde in aller Welt, ihr seid tüchtig! Macht weiter so, Forthiges sammeln und herschicken!

Zustandegekommen ist das Magazin wieder durch Nachfragen bei Leuten, die sich mit Forth befassen. Als Korrespondenz, als Artikel oder als Kolumne. Ihr braucht keine besonderen Formate zu beachten für euren Text, da machen wir schon was draus. Wir, das waren wieder Wolfgang Strauß, Bernd Paysan, Ewald Rieger und ich (Michael Kalus). Nach wie vor ist Wolfgang unser Lektor, Bernd hält die Technik in Gang, kann die wildesten Sprachen einbauen und damit unsere Laune hochhalten. Ich bin der Sammler. Ewald schließlich sorgt für den feinen Druck und flotten Versand der Hefte — und sucht nun einen würdigen Nachfolger.

Leider gibt es eine schlechte Nachricht: Unsere *Mitgliederversammlung 2020* mit allem Drum und Dran entfällt. Das ist wirklich jammerschade, finde ich.

Die EuroForth fand auch nicht in Rom, sondern online statt, also von zu Hause aus, vom 4. bis 6. September. Aus Los Angeles, Hong Kong und Tokyo, von Helsinki bis Kapstadt gab es Interessenten. Ein Format mit Zukunft? Bis zu 12 Teilnehmer waren zeitgleich in der Video–Konferenzschaltung, dazu noch einige pur Audio. Komplett gesehen auf *Twitch* haben es schon gut 50 Leute, zeitversetzt gab es 329 Aufrufe mit 9364 angesehenen Minuten (bis heute) — Gerald Wodni wird hoffentlich weiter berichten.

Nun zum Heft. Klaus Schleisiek nimmt uns mit ins Innenleben von Forth, beleuchtet das Konzept der *Recognizer* kritisch. Auf der EuroForth wird es auch thematisiert worden sein. Bin gespannt, wie sich das in den Forth–Standard einpassen wird. Dann tauchen wir ab in die Hardware. Denn was ist eine MCU ohne Ports, Interrupts und all die Peripheriebausteine zur Welt da draußen? Daher nimmt Matthias Koch euch mit auf einen *Tauchgang in die Welt der FPGAs* mittels *VERILOG*. Zum eigenen Forth–Prozessor *K1* und *SmallForth* führt uns dann Klaus Kohl–Schöpe. Der *MAX1000* ist seine Plattform dafür. Aber nicht nur dafür, auch für liebe alte Bekannte wie den *6502* geht das! Also, falls deren Vorrat zur Neige geht — selber machen. In die andere Richtung nimmt uns Jens Storjohann mit. Dass das Konzept, in seiner „natürlichen Sprache“ zu programmieren, leichter gesagt als getan ist, zeigt er uns kundig auf. Und zeigt, dass Forth da auch für andere Sprachen als Englisch was zu bieten hat. Und dass es Sprachkonzepte gibt, die in Forth noch gar nicht beachtet worden sind. Danach holt uns Martin Bitter vollends zurück ins praktische Programmieren. Zurück in diese winzigen Alleskönner an MCUs. Obschon, so ein *Longan Nano* ist sooo klein ja nicht — innen! Und mit einem *VIS–System* an Board geht da manches leichter. Schließlich zeigt Carsten Strotmann all die wunderbaren Updates auf, die das gute alte *VolksForth* inzwischen bekommen hat, um da überall mithalten zu können. Ich staune, was sich da alles getan hat!

Euch allen eine gute Zeit. 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-2020-03

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

Ulrich Hoffmann	Kontakt: Direktorium@Forth-ev.de
Bernd Paysan	
Carsten Strotmann	

Neubesetzung des Forth–Büros

Durch die Absage der Mitgliederversammlung in 2020 hat sich nun auch die Neuvergabe des Forth–Büros verschoben. Ewald Rieger, welcher das Forth–Büro in den letzten Jahren geführt hat, wird dankenswerterweise das Büro doch noch für ein paar weitere Monate führen. Er möchte jedoch gerne das Forth–Büro ganz abgeben. So ist es der Plan, das Forth–Büro im Jahr 2021 in die Hände eines anderen Mitglieds zu legen. Zu den derzeitigen Aufgaben des Forth–Büros gehören:

- Führung der Kasse und der Konten der Forth–Gesellschaft.
- Erstellen des jährlichen Kassenberichts.
- Führung der Mitgliederdatenbank.
- Versand (aber nicht Erstellung) der Vereinszeitschrift „Vierte Dimension".

Da die Forth–Gesellschaft kein großer Verein ist, hält sich die Arbeitsbelastung für das Forth–Büro in Grenzen. Eine gewissenhafte und zuverlässige Arbeitsweise ist jedoch erforderlich.

Wir bitten Mitglieder, die bereit sind, diese Aufgaben für den Verein zu übernehmen, sich beim Direktorium zu melden.[1] Carsten Strotmann

Drachenpreis

Der *Drachenhüter 2019*, Matthias Koch, hat den *Swap* neulich mir übergeben. Den sollte ich stellvertretend für ihn auf der Forth–Tagung weitergeben an den nächsten Würdenträger, der vom Drachenrat dort zu bestimmen gewesen wäre. Doch die Covid–19–Pandemie hat die Zusammenkunft verhindert und es gibt keinen neuen Drachenhüter in 2020.

So wartet Swap nun weiter geduldig und hofft auf bessere Zeiten. Nicht, dass es ihm schlecht ginge hier bei mir. Aber langweilig ist es ihm doch. Lieber wäre er dort, wo aktuell mehr Forthiges passiert. So vertreibt er sich die Zeit damit, mir über die Schulter zu schauen beim Layout unseres Forth–Magazins. Und flüstert mir schon mal seine Wünsche ins Ohr, zu wem er mal gerne möchte für ein Jahr. mk

Case–Sensitivität — soll man oder nicht?

Und sollte man lieber `gruen` oder `grün` definieren? Wurscht? „GRÜN und grün sind doch gleich im case*in*sensitiven Forth!?" Nö!

```
\ Gforth 0.7.3
: GRÜN cr ." GRÜN" ;
: grün cr ." grün" ;
GRÜN
grün
grÜn
GRüN
cr
: BLAU cr ." BLAU" ;
: blau cr ." blau" ;
BLAU
blau
cr bye

\\
Ergibt:
mka@mka-HP-355-G2:~$ gforth gruen.fs

GRÜN
grün
GRÜN
grün
redefined BLAU with blau
blau
blau

mka@mka-HP-355-G2:~$
```

Man stutzt!

Preisfrage: Wieso kommt da die Warnung „redefined BLAU with blau", aber keine Warnung bei der Definition von „grün"?

mk

Forth auf Rang 47 von 50 bei Spectrum IEEE

So rangiert Forth immer noch in den Top 50 der Programmiersprachen. Es erzielte 23.7 von 100 möglichen Bewertungspunkten. Als alleiniger Anwendungsbereich wird dort aber „Embedded" vermerkt. In den Bereichen „Web", „Enterprise" und „Mobile" komme es praktisch nicht vor. Hm, nun ja, Veröffentlichungen dazu habe ich auch keine gesehen. Doch bei regionalen Treffen, Tagungen und der EuroForth hörte ich auch, dass Forth nicht nur auf kleinen Systemen zu Hause ist.

Danke, Klaus Kohl–Schöpe, für den Hinweis. mk

`https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020`

comp.lang.forth war von Google vorübergehend gesperrt

> „Warnung bezüglich gesperrter Inhalte. Die Gruppe, die Sie anzeigen möchten (comp.lang.forth), enthält anscheinend Spam, Malware oder andere schädliche Inhalte. Die Inhalte dieser Gruppe sind jetzt nur noch für zugriffsberechtigte Personen im Lesemodus verfügbar. Gruppeninhaber können Widerspruch erheben, nachdem sie Maßnahmen zur Entfernung potenziell anstößiger Inhalte aus dem Forum unternommen haben. Weitere Informationen zu Inhaltsrichtlinien bei Google Groups erhalten Sie in unserem Hilfeartikel zum

[1] E–Mail: `direktorium@forth-ev.de`

Thema Missbrauch und unseren Nutzungsbedingungen.“ (26. Juli 2020 `https://groups.google.com/forum/#!forum/de.comp.lang.forth`

So war es neulich zu lesen, wollte man mit dem googleschen News–Reader mal schauen, was c.l.f Neues hat.

Anlass könnte ein Post vom 23.07.2020 gewesen sein, in dem ein User alten Spam zweier Zankhähne von Anfang des Jahres hochgeholt und in seinem Post verdichtet hatte. Und so wimmelte es darin nur so von Verbal–Injurien, auch unter der Gürtellinie — wirklich hässlich. Google scheint da pingelig zu sein. Nun, inzwischen geht's auch wieder mit Google, das c.l.f zu lesen.

Um das Usenet uneingeschränkt nutzen zu können, kauft man sich besser bei einem neutralen Zugangsprovider ein, z. B. beim *Individual.NET* `<https://news.individual.net/overview.php>` der FU Berlin für 10 Euro/Jahr.

Das „... Usenet ist ein verteiltes System; solange noch eine Person das System (und die Gruppe ‚comp.lang.forth‘) benutzt, existiert die Gruppe noch. Abschalten kann man das nur, wenn das gesamte Internet abgeschaltet wird.“ (cas)

News–Reader gibt es zuhauf. Die Experten haben dafür ihr *Emacs* zur Hand. Ich benutze nun *Pan*, eine komfortable GUI, auf Linux Mint.

Wer nur mal schauen will, ob das so geht, kann bei der FU Berlin das Probe–Abo wählen. mk

Mecrisp–Quintus 0.29 for RISC–V 32 IMC on GD32VF103CB

Werkelt ihr auch schon damit? Z. B. auf einem Longan Nano? Phantastisch, oder?

Nur, was bedeutet IMC? Matthias Koch hatte die Antwort parat:

„IMC, das ist die Befehlssatz–Ausbaustufe. RV32I ist die Basis, die immer da ist, und es gibt mehrere Erweiterungen: Mecrisp–Quintus unterstützt jetzt _I_nteger, _M_ultiply und _C_ompressed, wobei das letzte spezielle Kurzformen von häufig benutzten Befehlen verwendet, um Speicherplatz zu sparen.

Der GD32VF103CB ist eigentlich ein RV32IMAC, aber für die _A_tomic–Erweiterung habe ich noch keine Unterstützung geschrieben.“

Bin gespannt, was da noch alles kommt. mk

Forth–Prozessor K1 und SmallForth auf MAX1000 — Update

Seit Klaus Kohl–Schöpe den Artikel dazu für dieses Heft eingereicht hat, im Sommer, ist die Entwicklung nicht stehengeblieben. Inzwischen wurde ergänzt:

- Im Targetcompiler wird EXIT nun optimiert, wenn möglich in die letzten Befehle integriert.[2]
- Jetzt sind die kompletten 32 KByte Flash–Speicher im MAX1000 für Daten und Programm verwendbar.
- Beim Reset des Boards wird geprüft, ob die Taste neben den LEDs gedrückt wurde. Falls nicht, dann wird das Programm aus dem Flash gestartet (`'init` enthält die Startroutine). Und falls doch, dann wird der ursprüngliche SmallForth–Kern gestartet.
- Einige kleinere Fehler wurden korrigiert.

Die neue Version — zusammen mit der alten Version — ist schon auf `mcforth.net` verfügbar.

Auf Anfrage gibt es auch eine Version für das *CYC1000–Modul*. Jedoch hat es aktuell noch kein Autostart, weil der Zugriff auf das externe Quad–SPI–Flash noch in Arbeit ist.

Ebenfalls in Arbeit ist die Implementierung eines *6502–Prozessors* auf dem MAX1000.

Wie auch beim K1 beinhaltet dieses Paket wieder Target–Assembler und Simulator unter mcForth.

Die ersten Programme laufen schon und das SmallForth ist in Arbeit. mk

noForth RV

We are busy with an ITC version of noForth for the *GD32VF103*. It has the same architecture as noForth for the MSP430. It comes in two basic versions, one with 32–bit tokens and one with 16–bit tokens. With 16–bit tokens, the high level Forth code is 50% more compact and the compact version is only a few percent slower!

For now, noForth RV has been tested on the *Longan Nano* and the *SEEED RISC–V board*. It will be available mid October 2020 on the noForth website, because we have to finish the documentation. Willem Ouwerkerk

Fortsetzung der Rubrik auf Seite 28

[2] In der alten Version wurde der Opcode für EXIT — vorwiegend am Ende einer Definition — mit `;` als separatem Befehl abgelegt. In der neuen Version wird EXIT möglichst in den letzten Befehl eingefügt, was jeweils 2 Bytes und einen Befehlstakt spart. Weitere Optimierungen wären möglich, sind aber nicht so einfach zu integrieren. Beispiele dazu wären `2dup +` oder `@ +`. Das ist die Stärke eines Befehlssatzes, der zwei Datenstack–Werte, den Returnstack–Wert, Speicherinhalt und EXIT fast beliebig kombinieren kann. (KKS)

Poor Man's Recognizer

Klaus Schleisiek

µForth is the cross compiler for µCore. When I started the project on top of Gforth_062, I found a simple and powerful solution to cross compilation by patching the way [and] behave. The current version of Gforth_079 uses Matthias Trute's recognizer mechanism and I gave it a try as an alternative to patching.

The result of this endeavor was so complicated that I returned to the patch solution, which I regard as much more readable. It can be implemented on all versions of Gforth above 062 albeit minute differences, because Gforth's interpreter/compiler is a moving target. Alternatively, the patch version can be implemented using "Poor Man's Recognizer" in a way that it reads like recognizer code, though drastically simplified.

In order to clarify the issues, I will first present and discuss the patch solution, followed by the Gforth recognizer solution, and finally followed by the "Poor Man's Recognizer" solution.

µForth has two very different modes of operation:

1. `host-compile` compiles into the host's dictionary in order to add capabilities to µForth.

2. `target-compile` compiles into the target's dictionary in order to produce code for µCore.

These two modes require very different parsers, which are both different from the native Gforth parser, because µForth includes an OOP package for the sake of operator overloading (i.e. a `@` may do very different things depending on the object that preceded it). This would be straightforward, if `[` and `]` were deferred, but they are not and therefore, `[` and `]` have to be patched instead. For the sake of argument, I will only discuss the `target-compile` mode.

Patching

The patch magic is done in a portable way by `becomes`, which takes an xt and the name of an existing word:

After patching, everything still works ok, because we initialized both `'interpreter` and `'compiler` with Gforth's native parser code. Now we can define the new parser for the target.

First some lower–level words:

`d>target` `( d.host -- d.target )` converts a host's double number into a double number for µCore, because very often the data width of the host is greater than Core's data width.

`ClassContext` is a variable pointing to a linked list of methods. If it is zero, µForth's dictionary will be searched.

`search-classes` searches an object's list of methods down its inheritance order.

`debugger-wordlist` is a variable pointing to a wordlist of commands, which will be searched with preference when interactively debugging the target system.

`t_lit,` `( n -- )` compiles a number on the host's stack as a literal in the target system.

`>t` `( n -- )` transfers a number on the host's stack to the target's stack.

`not-found` `( addr len -- )` displays the string that was not found and aborts.

Now we are prepared to define `target-compiler`, the target's interpreter/compiler classical style, everything in one single definition:

When an object has been compiled or executed, `ClassContext` will have been set to its methods list to be searched. If a method is found, it will be executed producing target code and we are done. If nothing was found, we display the `not-found` message and abort.

Otherwise, we check whether we are interactively debugging, in which case we will search `debugger-wordlist` next. If it was a command, it will be executed and we are done. If it was no command, we will search the target's dictionary. If it was a target word, it will be executed producing target code and we are done.

If it was neither a command, nor a target word, `snumber?` will try to convert the string into a number. When successful, we may be

1. compiling. In this case, we have to compile the number as literal(s) into the target code and we are done.
2. debugging. In this case we transfer the number on the host's stack to the target's stack and we are done.
3. interpreting. In this case we throw away the double number flag and we are done, leaving the number on the host's stack.

You must admit, this is VERY different from what the native Gforth compiler does.

Finally, we can define `target-compile` that will activate the `target-compiler`, set `Targeting` to `true` and set the dictionary search order for target words.

Listing

```
1   \ Patching
2
3   : becomes ( <word> new-xt -- ) \ make existing <word> behave as new-xt
4      >r   here   ' >body dp !   postpone AHEAD
5      r> >body dp !   postpone THEN   dp !
6   ;
7   Variable 'interpreter   ' interpreter 'interpreter !
8   Variable 'compiler      ' compiler    'compiler !
9
10  :noname ( -- )  'interpreter @ IS parser   state off ;   becomes [
11  :noname ( -- )  'compiler @    IS parser   state on  ;   becomes ]
12  : target-compiler ( addr len -- )
13  \ search for methods
14     ClassContext @ IF  2dup search-classes ?dup
15                        IF  nip nip name>int execute  EXIT THEN
16                        not-found
17                    THEN
18  \ search for commands while debugging
19     dbg? IF  2dup debugger-wordlist search-wordlist
20              IF  nip nip name>int execute  EXIT THEN
21          THEN
22  \ search the target's dictionary
23     2dup find-name ?dup IF  nip nip name>int execute  EXIT THEN
24  \ try to convert to an integer number
25     2dup 2>r snumber? ?dup 0= IF  2r> not-found  THEN  2rdrop
26     comp? IF  0> IF  d>target swap t_lit,  THEN  t_lit,  EXIT THEN
27     dbg?  IF  0> IF  d>target swap >t      THEN  >t      EXIT THEN
28     drop
29  ;
30  : target-compile ( -- )
31     ['] target-compiler dup 'interpreter !   dup 'compiler !   IS parser
32     Targeting on   Only Target also
33  ;
34
```

Recognizers

Ok, now Gforth's recognizers. The lower level words we need have already been explained above.

The debugger words are handled separately, so we define our first recognizer `rec-debugger`.

Hm, we see that we have to do more than just define a recognizer, we also have to define recognizer types like `rectype-name` and `rectype-null`, which are defined in Gforth_079.

Now come the methods and the Forth dictionary. Remember, no matter whether we are in interpret or compile mode, we always execute the command of the target code compiler. So we have to define `rectype-target`, our first rectype, because the standard one, `rectype-name`, does not do what we need.

Now we can define `rec-methods`, the recognizer for methods compilation, and `rec-target`, the one for the target's dictionary:

Ok that takes care of the methods and normal Forth words. Now we must turn to numbers. Again, we cannot use the normal types `rectype-num` and `rectype-dnum`, because we may be compiling code for the target. Therefore, we have to define `rectype-tnum`, `rectype-tdnum`, and finally, we can define `rec-tnum` for the target compiler.

Now we are prepared to define `target-recognizer`, the recognizer for target compilation consisting of the four sub–recognizers `rec-methods`, `rec-debugger`, `rec-target`, and `rec-tnum`. Then we can define `target-compile` that will activate the `target-recognizer`, set `Targeting` to `true` and set the dictionary search order for target words.

Frankly, I find the patch version of the first chapter a lot easier to read.

Why is this so?

In the patched version, everything is close together in one definition: from string extraction through wordlist lookup to the final execute. Therefore, you may modify everything when needed. We could even do without patching, if `[` and `]` were deferred. Its drawback: Everything is merged into a single definition with a more or less complex conditional structure.

The recognizer version gets rid of the complex conditional structure, but it has newly invented complexities: Separation of the search action from semantic interpretation (rectypes), and the final execute is completely hidden.

Therefore, in order to understand the recognizer code, you must make yourself a mental image that merges the search action and its semantic interpretation, and you better understand the underlying recognizer machine that does the final execute for you. Let alone the definition of the final `-recognizer`, which is not a colon definition but a data structure that has to be read backwards. Challenging conditions for reliable code that ought to be easy to understand!

Listing

```
1   \ Recognizers
2
3   : rec-debugger ( addr u -- nt rectype | rectype-null )
4      dbg? IF  debugger-wordlist find-name-in
5              dup IF  rectype-name  EXIT THEN  dup
6           THEN  2drop   rectype-null
7   ;
8   :noname   name>int execute-;s ;  \ interpret action
9   dup                              \ compile action, same as above
10  ' lit,                           \ postpone action
11  rectype: rectype-target
12  : rec-methods ( addr u -- nt rectype | rectype-null )
13     ClassContext @ 0= IF  2drop rectype-null  EXIT THEN
14     2dup search-classes ?dup IF  nip nip rectype-target  EXIT THEN
15     not-found
16  ;
17  : rec-target ( addr u -- nt rectype | rectype-null )
18     find-name ?dup IF  rectype-target  EXIT THEN  rectype-null
19  ;
20  ' noop
21  :noname ( n -- )  comp? IF t_lit, EXIT THEN  dbg? IF >t EXIT THEN  drop ;
22  dup
23  rectype: rectype-tnum
24
25  ' noop
26  :noname ( d -- )  d>target swap
27     comp? IF  t_lit, t_lit,  EXIT THEN
28     dbg?  IF  >t     >t      EXIT THEN  2drop ;
29  dup
30  rectype: rectype-tdnum
31
32  : rec-tnum ( addr u -- n/d table | rectype-null )
33     snumber? ?dup 0= IF  rectype-null  EXIT THEN
34     0> IF  rectype-tdnum  ELSE  rectype-tnum  THEN
35  ;
36  $Variable target-recognizer
37  align here target-recognizer !
38  4 cells , ' rec-tnum A, ' rec-target A, ' rec-debugger A, ' rec-methods A,
39
40  : target-compile ( -- )
41     target-recognizer TO forth-recognizer
42     Targeting on   Only Target also definitions
43  ;
44
```

Poor Man's Recognizer

Here is a simplified version for a recognizer type construct that does not need all the overhead of Gforth's recognizer mechanism. It is not even a new mechanism, it is just exploiting Forth's capability to manipulate a return address on the return stack.

Foremost, we want to get rid of the convoluted control structure of the patch version. I.e. we want to be able to write down an easy to read specification for `target-compiler`, which behaves identical to the patch version of the first chapter.

In the recognizer chapter we defined `target-recognizer` as a reverse list of simple sub–recognizers, each of which only does a single string matching attempt. For the "Poor Man's" version we define `target-compiler`, which basically does the same thing. It has the limitation that it can not change its behaviour dynamically. But who needs this flexibility? After all, the parser has a much lower change rate compared to the dictionary's search order.

Each sub–recognizer has identical stack behaviour:

- It takes a counted string as input argument.
- If there is no match, it leaves the counted string unchanged handing it over to the next sub–recognizer.
- If there is a match, the input arguments are dropped, the return address into `target-compiler` is thrown away and therefore, we continue to execute the word that called `target-compiler`.
- If none of our sub–recognizers did match, we bump into `not-found`.

As in the recognizer version, we define four sub–recognizers: `method-find`, `debugger-find`, `target-find`, and `target-number`. Finally, we define `target-compile`, which happens to be identical to the definition of the patch version.

For completeness, I also included the definition for `host-compile`, consisting of the sub–recognizers `host-find` and `host-number`, and finally the recognizer `host-compiler`.

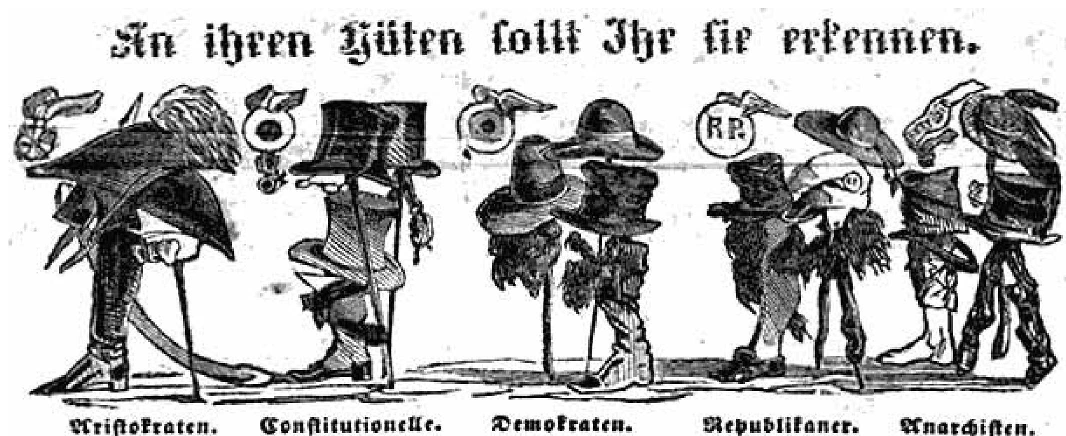
Listing

```
1   \ Poor Man’s Recognizer
2
3   : method-find ( addr len -- addr len | rdrop )
4      ClassContext @ 0= ?EXIT
5      2dup search-classes ?dup
6      IF  rdrop nip nip   name>int execute  EXIT THEN
7      not-found
8   ;
9   : debugger-find ( addr len -- addr len | rdrop )
10     dbg? 0= ?EXIT
11     2dup debugger-wordlist search-wordlist
12     IF  rdrop nip nip   name>int execute  EXIT THEN
13  ;
14  : target-find ( addr len -- addr len | rdrop )
15     2dup find-name ?dup IF  rdrop nip nip   name>int execute  THEN
16  ;
17  : target-number ( addr len -- addr len | rdrop )
18     2dup 2>r   snumber? ?dup 0= IF  2r> EXIT THEN  2rdrop rdrop
19     comp? IF  0> IF  d>target swap t_lit,  THEN  t_lit,  EXIT THEN
20     dbg?  IF  0> IF  d>target swap >t      THEN  >t      EXIT THEN
21     drop
22  ;
23  : target-compiler ( addr len -- )
24     method-find debugger-find target-find target-number not-found
25  ;
26
27  : host-find ( addr len -- addr len | rdrop )
28     2dup find-name ?dup 0= ?EXIT   rdrop nip nip
29     comp? IF  name>comp  ELSE  name>int  THEN   execute
30  ;
31  : host-number ( addr len -- addr len | rdrop )
32     2dup 2>r snumber? ?dup 0= IF  2r> EXIT THEN   2rdrop rdrop
33     comp? IF  0> IF  swap postpone Literal  THEN  postpone Literal  EXIT THEN
34     drop
35  ;
36  : host-compiler ( addr len -- )   host-find host-number not-found ;
37
38  : host-compile ( -- )
39     ['] host-compiler dup 'interpreter !   dup 'compiler !   IS parser
40     Targeting off   Only Host also definitions
41  ;
42
```

Summary

I am sorry to say, but Matthias Trute’s recognizer mechanism is over–engineered and therefore, it is difficult to understand, explain, and extend. An indication for this is the sheer number of articles, which try to explain it.[1] In addition, it is more flexibel than actually needed.

The “Poor Man’s Recognizer” had been a vague idea for several years. After writing the first two chapters of this paper, I finally implemented it. It exceeded my expectations as far as simplicity is concerned and therefore, this is what I will add to my Forth toolbox.



[1] In the issues of our magazine: Recognizer, Matthias Trute, 4d2011–02; Recognizer, Bernd Paysan, 4d2012–02; Recognizer 2, Matthias Trute, 4d2014–03/04; Recognizer 3, Matthias Trute, 4d2017–04.

Forth–Prozessor K1 und SmallForth auf MAX1000

Klaus Kohl–Schöpe

Ende 2018 hatte ich vier günstige FPGA–Boards vorgestellt — entwickelt und vertrieben durch Arrow und Trenz. Schon damals war meine Idee, auf diesen Boards einen Forth–Prozessor im Stil von J1 zu realisieren. Jedoch gab mir erst der Lockdown der letzten Monate genügend Zeit für dieses Projekt. So kann ich euch nun die schon funktionierende Version vorstellen. Das entsprechende Quartus–II–v18.1–Projekt *und das zugehörige* SmallForth *(kurz SMAF) kann von meiner Webseite* `http://www.mcforth.net` *heruntergeladen und getestet werden. Ich kann nur hoffen, dass ich für mein erstes eigenes FPGA–Projekt nicht gesteinigt werde.*

Vorgeschichte

Als Mikrocontroller–Spezialist arbeite ich aktuell fast ausschließlich mit C(++), jedoch mit einer langen Erfahrung in Assembler, Basic und Forth. Dabei hatte ich auch häufig intensiven Kontakt mit Forth–Prozessoren wie NC4000, RTX2000 oder FPR1600 und oft darauf eigene Forth–Versionen realisiert. Auch bei meiner letzten Implementierung des *mcForth* wurde für den virtuellen Prozessor ein eigener Befehlssatz definiert und in Assembler simuliert.

Da ich mich aktuell auch wieder mehr mit FPGAs befassen darf und ich mir auch schon längere Zeit verfügbare IPs wie z. B. den J1 von James Bowman angesehen habe, nutzte ich die durch Corona verursachte Freizeit und Urlaub, um mich sowohl in *Verilog* als auch in Prozessordesign auf FPGA einzuarbeiten. Was lag dabei näher, als diesen J1–Ansatz zu verwenden und auf einer der damals vorgestellten günstigen FPGA–Boards zu implementieren. Die Wahl fiel auf das MAX1000–Board (Abb. 6), weil der MAX10 flashbasiert ist und damit sofort startet, der Chip mit 8 K LE und 42 KByte RAM groß genug für das Forth–IP ist und genügend Speicher für das SmallForth hat. Außerdem erlauben Schnittstellen wie UART (über FDTI–Chip), 8 LEDs, Taster, Beschleunigungssensor und je 8 MByte großes SDRAM und SPI–Flash viele Anwendungen.

Von J1 zu K1

Als „Geschädigter“ von Forth–Prozessoren versuche ich, Eigenheiten wie 16 oder 32 Bit pro Adresse oder weit vom aktuellen Standard abweichende Forth–Versionen ohne Prüfung von Kontrollstrukturen zu vermeiden. Deshalb war meine erste Aktivität die Überarbeitung des Befehlssatzes und des Speicherinterfaces.

Im Einzelnen gab es folgende Wünsche:

- Speicher ist als 8 Bit pro Adresse organisiert
- Befehle und Variablen sind 16–bit–aligned
- High–Endian (ist einfacher im Dump zu lesen)
- Literals können ±16K abdecken (Bit 15 <= Bit 14)
- Calls verwenden absolute Adressen (addr/2 + $8000)
- Sprünge sind relativ ±4 KByte
- Zusätzlich ein Sprungbefehl für die `FOR...NEXT`–Schleife
- Arithmetik ist erweitert und auch bei `@` nutzbar
- Neue Befehle: `UM* * Divstep SP! EXECUTE GOTO`

Dazu wurde der Befehlssatz umgestellt, wie man im Vergleich von Abb. 1 bis Abb. 3 sehen kann. In Abb. 4 sind dann noch die verwendeten Abkürzungen aufgelistet.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	value															Literal (0..32K)
0	0	0	target													Sprung
0	0	1	target													Sprung wenn T=0
0	1	0	target													Call
0	1	1	;	ALU				>N	>R	!		rr		ss		Arithmetik

Abbildung 1: Befehlssatz J1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	literal															Literal (±16k)
1	0	addr														Call
1	1	0	0	relativ												Sprung
1	1	0	1	relativ												Sprung wenn T=0
1	1	1	0	relativ												Sprung wenn R<>0
1	1	1	1	0	;	>R	>N	ALU				BW	@!	ss		Speicherzugriff
1	1	1	1	1	;	>R	>N	ALU				rr		ss		Arithmetik

Abbildung 2: Befehlssatz K1

ALU	J1	K1	K1 special
0	T	Tx	R
1	N	N	RA (R mit Bit 0=0)
2	T+N	N+Tx	Tx+1
3	T and N	N-Tx	Tx-1
4	T or N	N and Tx	Tx+2
5	T xor N	N or Tx	Tx-2
6	~T	N xor Tx	Carry (0 oder 1)
7	N = T ?	~Tx	INV14: Tx xor $4000
8	N < T ?	N ashift Tx	Tx/2
9	N rshift T	N rshift Tx	Tx/2 unsigned
11	R	N lshift Tx	Tx*2
10	T-1	Tx = 0 ?	Tx < 0 ?
12	[T]	N u< Tx ?	DSP
13	N lshift T	N < Tx ?	RSP
14	DSP	Divstep	SP!
15	N u< T ?	*	UM*

Abbildung 3: Vergleich ALU–Befehle J1 und K1

Abkürzung	Bedeutung
TOS oder T	Oberster Datenstack-Wert
Tx	TOS oder Memory
~T oder ~Tx	Invert T(x) = T(x) xor $FFFF
NOS oder N	Zweiter Datenstack-Wert
TOR oder R	Oberster Returnstack-Wert
RSP	Returnstackpointer
DSP	Datenstackpointer
;	Exit: PC <= TOR
>N	Kopiere TOS nach NOS
@!	Lesen (0) oder Schreiben (1)
>R	Kopiere TOS nach TOR
BW	Byte (0) oder 16-Bit-Wort (1)
value	Wert (Bit 15 wird 0)
literal	Wert (Bit 15 wird Bit 14)
addr	PC wird addr*2
relativ	PC wird PC+2±(relativ*2)
u<	Vergleich: Unsigned kleiner
INV14	Bit 14 invertieren (für Literal)
ashift	Shift rechts mit Bit 15=Bit 14
rshift	Shift rechts mit Bit 15=0
lshift	Shift links mit Bit 0=0

Abbildung 4: Abkürzungen

Der Speicher wurde mit 32 KByte für Programme so belassen (war 16K*16), weil sowohl der interne Speicher des MAX10 als auch die verwendbaren Opcodes dies limitieren. Der restliche Speicher ab $8000 enthält die Schnittstellen zur Peripherie und evtl. in Zukunft noch Videospeicher. Beide Stacks verwenden auch einen 9–Kb–Speicher des MAX10 und haben damit je 256 Einträge. Die Stackpointer sind aber 9 Bit, um Überläufe zu erkennen.

Bei Literalen hat der J1 die unteren 15 Bits verwendet und damit 0 . . . 32767 realisieren können. Negative Zahlen konnten durch den `INVERT`–Befehl erreicht werden. Da in Anwendungen eher auch negative Zahlen und Adressen im $FFxx–Bereich vorkommen, habe ich definiert, dass bei Literals das Bit 14 auch in Bit 15 übernommen wird und damit -16364 (=49152) bis 16383 mit einem Takt realisiert wurde. Ansonsten invertiere ich vorher Bit 14 bei einem Literal und führe danach den `INV14`–Befehl aus.

Auch bei der Definition von Calls und Sprüngen habe ich einiges geändert. Zum einen kann immer noch der gesamte 32–KByte–Speicher bei Calls erreicht werden und da die Sprünge normalerweise nur in Bedingungen innerhalb einer `:`–Definition vorkommen, sollten ±4 KByte ausreichen. Der zusätzliche Sprungbefehl für die `FOR ... NEXT`–Schleife erniedrigt TOR um 1 und springt, solange der Wert vorher ungleich 0 war. Ist er 0, dann wird der Wert vom Returnstack entfernt und die Schleife verlassen. Gleichzeitig `>R` und `EXIT` sollte eigentlich nicht vorkommen. Deshalb habe ich dies im K1 genutzt, um TOS direkt als neue Befehlsadresse zu verwenden und damit `EXECUTE` (bei rr=%01 — rettet Rücksprungadresse), `GOTO` und sogar `PERFORM` (ist `@ EXECUTE`) zu implementieren.

Meist wird TOS[1] oder/und NOS[2] für Arithmetik verwendet. K1 verwendet jedoch beim Lesen vom Speicher nicht TOS (enthält die Adresse des Speicherzugriffes), sondern den vom Speicher gelesenen Wert, um z. B. Kombinationen von `@` mit `+` . . . zu erlauben. Multiplikation in Hardware kostet beim MAX10 nur eine DSP–Einheit, eine Zeile Verilog und auch nur einen Takt. Für Division gibt es den `DivStep` in Anlehnung an *NC4000* oder *RTX* und braucht dann ca. 20 Takte. Allerdings wird dabei kein getrenntes MD–Register, sondern TOR[3] als Teiler genutzt. Damit entfallen alle Klimmzüge zur Rettung von Registern. Es gibt auch ein Carry–Flag für Addition, Subtraktion und Shiftbefehle. Dafür nutzt K1 das Bit 0 des PC, das ja nicht gebraucht, aber bei Calls . . . gerettet wird. Es gab nur ein kleines Problem bei Inline–Parametern wie Strings, weshalb es den Befehl `RA` (Returnstack–Adresse) gibt, welcher die Rücksprungadresse aus TOR mit Bit0=0 liefert.

Durch die Umstellung des Befehlssatzes (Abb. 5) konnte ich das freie Bit des J1 (grau in Abb. 1) gut nutzen. Das war mir aber doch zu wenig und deshalb habe ich auch noch eine Sonderbedingung herangezogen. Bei ALU und Speicherbefehlen sind Änderungen der beiden Stackzeiger von -2 bis +1 möglich, wobei -2 beim Datenstack eigentlich nicht sinnvoll genutzt werden kann und deshalb hier missbraucht wurde, um weitere 16 „Spezialbefehle“ zu ermöglichen. Diese ändern die Stacktiefe nicht oder um +1, wenn das Bit 8 (>N) im Befehl gesetzt ist.

	J1 (rr/ss)	K1 (rr)	K1 (ss)
00	---	---	--
01	1	1	1
10	-2	-2	1 (bei >N) oder 0
11	-1	-1	-1

Abbildung 5: Stack–Änderungen

Das SMAF (SmallForth)

Damit kommen wir schon zu dem zugehörigen 16–Bit–Forth, das hier nur im RAM läuft und deshalb keine Besonderheiten für Nutzung von Flash enthält. Die `Words` sind im Anhang wiedergegeben.

Es ist weitestgehend ANS–konform und überwacht die Kontrollstrukturen in `:`–Definitionen. Auf `DO ... LOOP` wurde verzichtet und dafür das sehr schnelle (3 + n Takte) `FOR ... R@ ... NEXT` verwendet, wobei bei `FOR` ein `1- >R` compiliert wird und deshalb die gewünschte Schleifenanzahl anzugeben ist. Bei `0` wäre es dann `65536` mal oder man verwendet `?FOR`, was dies verhindert und die Schleife überspringt. Zusätzlich gibt es die schon häufig beschriebene `CASE ... OF ... ENDOF ... ENDCASE` Struktur.

[1] Top item Of Stack
[2] Next item On Stack
[3] Top Of Returnstack

Mit `FORGET <Name>` kann man die neueren Befehle wieder löschen. Da es keinen getrennten Speicher für Variablen oder Header und nur ein Vokabular gibt, wird einfach `HERE` wieder zurück und der Linkpointer auf das letzte Wort gesetzt.

Mit `'init` und `'abort` gibt es zwei Variablen mit der CFA für Initialisierung und nach Fehlern, hier mit `NOP` vorbelegt. Die Eingabe geht aktuell über den integrierten *UART* mit 115200 Baud. Da kein Filesystem realisiert ist, nutze ich z. B. Teraterm mit Copy/Paste für Eingabe bei 20 ms pro Zeile. Bei einem Fehler werden alle weiteren Zeichen ignoriert, bis 100 ms lang nichts mehr kommt.

Entwicklungsschritte

Eigentlich musste ich mich mit vier Themen befassen:

- K1 mit Speicher–Anbindung (nutzt 9–Kb–RAM–Blöcke)
- Targetcompiler und Assembler für das SmallForth
- Das SmallForth selbst
- K1–Simulator zum Testen des SmallForth

Beginnend mit dem Verilog–Code des J1 (kann man noch teilweise erkennen) ging es an die Umstellung des Befehlssatzes. Viel Kopfzerbrechen hat mir das Speicherinterface bereitet, das jetzt wie die Stacks die 9–Kb–Blöcke des MAX10 nutzt. Bei diesen muss immer bei der positiven Flanke des Taktes die nächste Adresse anstehen. Das ist beim Lesen des Befehlscodes einfach, da die nächste Adresse rechtzeitig berechnet wird. Auch beim Lesen von Daten steht schon die Adresse vor dem eigentlichen Befehl bereit und kann genutzt werden. Aber dass man lesen oder schreiben will, wird erst durch den Befehl ermittelt und es können deshalb erst am Ende des Taktes neue Daten geschrieben oder der Peripherie angezeigt werden, dass etwas gelesen wurde. Dies führt dazu, dass Lesen unmittelbar nach Schreiben nicht geht, was aber normalerweise nicht vorkommt.

Im Betrieb will man auch mit dem Forth kommunizieren und die vorhandenen Schnittstellen nutzen. Darunter fallen die 8 verfügbaren LEDs, ein Taster (neben den LEDs, der zweite auf der anderen Seite ist Reset) und die UART über den FDTI–Chip, der auch für die Programmierung verwendet wird. Dafür verwendet der K1 den Adressbereich ab $FF00, was $-100 entspricht und damit auch durch ein 16–Bit–Literal ohne `INV14` erreichbar ist.

Für erste Tests des Prozessors war der Simulator (Intel ModelSim) für das FPGA sehr nützlich, wenn auch aufwendig und es braucht Zeit. Deshalb habe ich im ersten Schritt auf den ganzen Spaß der Optimierung verzichtet und nur die für SMAF genutzten Befehle getestet.

Damit ich nicht alles mit der Hand codieren muss, folgten dann ein Assembler und ein Targetcompiler — natürlich wie bei mir üblich auch in Forth realisiert. Bei über 35 Jahren Erfahrung darin eigentlich kein Problem, jedoch fällt man trotzdem immer wieder über die eigentlich bekannten Probleme von 16–Bit–Target (K1) auf 32–Bit–System (mein mcForth), High–Endian und Alignment.

Das SMAF gab es schon größtenteils aus einem älteren FPGA–Forth–Projekt und wurde nur angepasst und erweitert. Jedoch läuft sowas garantiert nicht gleich beim ersten Mal und die Simulation auf der mitgelieferten FPGA–Software hat sehr lange Testzyklen und braucht Millionen von Takten, z. B. bei Ausgaben. Deshalb musste auch noch ein eigener Simulator her, der mir die Testzeit auf Sekunden reduziert und Single–Step, Breakpoints oder einfache Debug–Ausgaben erlaubt. Da ich bisher auf Optimierungen verzichtet habe und damit die zu simulierenden Befehle überschaubar sind, war auch dieses Programm relativ schnell realisiert und zeigte mir in kurzer Zeit die noch vorhandenen Probleme. Für die Ein–/Ausgabe verwendete ich die simulierte UART–Schnittstelle.

Erste Tests

Glücklicherweise lief dann die erste Version des SmallForth innerhalb kurzer Zeit auf dem simulierten K1. Auch im FPGA lief es fast sofort und zu meiner Verblüffung sogar mit 100 MHz, obwohl ich mir das Timing überhaupt noch nicht angeschaut hatte. Als ich dann noch einige Features in das FPGA integriert hatte, zeigten sich erste Abstürze, was mich dazu bewog, das Board aktuell mit 48 MHz laufen zu lassen.

Damit habe ich jetzt ein System, mit dem ich arbeiten kann und das viele Möglichkeiten zur Erweiterung bietet. Mittels des angepassten ANS–Testers habe ich den Befehlssatz geprüft und auch den Beschleunigungsaufnehmer kann ich über eine SPI–Schnittstelle schon ansprechen.

Nächste Schritte

Der Befehlssatz des Prozessors bietet enormes Potential zur Optimierung und dieses sollte genutzt werden. Dies bedeutet aber auch erhebliche Arbeit am Target–Assembler und –Compiler. Auch der Simulator sollte dann der vollständigen FPGA–Implementierung entsprechen. Erste Tests zeigen, dass ich dabei >180 Byte von 6 KB spare.

Hardwareseitig werden schon ein 32–Bit–Systemcounter (mit Prozessortakt), UART, LEDs, Taster und die I/O–Ports vom PMOD sowie die SPI–Schnittstelle zum Beschleunigungsaufnehmer unterstützt. Die Schnittstellen zum integrierten Flash und dem 8–MByte–SPI–Flash stehen ganz oben auf der Wunschliste, weil dann Autostart–Anwendungen möglich sind und ein Filesystem wieder interessant wird. Die Schnittstellen zu den Arduino–Nano–Pins und dem im MAX10 integrierten A/D–Wandler sind noch nicht realisiert. Da der J1 für Telespiele genutzt wurde, steht auch auf dem K1 einem Display und einer Joystick–Anbindung nichts im Wege. Die restlichen 9 KByte RAM und mehr als 6 K LE (der K1 braucht aktuell weniger als 20% der 8 K LE) sollten dafür eigentlich ausreichen. Vermutlich werden nur die auf dem MAX1000 verfügbaren 8 MByte SDRAM noch länger warten müssen, da dafür ein entsprechendes Speicherinterface benötigt wird.

Wie gleich am Anfang erwähnt, habe ich das entsprechende FPGA–Projekt, aber auch meine Entwicklungsumgebung für das SmallForth samt den Sourcen des Target–Assemblers/–Compilers, SmallForth, Simulator und auch die Beispielprogramme auf `http://www.mcForth.net` zur privaten Nutzung zur Verfügung gestellt. Als Entwicklungsumgebung verwendete ich das mcForth unter Windows; ebenfalls auf dieser Webseite ausführlich beschrieben. Es gibt entsprechende Batch–Dateien und zusätzliche Beschreibungen, die sollten die Einarbeitung einfacher machen.

Ich würde mich über Anregungen, Hinweise auf Fehler und Verbesserungen, aber natürlich auch über damit realisierte Projekte freuen. Für Fragen erreicht ihr mich unter `kks@designin.de`.

Viel Spaß mit dem Forth–Prozessor K1 und SmallForth.

Anhang: Words

```
init 'init quit find words .s dump ?Abort"
?Abort" ?abort abort 'abort postpone ['] '
name> >name body> >body \IFNDEF \IFDEF \IF
\ELSE \THEN ENDCASE ENDOF OF CASE NEXT ?FOR
FOR BEGIN WHILE UNTIL REPEAT AGAIN ELSE THEN
IF AHEAD recurse Does> Create Variable Constant
; : :noname Header forget indirect restrict
immediate ." s" [char] char Literal ] [ call,
$, , c, align allot $>number? >number . .r
u. u.r 0u.r d. d.r decimal hex #> sign #s #
hold <# \\ \ ( -parse parse refill source
upc accept type cr spaces space ms key key?
emit emit? button? acc_data acc_baud pmod_dir
pmod_data counter_hi counter_lo leds uart_data
uart_state uart_baud 2@ 2! count ! @ c! c@
dabs d- d+ / m/mod um/mod * m* um* nop negate
invert xor or and cells chars flip du2/ d2/
d2* rshift ashift lshift u2/ 2/ 2* within
max min abs u< < <> = 0<> 0= 0< aligned cell+
char+ 2- 1- 2+ 1+ - + goto execute ?exit exit
rdrop ra> r> r@ >r rclear rdepth ?dup -rot rot
2drop 2swap 2over 2dup nip drop swap tuck over
dup dclear depth span >in hld dpl base last
state here voc-link #ramstart #ramend #tib
#c/tib true false bl #msb #bits
```

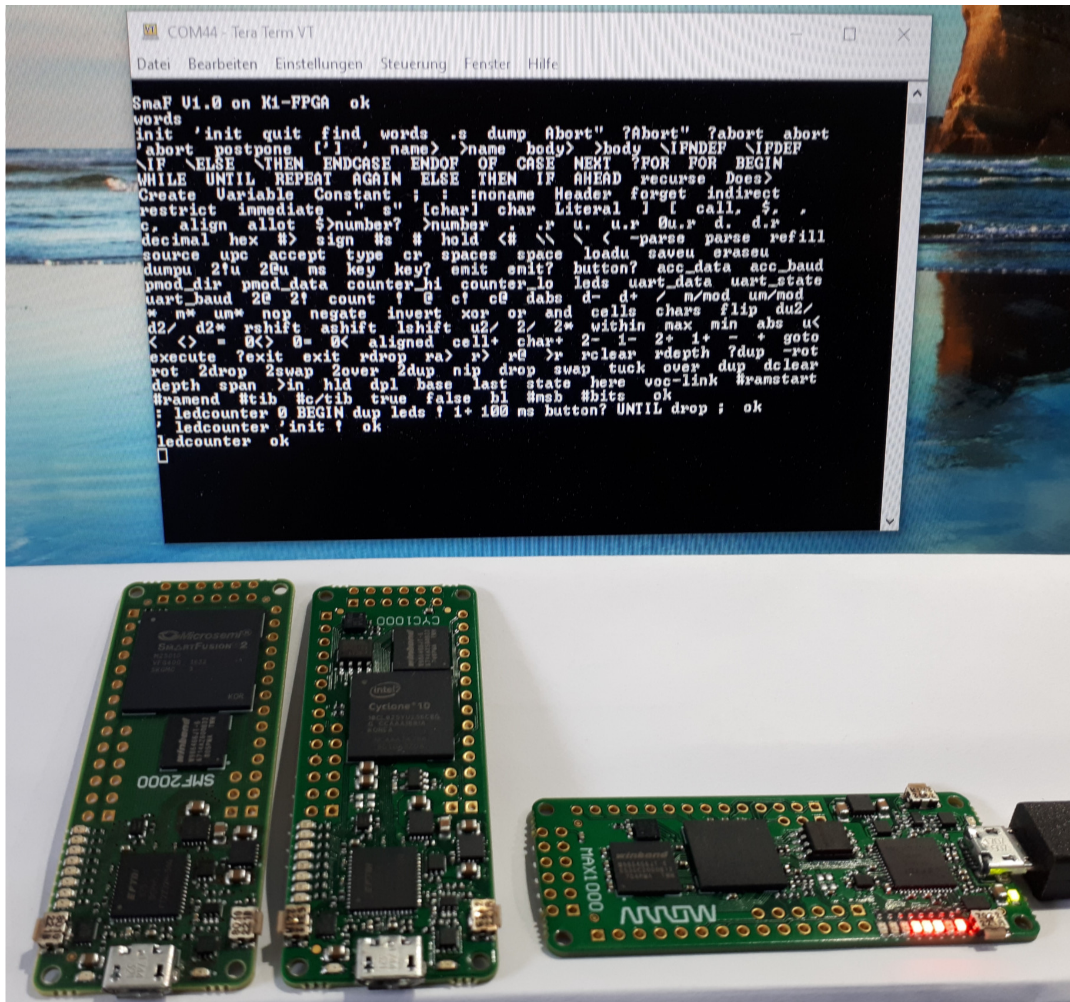


Abbildung 6: Die angesprochenen Boards und ein Terminalfenster mit einem einfachen Programm in SmallForth.

Peripherie für Mecrisp–Ice oder: Ein tiefer Tauchgang in die Welt der FPGAs

Matthias Koch

Außer Neugier sind es hauptsächlich zwei Gründe, einen FPGA zu wählen: Entweder, um einen eigenen Prozessor mit einem maßgeschneiderten Befehlssatz zu entwickeln, oder um ganz besondere Peripherie zu implementieren, die es in der gewünschten Funktionsweise und Geschmacksrichtung in Mikrocontrollern nicht gibt.

Nachdem ich schon mehrere Artikel über die Gestaltung von Befehlssätzen und tolle Ideen für Stackprozessoren gelesen habe, soll es diesmal um die Peripherie gehen, um die vielen kleinen Register, die für den Prozessor das Tor zur Welt sind. Benutzt haben wir sie schon oft und in vielen Varianten, aber wie wird eigentlich ein Ein–Ausgabe–Port implementiert, der auch Interrupts auslösen kann? Wer es schon weiß, der mag nun wissend schmunzeln, und wer weiterliest, der wird es erfahren. Gewürzt wird natürlich mit ein paar Tricks aus dem Nähkästchen. Gedanken zur Gestaltung des Adressraumes für die Peripherie runden diesen Artikel ab.

Eins aber noch vorneweg: Während in Mikrocontrollern meistens eine verwirrende Vielfalt von Konfigurationsbits für alle erdenklichen Anwendungsfälle vorhanden ist, wirkt die in FPGAs verwendete Peripherie oft geradezu primitiv. Natürlich könnten wir uns alle Möglichkeiten offen halten — aber wozu, wenn eine kleine Änderung in der Logik das Gewünschte bringt? Im Prinzip wird hier die Idee von Forth, kleine, einfache und wohldefinierte Bausteine zusammenzufügen, auch auf die „Hardware“ übertragen.

Der Peripherie–Bus in Mecrisp–Ice

Möge der Artikel dort beginnen, wo Forth mit der Außenwelt in Verbindung tritt. In Mecrisp–Ice sieht das „Tor zur Welt“ des Forth–Kernes mit Arbeitsspeicher, den beiden Stacks und dem Prozessor so aus:

```
module j1(
  input wire clk,
  input wire resetq,

  output wire io_rd,
  output wire io_wr,
  output wire [15:0] io_addr,
  output wire [15:0] io_dout,
  input  wire [15:0] io_din,

  input  wire interrupt_request
);
```

Wir haben eine Taktleitung, eine Möglichkeit, den Prozessor neu zu starten, eine Leitung, um einen Interrupt auszulösen, und schließlich einen Bus zur Peripherie. Dieser ist simpel und zweckmäßig aufgebaut: Immer dann, wenn `io_wr` gesetzt ist, sollen die Datenbits aus `io_dout` in das mit `io_addr` ausgewählte Register geschrieben werden. An `io_din` nimmt der Prozessor die Inhalte der ausgewählten Register (ja, es können möglicherweise mehrere zugleich sein) entgegen und teilt der Peripherie durch Setzen von `io_rd` mit, dass er gerade einen Wert liest, falls der Lesezugriff eine Nebenwirkung haben soll.

Bei *Mecrisp–Ice* sind die Peripherie–Zugriffe und deren Adressen separat vom Speicher, aus dem Forth ausgeführt wird. Während auf den Arbeitsspeicher mit `@` und `!` zugegriffen wird, sind für die Peripherie `io@` und `io!` zuständig. Somit sind wir frei, die Adressbits für die Peripherie beliebig zu belegen — aber auch, wenn es getrennte Zugriffe gibt, ist `io_addr` nur dann gültig, wenn `io_wr` oder `io_rd` gesetzt sind, und kann ansonsten beliebige andere Werte enthalten.

Grundausstattung

Zur Grundausstattung von Forth gehört außer einem Taktgeber natürlich ein Terminal. Da sich ein einfacher Timer, der nach einer gewissen Zeit einen Interrupt auslösen kann, immer wieder als nützlich erwiesen hat, ist auch ein „Ticks–Zähler“ stets mit dabei, obwohl er natürlich nicht zwingend erforderlich ist und bei Platzproblemen auch weggelassen werden kann.

Listing 1 zeigt das Grundgerüst — es lohnt sich, dieses einmal gründlich zu durchdenken und zu verstehen. Damit es gelingt, soll das Grundgerüst gleich ein wenig auseinandergenommen werden:

Die Reset–Logik sorgt dafür, dass nach dem Loslaufen des Oszillators erstmal 16 Taktzyklen lang gewartet wird, bevor die Reset–Leitung des Prozessors freigegeben wird. Wie lange der Prozessor im Reset bleiben sollte, kann von Fall zu Fall verschieden sein, am besten so lange, bis der Takt stabil schwingt.

Anschließend werden die Busleitungen definiert und mit dem Prozessor verbunden.

Der Ticks–Zähler, ein ganz einfacher Timer, und das Terminal werden definiert, und am Ende werden die Register, die gelesen werden können, mit einer Art Adressdekoder ausgewählt und allesamt „verodert“ in den Bus eingebunden.

Während `ticks` und die Flags des Terminals einfach so gelesen werden können und nicht auf `io_rd` achten müssen, hat das Lesen von `uart0_data` eine Nebenwirkung, indem der seriellen Schnittstelle mit dem Hilfssignal `uart0_rd` die Abholung des zuletzt empfangenen Bytes mitgeteilt wird.

Somit ergibt sich die Standard–IO–Registerbelegung für Mecrisp–Ice:

$1000: Hier kann das zuletzt empfangene Zeichen abgeholt oder ein neues Zeichen versendet werden.

$2000: Flags für `key?` und `emit?` .

$4000: Lesen gibt den aktuellen Zählerstand zurück, Schreiben setzt den Zähler auf den gewünschten Wert.

Leuchtdioden leuchten lassen!

Nun soll ein ganz einfaches Register eingefügt werden, das eine Handvoll Leuchtdioden ein– und ausschalten kann:

```
reg [15:0] LEDs = 0;

always @(posedge clk)
  if (io_wr & io_addr[15]) LEDs <= io_dout;

assign io_din =
  ...
  (io_addr[15] ? LEDs : 16'd0) |
  ...
```

Immer dann, wenn das richtige Adressbit gesetzt ist, wird der Inhalt von `LEDs` an den Prozessor übergeben (übrigens in jedem Taktzyklus, der Prozessor sucht sich selber aus, ob er `io_din` gerade braucht oder nicht) und wenn außerdem ein IO–Schreibzugriff gewünscht ist, wird das Register verändert.

Und schon können die angelöteten Leuchtdioden mit dem Inhalt des Registers `$8000` angesteuert werden, zum Beispiel so:

```
assign {Blau, Gruen, Rot} = LEDs[2:0];
```

Einfache Register, bei denen Lesen und Schreiben keine weiteren Nebenwirkungen haben sollen, sehen immer so aus und können für alles Mögliche verwendet werden.

Eine Besonderheit gibt es noch zu beachten: Normalerweise sind wir es gewöhnt, in Bytes zu denken — hier haben wir es aber mit Word–Adressierung zu tun. Das bedeutet konkret: Wenn an Adresse `$8000` ein 16 Bit breites Register für die Leuchtdioden ist, ist an Adresse `$8001` nicht etwa das High–Byte desselben Registers zu finden, sondern ein ganz anderes Register, das wiederum 16 Bits breit sein kann. Sollten wir die verfügbaren Adressbits voll ausnutzen, könnten also theoretisch 128 KB an IO–Registern untergebracht werden.

Einfache Ein– und Ausgabepins

Ein einfacher Ein–Ausgabe–Port braucht drei Register: ein Eingaberegister, das den aktuellen elektrischen Zustand der Pins wiederspiegelt, ein Ausgaberegister, worin die Ausgänge zwischen Low und High gewechselt werden können, und ein Richtungsregister, worin gewählt wird, ob ein Pin ein Eingang oder ein Ausgang sein soll:

```
wire [15:0] port_in;
reg  [15:0] port_out;
reg  [15:0] port_dir;
```

Wie genau die Bits an die als „inout“ deklarierten Pins angeschlossen werden, hängt vom FPGA ab. Bei der iCE40–Familie sieht es so aus:

```
SB_IO #(.PIN_TYPE(6'b1010_01)) io0 (
  .PACKAGE_PIN(PORT0),
  .D_OUT_0(port_out[0]),
  .D_IN_0(port_in[0]),
  .OUTPUT_ENABLE(port_dir[0])
);
```

Und beim ECP5 schließlich so:

```
BB io0 (
  .B(PORT0),
  .I(port_out[0]),
  .O(port_in[0]),
  .T(~port_dir[0])
);
```

Es gibt auch eine allgemeine Lösung, die insbesondere beim Portieren auf neue FPGA–Familien nützlich sein kann:

```
assign port_in[0] = PORT0;
assign PORT0 = port_dir[0] ? port_out[0] : 1'bz;
```

Der Unterschied spielt aber fürs Verständnis keine große Rolle und kann aus den Datenblättern oder den vorhandenen Quelltexten in Mecrisp–Ice entnommen werden.

Nun wollen wir uns mal anschauen, wie die drei Register eingebunden werden. Zum Lesen werden einfach die entsprechenden Bits in den Bus „verodert“, wenn das entsprechende Adressbit gesetzt ist:

```
assign io_din =
  ...
  (io_addr[4] ? port_in  : 16'd0) |
  (io_addr[5] ? port_out : 16'd0) |
  (io_addr[6] ? port_dir : 16'd0) |
  ...
```

und auch das Schreiben in die Register ist keine Zauberei:

```
always @(posedge clk)
begin
  if (io_wr & io_addr[5])  port_out <= io_dout;
  if (io_wr & io_addr[6])  port_dir <= io_dout;
end
```

Und schon ist `port_in` an der Adresse `$0010`, `port_out` an `$0020` und `port_dir` an `$0040` ansprechbar.

Manche werden nun fragen, wieso `port_in` und `port_out` nicht auf dem gleichen Adressbit zusammengelegt werden — die Antwort ist einfach: Es ist gut möglich, dass der gewünschte Ausgabezustand eines Ausgangs–Pins nicht mit dem tatsächlichen elektrischen Zustand übereinstimmt, nämlich dann, wenn eine große Kapazität getrieben wird oder falls der Pin aus Versehen kurzgeschlossen wurde.

Gedanken zur Aufteilung des Adressraumes

Eine ganz wesentliche Frage ist, wie die Register im IO–Adressraum untergebracht werden sollen. Bei Mecrisp–Ice sind es 16 Adressbits, die zum Einsatz kommen können, und in FPGAs sind wir frei zu implementieren, was immer wir möchten.

Wer sich bislang mit Mikrocontrollern beschäftigt hat, wird damit vertraut sein, dass jedes Register unter genau einer Adresse ansprechbar ist. In Verilog sieht das so aus: `(io_wr & (io_addr == 16'h0020))`

Damit lassen sich sehr viele Register unterbringen und die Softwareseite fühlt sich für viele an „wie immer". Der Nachteil ist, dass die Vergleiche natürlich auch Gatter erfordern und der Bedarf für die Dekodierlogik beträchtlich sein kann.

Wenn es nur wenige Register sind, so ist die einfachste und auch in punkto Gatterbedarf sparsamste Lösung, einfach jedem Register ein eigenes Adressbit zuzuordnen, in der Art: `(io_wr & (io_addr[5]))`

Die Besonderheit dieser Konfiguration ist, dass so mehrere oder nach Wunsch auch alle Register zugleich angesprochen werden können — und beim Lesen, durch die Oder–Verknüpfung zwischen den Datenbits aus verschiedenen Quellen — gesetzte Bits in mehreren Registern zugleich detektiert werden können.

So, wie der Forth–Kern von Mecrisp–Ice implementiert ist, werden im Grundgerüst (Listing 1) eigentlich nur zwei Register vorausgesetzt und ein drittes gewünscht: An `$1000` sei das Datenregister des Terminals zu finden, an `$2000` das Flagregister des Terminals, und — nützlich, aber optional — an `$4000` der Timer, der auch Interrupts auslösen kann und auf den ich noch einmal zurückkommen möchte. Diese Adressen habe ich so gewählt, dass sie in beiden Varianten der Adressdekodierung gleichermaßen gut verwendet werden können.

Doch insbesondere Mischformen können interessant sein, mit teilweiser Dekodierung und gemeinsamen „Modifikationsbits".

Besonders praktische Ein– und Ausgabepins

Häufig kommt es vor, dass in einem Register einzelne Bits gesetzt, gelöscht oder umgeschaltet werden sollen, ohne dabei die anderen Bits zu beeinflussen. Damit dies gelingt, wenn Interrupts ins Spiel kommen oder es einfach sehr schnell gehen muss, gibt es in Listing 2 eine schöne Lösung, die mir zum ersten Mal in den *PIC32MX–Mikrocontrollern* begegnet ist. Die untersten beiden Adressbits geben an, ob es ein direkter Schreibzugriff sein soll (`+0`), oder ob Bits gelöscht (`+1`), gesetzt (`+2`) oder gekippt (`+3`) werden sollen.

Die dazugehörigen Registerkonstanten in einem Forth–Programm könnten so aussehen:

```
$0010 constant Port-In

$0020 constant Port-Out
$0021 constant Port-Out-Clear
$0022 constant Port-Out-Set
$0023 constant Port-Out-Toggle

$0040 constant Port-Dir
$0041 constant Port-Dir-Clear
$0042 constant Port-Dir-Set
$0043 constant Port-Dir-Toggle
```

So etwas ist sehr, sehr praktisch — ich kann nur empfehlen, zwei Bits des IO–Adressraumes dafür zu reservieren und diese Konvention durchgängig bei allen Registern, wo der Zugriff auf Einzelbits nützlich ist, zu verwenden.

Wer übrigens von `Port-Out-Clear` liest, wird das gleiche Ergebnis wie von `Port-Out` bekommen, weil beim Lesen die beiden dafür zuständigen Adressbits nicht dekodiert werden. Es lassen sich aus den Adressbits interessante Kombinationen bauen — aber bitte mit Überlegung und Verstand, es sei denn, es soll ein undurchdringliches Registerlabyrinth entstehen, was während der ersten Erkundungen, wo schnell noch dies und das hinzugefügt wird, leicht passiert.

Der Ticks–Zähler im Detail

Wer ist nicht schon einmal über eine komplizierte Timer–Konfiguration gestolpert ... Aber hier soll es ganz einfach zur Sache gehen, versprochen! Der Ticks–Zähler ist ein ganz einfacher Timer, der bei jedem Taktzyklus einen Schritt weiterzählt, auf einen beliebigen Wert gesetzt werden kann und der bei Überlauf einen Interrupt auslöst. Auch wenn er zum Grundgerüst gehört, sollten die entsprechenden Zeilen der Übersichtlichkeit halber hier noch einmal erscheinen:

```
reg [15:0] ticks;

wire [16:0] ticks_plus_1 = ticks + 1;

always @(posedge clk)
  if (io_wr & io_addr[14])
    ticks <= io_dout;
  else
    ticks <= ticks_plus_1;

always @(posedge clk)
  interrupt <= ticks_plus_1[16];
```

Zunächst wird ein Register gebraucht, in dem gezählt wird: `ticks`. Dieses wird immer dann inkrementiert, wenn gerade kein IO–Schreibzugriff auf Adressbit 14 (`$4000`) vorliegt. Bei einem Überlauf von `$FFFF` nach `$0000` wird über das oberste Bit in `ticks_plus_1`, quasi dem Carry–Flag des Zählers, ein Interrupt signalisiert. Bei einem Schreibzugriff wird `ticks` nicht inkrementiert, sondern auf den vom Prozessor gewünschten Wert gesetzt, wobei allerdings bei dieser Implementierung dennoch ein Interrupt ausgelöst wird, falls der Schreibzugriff genau dann stattfindet, wenn `ticks` gerade von selbst bei `$FFFF` gewesen ist. Wird der Timer auf Null gesetzt, wird dabei

kein Interrupt ausgelöst, allerdings führt ein Setzen auf `$FFFF` zu einem Interrupt im nächsten Taktzyklus.

Gelesen wird der Timer wie ein ganz normales Register, indem `ticks` bei passend gesetztem Adressbit in `io_din` „verodert“ wird.

```
assign io_din =
  ...
  (io_addr[14] ? ticks : 16'd0) |
  ...
```

Zu beachten ist, dass das Interrupt–Bit nur für je einen einzigen Taktzyklus erscheint. Wenn der Prozessor der Einfachheit halber nur eine Interruptquelle hat und die einzelnen Auslösungen des Timer–Interrupts zeitlich nicht zu dicht beieinander liegen, ist das jedoch kein Problem, da jeder Befehl genau einen Taktzyklus dauert. Ansonsten wäre noch ein bisschen mehr Mühe nötig, die Interrupt–Bits zu sammeln, bis sie abgearbeitet werden können.

Freilaufend erzeugt dieser Timer Interrupts mit $12MHz/2^{16} = 183.11Hz$, was natürlich mit der gewählten Taktfrequenz variiert. Aber auch schnellere periodische Interrupts lassen sich mit einem kleinen Software–Trick implementieren:

```
: ticks ( -- u ) $4000 io@ ;

\ Trigger next irq u cycles after the last one.
: nextirq ( cycles -- )
  ticks  \ Read current tick
  -       \ Subtract the cycles already elapsed
  4 -      \ Cycles neccessary to do this
  negate    \ Timer counts up to zero to trigger
  $4000 io!  \ Prepare timer for next interrupt
;
```

Wird `nextirq` an beliebiger Stelle im Interrupthandler aufgerufen, lässt sich eine exakte Anzahl von Taktzyklen nach dem aktuellen Interrupt der nächste auslösen. So ist es bereits mit ganz einfachen Mitteln möglich, ein präzises Timing zu halten — ein gutes Beispiel ist der Bit–Bang–VGA–Grafiktreiber auf dem *Nandland Go*.

Dem weiteren Ausbau sind keine Grenzen gesetzt — wie wäre es mit einem Inkrement–Register, das nicht immer nur 1 sein muss? Oder einer einstellbaren Grenze für den Überlauf? Dabei geschieht aber vom Prinzip her nichts Besonderes, so dass ich hier nur die grundlegende Variante zeigen möchte.

Raffinierte Ein– und Ausgabepins mit Interrupts

Mein absoluter Traum ist schließlich im *ULX3S* erreicht, wo die zusätzlich mit Interrupts ausgestatteten IO–Pins folgenden an den *MSP430* angelehnten Registersatz haben:

```
 Addr  Bit  Read  Write
------------------------

+ ...0          Write as usual
+ ...1          _C_lear bits
+ ...2          _S_et bits
+ ...3          _T_oggle bits

0004  2   IN               Input
0008  3   OUT   OUT (cst)  Output
0010  4   DIR   DIR (cst)  Direction
0020  5   IFG   IFG (cst)  Interrupt Flag
0040  6   IES   IES (cst)  Interrupt Edge Select
0080  7   IE    IE  (cst)  Interrupt Enable

0100  8   --- PORT1 --->
0200  9   --- PORT2 --->
0400  10  --- PORT3 --->
0800  11  --- PORT4 --->
```

Ein kleines „Rätsel“: Was bewirken diese Schreibzugriffe?[1]

```
 0 $0FF8 io! ( Port 1,2,3,4 OUT,DIR,IFG,IES,IE )
64 $0382 io! ( Port 1, 2 OUT Set )
 1 $0821 io! ( Port 4 IFG Clear )
```

Was für eine Funktion die Register `IN`, `OUT` und `DIR` erfüllen, wurde vorhin schon besprochen, doch bevor es in die Tiefe der Implementierung geht, soll an dieser Stelle erst einmal die Funktionsweise der Interrupt–Register `IFG`, `IES` und `IE` beleuchtet werden, die von dem Registersatz der Ein–Ausgabe–Ports des MSP430 inspiriert sind:

Wenn Interrupts für einen Pin durch ein gesetztes Bit in `IE` aktiviert sind, so wird je nach Inhalt von `IES` bei einer fallenden (`0`) oder steigenden (`1`) Flanke ein Bit in `IFG` gesetzt. Wenn mindestens eins der Bits in `IFG` gesetzt ist, wird ein Interrupt an den Prozessor gemeldet, der dann in `IFG` prüfen kann, an welchen Pins Interrupts aufgetreten sind und anschließend das entsprechende Bit in `IFG` löschen muss, bevor der Interrupthandler zurückkehrt und/oder Interrupts mit `eint` wieder aktiv werden.

Zuallererst benötigen wir also eine Erkennung der Flanken. In der einfachen Variante wurden die Pins direkt an den Prozessorbus übergeben — und sollte ein Pin einmal genau während der Taktflanke „wackeln“, ist das nicht so schlimm, weil die Signale im Prozessor selbst durch Flipflops laufen, bevor sie im Stack gespeichert werden. So mussten wir uns bislang über die Synchronisierung wenig Gedanken machen — wenn aber Flanken fürs Auslösen von Interrupts sicher erkannt werden sollen, dann ist es wichtig, die Signale vorher zu synchronisieren (siehe Listing 3, Zeilen 1 bis 11).

Das Modul `omsp_sync_cell` habe ich direkt aus dem *openMSP430* von Olivier Girard übernommen — schließlich ist es ja der Aufbau der *MSP430–GPIOs*, den ich hier mit kleinen Anpassungen übernommen habe. In `omsp_sync_cell` sind einfach nur zwei Flipflops hintereinander, die dafür sorgen, dass keine metastabilen

[1] Auflösung: a) Alles zurücksetzen, Pins Eingänge. b) Bit 6 in P1OUT und P2OUT setzen. c) Interrupt für P4.0 löschen.

Zustände auftreten und wir so ein „sauberes“ Signal in unsere Logik hereinbekommen. Außerdem hat es noch eine Nebenwirkung: Was außen elektrisch geschieht, sehen wir innendrin durch die Synchronisierung erst zwei Taktzyklen später.

Nun müssen die Flanken in den synchronisierten Leitungen erkannt werden. Dafür wird der letzte bekannte Zustand der Pins festgehalten und mit dem aktuellen Zustand verglichen (Listing 3, Zeilen 13 bis 17).

Wurde eine Flanke erkannt und ist diese — wie gewünscht — entweder fallend oder steigend, so wird vermerkt, dass die Bedingung für einen Interrupt erfüllt wäre (Listing 3, Zeilen 19 bis 23).

Wir können aber nicht einfach so die zu setzenden Bits in das IFG–Register schreiben, weil auch der Prozessor schreibend darauf zugreifen kann. Um keine Interrupts zu verlieren, sollen neue Interrupts außerdem unbedingt auch bei Schreibzugriffen des Prozessors auf dieses Register gesammelt werden. Deshalb ist die Logik zum Schreiben auf das IFG–Register ein bisschen komplizierter, aber zum Glück immer noch gut überschaubar (Listing 3, Zeilen 25 bis 37).

Es gibt wieder die Möglichkeit, den Inhalt zu schreiben, Bits zu löschen, zu setzen oder umzukippen, wobei neue Interrupts auf jeden Fall mit aufgenommen werden.

Jetzt haben wir alles beisammen, um alle aktiven Pin–Interrupts an den Prozessor zu melden (Listing 3, Zeile 39).

Im Gegensatz zum Interrupt des einfachen Tick–Zählers bleibt dieser Interrupt so lange gesetzt, bis die entsprechenden Bits in dem IFG–Register wieder zurückgesetzt werden. Soll so ein Port gemeinsam mit dem Tick–Zähler verwendet werden, so muss dessen nur ein Taktzyklus lang anhaltendes Interrupt–Bit noch gepuffert werden, da es sonst verloren gehen würde, wenn ein Tick–Interrupt gerade dann anstehen sollte, wenn der Prozessor gerade einen anderen Interrupt abarbeitet.

Zufallszahlen und Ringoszillatoren

Wenn der Ausgang eines Inverters mit dessen Eingang verbunden wird, entsteht ein Oszillator, also eine Schaltung, die nie einen stabilen Zustand erreicht. Was in FPGAs meistens vermieden werden soll, kann ganz praktische Anwendungen haben, zum Beispiel als Zufallszahlengenerator:

```
wire [1:0] buffers_in, buffers_out;
assign buffers_in =
      {buffers_out[0:0], ~buffers_out[1]};

SB_LUT4 #(
        .LUT_INIT(16'd2)
) buffers [1:0] (
        .O(buffers_out),
        .I0(buffers_in),
        .I1(1'b0),
        .I2(1'b0),
        .I3(1'b0)
);

wire random = ~buffers_out[1];
```

Das sieht jetzt erstmal ein bisschen komisch aus, aber es wird verständlich, wenn in Betracht gezogen wird, dass die Werkzeuge für die FPGAs so viel wie möglich wegoptimieren wollen. Hier werden explizit zwei Puffer verwendet, um den Ausgang eines Inverters mit einer kleinen Verzögerung durch die Laufzeit mit dessen Eingang zu verbinden.

Das Ergebnis ist ein sehr, sehr schneller Oszillator, der sich relativ zum Quarztakt chaotisch verhält und empfindlich von der Temperatur, der Versorgungsspannung und der tatsächlichen Platzierung im FPGA abhängt. Die Leitung `random` muss dann nur noch irgendwo lesbar eingebunden werden, und schon stehen Zufallszahlen zur Verfügung. Wichtig ist nur, zwischen dem Lesen der einzelnen Bits dem Ringoszillator ein kleines bisschen Zeit zum „Wegdriften“ zu geben, weil ansonsten bei dicht aneinander liegenden Lesezugriffen Korrelationen zwischen den einzelnen Zufallsbits auftreten.

Mit einer größeren Laufzeit durch die „Verzögerungsleitung“ zwischen Ausgang und Eingang des Inverters wird die Frequenz niedriger. Dies sei hier zu einem praktischen Modul zusammengefasst:

```
module ring_osc ( input resetq, output osc_out );

  parameter NUM_LUTS = 42;

  wire [NUM_LUTS:0] buffers_in, buffers_out;
  assign buffers_in =
        {buffers_out[NUM_LUTS - 1:0], chain_in};
  wire chain_out = buffers_out[NUM_LUTS];
  wire chain_in = resetq ? !chain_out : 0;

  SB_LUT4 #(
      .LUT_INIT(16'd2)
  ) buffers [NUM_LUTS:0] (
      .O(buffers_out),
      .I0(buffers_in),
      .I1(1'b0),
      .I2(1'b0),
      .I3(1'b0)
  );

  assign osc_out = chain_out;

endmodule
```

Mit dem Reset–Eingang lässt sich der Ringoszillator ein- und ausschalten, was insbesondere beim ersten Anschwingen nötig ist, da ein beliebiger anfänglicher elektrischer Zustand in den Puffern im schlimmsten Falle zu einem chaotischen Schwingverhalten führen könnte.

Dadurch, dass die Frequenz eines Ringoszillators auch (aber nicht nur) von der Temperatur abhängt, kann somit, wenn ein Frequenzzähler programmiert wird, auch ein

Temperaturfühler in den FPGA gezaubert werden, der aber nicht sonderlich genau ist und nach jedem Kompilieren des Verilog–Quelltextes und für jeden individuellen FPGA neu kalibriert werden muss. Ein Ringoszillator ist aber prima dafür geeignet, mal schnell herauszufinden, ob einer Schaltung zwischendurch heiß wird oder ob sie im Einsatz großen Temperaturschwankungen ausgesetzt ist.

Flackern, Dimmen, Schwingen

Pulsbreitenmodulation ist ganz einfach zu implementieren, wobei sich die Wiederholrate aus dem Takt und der Breite der Register ergibt.

```
reg [15:0] pwm;
reg [15:0] pwm_counter;

always @(posedge clk)
  pwm_counter <= pwm_counter + 1;

wire pwm_out = pwm_counter < pwm;
```

Das Register `pwm` wird in den Prozessor eingebunden. Solange der freilaufende Zähler `pwm_counter` kleiner ist als der Steuerwert in `pwm`, sei der Ausgang gesetzt, ansonsten nicht. Es funktioniert, hat aber zwei unschöne Eigenschaften: Wenn der Steuerwert in `pwm` geändert wird, kann der Ausgang plötzlich springen, und — falls `pwm` nicht gerade Null ist — wird der Ausgang nur genau zweimal pro Zählerdurchlauf geschaltet, egal, welcher Wert gerade ausgegeben wird. Das bedeutet, dass ein RC–Tiefpassfilter mit relativ großer Zeitkonstante zum Glätten des Signales erforderlich ist, falls ein analoges Signal ausgegeben werden soll.

Es gibt aber eine viel schönere Lösung, die nur unwesentlich komplizierter ist, diese beiden Unarten nicht zeigt und sich Sigma–Delta–Modulator nennt:

```
reg  [15:0] control;
reg  [15:0] phase;

wire [16:0] phase_new = phase + control;

always @(posedge clk)
  phase <= phase_new[15:0];

wire sdm_out = phase_new[16];
```

Ganz wichtig ist, dass `phase_new` um ein Bit breiter ist als die anderen beiden Register. Immer dann, wenn das Register `phase` beim Summieren überläuft, gibt es für einen Taktzyklus lang einen Puls am Ausgang. Je größer der Steuerwert `control` ist, desto schneller läuft `phase` über und umso früher erscheint der nächste Puls am Ausgang. Dadurch, dass viele schmale Pulse ausgegeben werden, sind die Ansprüche an den analogen Tiefpassfilter viel geringer — es kann also schneller geregelt werden, und es gibt auch bei einem plötzlichen Wechsel des Steuerwertes keine Artefakte.

Mit einer ganz kleinen Variation, nämlich mit einem zusätzlichen Bit im Phasenregister, wird ein numerisch gesteuerter Oszillator (*Numerical Controlled Oscillator, NCO*) daraus, der mit feinen Stufen jede beliebige Frequenz zwischen 0 und (fast) der halben Taktfrequenz ausgeben kann:

```
reg [15:0] nco_freq;
reg [16:0] nco_phase;

always @(posedge clk)
  nco_phase <= nco_phase + nco_freq;

wire nco_out = nco_phase[16];
```

Wenn das Register `nco_freq` auf `0` steht, ist der Oszillator aus, ansonsten schwingt er mit:

$$f_{nco} = nco_freq * f_{clk}/2^{17}$$

Die größtmögliche Frequenz ist somit:

$$f_{clk} * \$FFFF/2^{17} = f_{clk} * 65535/2^{17}$$

oder $f_{clk} * 0,499992370605$.

Schwingen mit mehr Schwung

Wer möchte, dass der numerische Oszillator bis (fast) zur eigenen Taktfrequenz hin durchgestimmt werden kann, kommt bestimmt auf die Idee, etwas wie `always @(posedge clk or negedge clk)` zu verwenden, wird dann aber feststellen, dass die Konstruktion nicht synthetisiert werden kann, weil es im FPGA keine Flipflops gibt, die auf beiden Flanken des Taktsignales neue Daten übernehmen können. Dafür gibt es jedoch einen Trick:

```
reg [15:0] nco_freq;
reg [16:0] nco_phase;

wire [16:0] nco_next = nco_phase + nco_freq;

DEFF _nco [16:0] (
  .clock(clk),
  .in(nco_next[16:0]),
  .out(nco_phase[16:0]),
  .resetq(resetq)
);

wire nco_out = nco_phase[16];
```

Wobei besondere Flipflops zum Einsatz kommen, die auf beiden Flanken des Taktes schalten:

```
module DEFF(
  input  clock, resetq, in,
  output out
);

  reg trig1, trig2;

  assign out = trig1^trig2;

  always @(posedge clock, negedge resetq)
  begin
```

```
    if (~resetq)  trig1 <= 0;
    else  trig1 <= in^trig2;
  end

  always @(negedge clock, negedge resetq)
  begin
    if (~resetq)  trig2 <= 0;
    else  trig2 <= in^trig1;
  end
endmodule
```

DEFF bedeutet „Dual Edge Flip Flop", und das Modul liegt Mecrisp–Ice bei. Es funktioniert tadellos, da stets nur einer der beiden mit XOR vereinten Flip–Flop–Ausgänge umschaltet, nie beide zugleich.

Zu beachten ist allerdings, dass dieser Trick die Logik insgesamt nicht „schneller" macht — die maximale Geschwindigkeit hängt davon ab, wie viele Gatter, Flipflops und Verbindungen in einer langen Kette hintereinanderhängen, die innerhalb eines Taktzyklus vollständig durchlaufen werden muss. Je länger diese ist, desto langsamer wird der Gesamtentwurf maximal laufen. Es ist aber praktisch, wenn der Takt des Prozessors aus anderen Gründen festgelegt ist und kein zweiter Takt verwendet werden soll.

Ausblick

Hoffentlich hat dieser kleine Einblick in die Peripherie gezeigt, was eine der großen Stärken eines FPGAs im Vergleich zu einem Mikrocontroller ist: die große Flexibilität bei der Gestaltung und Wahl der Peripherie. Vor allem, wer präzises Timing benötigt, wird FPGAs sehr zu schätzen wissen.

Dem neugierigen Entdecker sei zum Schluss noch ein guter Tipp mit auf den Weg gegeben: Es ist eine gute Balance aus Software und Logik zu finden.

Manche Funktionen lassen sich ganz einfach vollständig in Logik implementieren, andere lassen sich gut mit ein wenig unterstützender Logik vom Prozessor erledigen, und schließlich gibt es Fälle, insbesondere beim ersten Kennenlernen, wo es am praktischsten ist, einfach nur ein paar Ein–Ausgabe–Leitungen zu definieren und den Prozessor an den Leitungen wackeln zu lassen. Manchmal hängt es auch davon ab, welches Ziel erreicht werden soll: Mein persönliches Steckenpferd „Ledcomm"[2], die Kommunikation über eine Leuchtdiode, lässt sich zum Experimentieren gut in Software implementieren, Mecrisp–Ice liegt aber auch eine Variante in Logik bei, die anstelle der seriellen Schnittstelle direkt ins Grundgerüst eingefügt werden kann und so den Prozessor für andere Aufgaben frei hält.

Listing 1: Grundgerüst

```
1  `default_nettype none
2
3  `define cfg_divider 104 // 12 MHz/115200 = 104.17
4
5  `include "../common-verilog/uart.v"
6  `include "../common-verilog/j1-universal-16kb.v"
7
8  module top(input  clk,
9             output TXD,
10            input  RXD,
11            input  reset_button
12 );
13
14   // ######   Resetlogik   ######################
15
16   reg [3:0] reset_cnt = 0;
17   wire resetq = &reset_cnt;
18
19   always @(posedge clk) begin
20     if (reset_button)
21          reset_cnt <= reset_cnt + !resetq;
22     else reset_cnt <= 0;
23   end
24
25   // ######   Bus   #############################
26
27   wire io_rd, io_wr;
28   wire [15:0] io_addr;
29   wire [15:0] io_dout;
30   wire [15:0] io_din;
31
32   reg interrupt = 0;
33
34   // ######   Prozessor   #######################
35
36   j1 _j1(
37     .clk(clk),
38     .resetq(resetq),
39
40     .io_rd(io_rd),
41     .io_wr(io_wr),
42     .io_dout(io_dout),
43     .io_din(io_din),
44     .io_addr(io_addr),
45
46     .interrupt_request(interrupt)
47   );
48
49   // ######   Ticks-Zähler   ####################
50
51   reg [15:0] ticks;
52
53   wire [16:0] ticks_plus_1 = ticks + 1;
54
55   always @(posedge clk)
56     if (io_wr & io_addr[14])
57       ticks <= io_dout;
58     else
59       ticks <= ticks_plus_1;
60
61   // Generate interrupt on ticks overflow
62   always @(posedge clk)
63     interrupt <= ticks_plus_1[16];
64
65   // ######   Terminal   ########################
66
67   wire uart0_valid, uart0_busy;
68   wire [7:0] uart0_data;
69   wire uart0_wr = io_wr & io_addr[12];
70   wire uart0_rd = io_rd & io_addr[12];
71
72   buart _uart0 (
73      .clk(clk),
74      .resetq(resetq),
75      .rx(RXD),
```

[2] „Ledcomm — den Glühwürmchen nachempfundene Kommunikation zwischen zwei Leuchtdioden", Matthias Koch, Vierte Dimension — Das Forth–Magazin, 29. Jahrgang, Ausg. 2/2013, Seite 13 ff.

```
76        .tx(TXD),
77        .rd(uart0_rd),
78        .wr(uart0_wr),
79        .valid(uart0_valid),
80        .busy(uart0_busy),
81        .tx_data(io_dout[7:0]),
82        .rx_data(uart0_data));
83
84    // ######   IO-Register   ######################
85
86    assign io_din =
87
88    (io_addr[12] ?
89      {  8'd0,               uart0_data } : 16'd0) |
90    (io_addr[13] ?
91      { 14'd0, uart0_valid, !uart0_busy } : 16'd0) |
92    (io_addr[14] ?                     ticks    : 16'd0) ;
93
94  endmodule
```

Listing 2: Ein- und Ausgabe mit „Clear, Set, Toggle"

```
1   if (io_wr & io_addr[5] & (io_addr[1:0] == 0))  port_out  <=              io_dout;
2   if (io_wr & io_addr[5] & (io_addr[1:0] == 1))  port_out  <=  port_out & ~io_dout; // Clear
3   if (io_wr & io_addr[5] & (io_addr[1:0] == 2))  port_out  <=  port_out |  io_dout; // Set
4   if (io_wr & io_addr[5] & (io_addr[1:0] == 3))  port_out  <=  port_out ^  io_dout; // Toggle
5
6   if (io_wr & io_addr[6] & (io_addr[1:0] == 0))  port_dir  <=              io_dout;
7   if (io_wr & io_addr[6] & (io_addr[1:0] == 1))  port_dir  <=  port_dir & ~io_dout; // Clear
8   if (io_wr & io_addr[6] & (io_addr[1:0] == 2))  port_dir  <=  port_dir |  io_dout; // Set
9   if (io_wr & io_addr[6] & (io_addr[1:0] == 3))  port_dir  <=  port_dir ^  io_dout; // Toggle
10
11  assign io_din =
12    ...
13    (io_addr[4] ? port_in  : 16'd0) |
14    (io_addr[5] ? port_out : 16'd0) |
15    (io_addr[6] ? port_dir : 16'd0) |
16    ...
```

Listing 3: Ein- und Ausgabe mit Interrupts

```
1   reg  [15:0] porta_dir;
2   reg  [15:0] porta_out;
3   wire [15:0] porta_in_async;
4
5   wire [15:0] porta_in;
6   omsp_sync_cell porta_synchronisers [15:0] (
7     .data_in(porta_in_async),
8     .data_out(porta_in),
9     .clk(clk),
10    .rst(!resetq)
11  );
12
13  reg [15:0] porta_in_delay;
14  always @(posedge clk) porta_in_delay <= porta_in;
15
16  wire [15:0] porta_in_re =  porta_in & ~porta_in_delay;  // Rising Edge
17  wire [15:0] porta_in_fe = ~porta_in &  porta_in_delay;  // Falling Edge
18
19  reg [15:0] porta_ifg;
20  reg [15:0] porta_ies;
21  reg [15:0] porta_ie;
22
23  wire [15:0] porta_ifg_set = (porta_ies & porta_in_fe) | (~porta_ies & porta_in_re);
24
25  always @(posedge clk)
26  begin
27    if (io_wr & io_addr[8] & io_addr[5])
28    begin
29      casez (io_addr[1:0])
30        2'd0: porta_ifg <= (             io_dout) | porta_ifg_set;
31        2'd1: porta_ifg <= (porta_ifg & ~io_dout) | porta_ifg_set; // Clear
32        2'd2: porta_ifg <= (porta_ifg |  io_dout) | porta_ifg_set; // Set
33        2'd3: porta_ifg <= (porta_ifg ^  io_dout) | porta_ifg_set; // Toggle
34      endcase
35    end
36    else porta_ifg <= (porta_ifg | porta_ifg_set);
37  end
38
39  wire irq_porta = |(porta_ie & porta_ifg);
```

Natürliche Sprachen und Forth

Jens Storjohann

Ein Forth–Programmierer, und auch ein Nutzer einer anderen Programmiersprache, kann oft eine Aufgabe dadurch lösen, dass er eine an das Problem angepasste Sprache entwickelt. Auch kann die Aufgabe gestellt sein, einen Dialog oder Fragebögen zu erstellen, die in andere natürliche Sprachen, möglichst automatisch, übersetzbar sind. Für beide Aufgaben kann es nützlich sein, etwas über die Eigenschaften und Konstruktionen natürlicher Sprachen zu erfahren. Außerdem soll dieser Text dazu anregen, über die eigene Muttersprache oder das allgegenwärtige Englisch nachzudenken. Vielleicht ist er auch nützlich im Zusammenhang mit künstlicher Intelligenz oder — ganz einfach — beim Programmieren in Forth.

Wortklassen

Verben (Zeitwörter)

Schwierig zu lernen sind in vielen Sprachen die Verben. Es liegt in ihrer Natur, dass sie in vielen Modifikationen auftreten. In den gängigen imperativen Computersprachen tritt eben nur der Imperativ, die Befehlsform, von Verben auf. Weil die englische Sprache kaum Flexionen kennt, ist der Imperativ nur durch den Sinnzusammenhang erkennbar.

Im Lateinischen muss man für jede Konjugationsart 2*6=12 solche Tabellen und noch ein paar weitere Formen lernen. Damit hat man aber klare Aussagen über Person, Zeit, Modus (Indikativ/Konjunktiv), Genus verbi (Aktiv/Passiv). An der Tabelle 1 sieht man, dass die lateinische Sprache wesentlich mit Modifikationen durch an einen Stamm angehängte Endungen arbeitet und Pronomina als Subjekt weggelassen werden können, weil sie durch die Verbform ausgedrückt sind.

amo	ich liebe
amas	du liebst
amat	er/sie/es liebt
amamus	wir lieben
amatis	ihr liebt
amant	sie lieben

Tabelle 1: A–Konjugation, lateinisch, Präsenz, Indikativ, Aktiv

Im Russischen gibt es ähnlich wie im Lateinischen Konjugationstabellen zu lernen. Alle Verben treten doppelt auf, jeweils für den unvollendeten und den vollendeten Aspekt.

Я написал письмо
YA napisal pis'mo
I wrote a letter

Я писал письмо
YA pisal pis'mo
I was writing a letter

Diese Unterscheidung der Aspekte ist ähnlich wie die Unterscheidung im Englischen: "I speak" oder "I am speaking".

Im Chinesischen sind die Verben, wie alle Worte, unveränderlich. Varianten wie z. B. Vergangenheit werden durch Partikel wiedergegeben.

Viele Sprachen modifizieren Wörter durch „Anhängsel", d. h. durch Suffixe oder Präfixe. Im Hebräischen gibt es eine weitere Variante. Zusätzlich zu den Präfixen und Suffixen gibt es dort die Methode des erhaltenen Konsonantengerüsts, in dessen Lücken variable Vokale eingesetzt werden. Die drei Konsonanten `b--r--kh` bedeuten `Segen`. Dann ist `Baruch` („Barukh"), entstanden durch Einsetzen der Vokale `a` und `u`, der `Gesegnete`. Und als Name verwendet entspricht es dem lateinischen „Benedikt".

Im Hebräischen und im Russischen gibt es Verbformen, bei denen männlich und weiblich zu unterscheiden sind.

Welche Formen treten beim Programmieren auf?

Ein typisches Beispiel aus einer imperativen Programmiersprache ist eine Kombination von Imperativ und Objekt wie `PRINT X`. Weil die englische Sprache begrifflich verschiedene Formen in der Schrift und der Aussprache gleich erscheinen lässt, kommt der Gedanke, die Vielseitigkeit der Verben auszunutzen, nicht so leicht auf.

Aorist

Diesen Begriff kennen wohl nur die, die in der Schule Altgriechisch gelernt haben.

Der dazugehörige Wikipedia–Eintrag ist sehr ausführlich. Aber man kann aus der verwirrenden Menge von Beispielen und Betrachtungen, die vom alt–ehrwürdigen Sanskrit bis zur niedersorbischen Schriftsprache reichen, als gemeinsame Größe das „Fortdauern" finden. Ein für heutige Deutsche leicht zu merkendes Beispiel eines Aorist im Türkischen ist `döner`, denn der Spieß dreht sich fortdauernd. Im Englischen und im Deutschen gibt es keine spezielle Form dafür. Der „unvollendete Aspekt" im Russischen ist etwas Vergleichbares.

Professor Haiim Rosén schlägt diese Bezeichnung vor für eine Zeitform im Israeli–Hebräisch, die sonst weniger zutreffend auch als Gegenwart bezeichnet wird [2].

Was hat das mit *Forth* oder dem Programmieren zu tun? Das Setzen von *Konstanten* entspricht einer Aussage in der Zeitform des Aorist, denn der Wert soll ja fortdauern.

Im Forth haben einige Einstellungen von Variablen, wie z. B. die durch `BASE` bewirkte, auch etwas von diesem Charakter.

Speziell für Echtzeit–Programme wäre eine Unterscheidung zwischen aktuellen Größen und Konstanten wichtig. Dort warten Interrupt bedienende Programme dauernd, z. B. auf das Überschreiten von einmal festgelegten Grenzwerten.

Aber es ist Sache des Programmierers, den Überblick zu behalten. Eine gute Programmier–Umgebung sollte das Finden von Programmteilen und Interrupt–Service–Routinen, die einem Aorist entsprechen, erleichtern.

Der Aorist sollte eigentlich für die Formulierung aller mathematischen Sätze und Definitionen eingesetzt werden. Denn eine Zeitform von Verben, die Gegenwart mit Vergangenheit und Zukunft verbindet, passt zur Mathematik. Wir stolpern, dank unserer Gewöhnung, nicht darüber, dass mathematische Aussagen in der Gegenwartsform formuliert werden, was doch eigentlich unsinnig ist.

Aber auch von Gesetzestexten und juristischen Drohungen kennen wir diese Merkwürdigkeit, die uns nicht mehr auffällt, obwohl dort Handlungen, die in der Zukunft begangen werden könnten (!), in der Sprachform der Gegenwart mit Sanktionen belegt werden: „Unberechtigt parkende Fahrzeuge werden kostenpflichtig abgeschleppt".

Zeitformen

Hier stellt sich die Frage, welche Zeitformen eine Programmiersprache benötigt. Wie drücken wir den Konjunktiv, also die Möglichkeitsform, in unseren Programmen und Daten aus?

Wenn man nur in Begriffen eines Programms denkt, das in einer imperativen Sprache geschrieben ist, und zum Rechnen oder Verwalten dient, lautet die Zeitangabe zunächst „irgendwann in der Zukunft". Wenn man abgearbeitete Programme oder Programm–Teile betrachtet, muss man feststellen „in der Vergangenheit".

Für eine Datenbank, die zur Verwaltung einer Firma oder einer sonstigen beständigen Organisation dient, sollte der Aorist als angemessene Zeitform betrachtet werden. Denn die Datenbank wurde in der Vergangenheit aufgesetzt, lebt in der Gegenwart und hoffentlich auch in der Zukunft.

In kommerziellen Datenbanken protokolliert man, wann Einträge angelegt und bearbeitet wurden. Ebenso stehen dort Daten für die Zukunft, wie z. B. Termine und Beträge für Zahlungsverpflichtungen.

In Echtzeit–Systemen, wie sie zur Steuerung von Prozessen eingesetzt werden, sind die Gegenwart und Vergangenheit relevant. Aber in der Prozesstechnik kann auch die Zeit in der Zukunft, zu der der Prozess seinen Endzustand erreicht haben soll, eine Rolle spielen.

Wenn man sich entfernt von der Denkweise einer imperativen Programmiersprache, wird man Vergangenheit, Gegenwart und Zukunft wie in der natürlichen Sprache einsetzen können. Man denke an Wetterberichte, deren Inhalte in Planungen mit Hilfe von künstlicher Intelligenz genutzt werden.

Pronomina (Fürwörter)

Alle Sprachen, die ich kenne, enthalten Pronomina. Leo Brodie vergleicht die Verwendung von auf dem Parameter–Stapel eines Forth–Systems zwischengespeicherten Daten mit dem Gebrauch von Pronomina. Man gebraucht die Daten, ohne ihnen Variablennamen zu geben oder ihre Herkunft noch einmal zu benennen. So kann man `dup` beschreiben als dupliziere „es".

Überraschend ist, dass im Thailändischen das Pronomen für „ich" nicht nur männlich und weiblich unterschiedet, sondern auch noch verschiedene Formen der Höflichkeit oder (In–)Formalität. Dies findet sich in Tabelle 2. Unsere gerne genutzten automatischen Übersetzer im Internet (Google Translate) verwenden die männliche oder weibliche Form etwas willkürlich. Merkwürdigerweise wird: „Ich will ein Bier!" mit der weiblichen Form übersetzt.

Im Hebräischen sind die Fürwörter „Du" und „Dein", „Ihr" und „Euer" nach männlich und weiblich unterschieden.

Im Chinesischen werden die Pronomina „er, sie, es" durch dasselbe Wort wiedergegeben, was plausibel ist, wenn man bedenkt, dass es in dieser Sprache kein grammatisches Geschlecht gibt. Deshalb verwechseln Chinesen, wenn sie englisch oder deutsch sprechen, leicht „er" und „sie".

Angesichts dieser vielfachen Möglichkeiten unterschiedlicher Pronomina verwirrt es den Forth–Programmierer nicht, dass „es" bei den meisten Befehlen auf dem Parameter–Stapel residiert, seltener auf dem Rücksprung–Stapel, dem Gleitkomma–Stapel oder auf einem vom Programmierer angelegten Stapel.

Aber alle verschiedenen Stapel im Standard–Forth sind gewissermaßen geschlechtsneutral, wie man es zum Beispiel auch im Chinesischen erwarten würde. Es gibt kein *strong typing*. Es lassen sich aber Stapel konstruieren, deren Elemente weiter gekennzeichnet sind.

Adverbien (Umstandwörter)

Den Linux/Unix–Nutzern sind Angaben wie `/verbose` in der Kommandozeile vertraut. In natürlichen Sprachen entspricht dies einem *Adverb*. Ein Adverb bestimmt ein Verb, Adjektiv oder ein anderes Adverb näher und ist unflektierbar.

Auch Adverbien lassen sich in Forth realisieren. Man denke an Numerik–Programme, deren Resultate sich mit verschiedenen Algorithmen, die spezifiziert werden können, oder anpassbaren Parametern gewinnen lassen.

	Female Formal	Male Formal	Informal/Rude
I	ฉัน / ดิฉัน / ข้าพเจ้า/หม่อมฉัน	ผม/กระผม/ข้าพเจ้า/หม่อมฉัน	กู / ข้า
you	เธอ/คุณ	เธอ/คุณ	มึง /เอ็ง
he	เขา	เขา	มัน
she	เธอ	เธอ	มัน
they	พวกเขา	พวกเขา	พวกมัน
we	เรา	เรา	พวกก
it	มัน	มัน	มัน

Tabelle 2: Thai–Pronomina mit verschiedenen Stufen der Höflichkeit

Wortklassen und Formen, die in manchen Sprachen nicht auftreten.

Artikel (Geschlechtswörter)

Auffällig ist, dass einige Sprachen Artikel haben, andere nicht. Zum Beispiel im Russischen, Polnischen und weiteren slawischen Sprachen gibt es keine Artikel, im Chinesischen und Thailändischen auch nicht. In heutigen romanischen Sprachen hingegen findet man Artikel. Auch die germanischen Sprachen haben Artikel. In den Einzelheiten des Gebrauchs unterscheiden sie sich erheblich.

Kennt Forth Artikel? Betrachten wir die Funktion eines Artikels, so wird klar, dass Artikel den *unbestimmten Fall* (Beispiel „ein“ Mann) vom *bestimmten Fall* (Beispiel „der“ Mann) unterscheiden. Wenn man den bestimmten Artikel verwendet, zeigt man, dass man sich auf jemanden oder etwas Bekanntes oder vorher Erwähntes bezieht.

In der hebräischen Sprache zeigt sich etwas Ähnliches. Die Benutzung eines Namens zeigt an, dass nun die Regeln des bestimmten Falls gelten.

רחל היקרה

Rakhel hayakarah
Dear Rachel

ברכה גדולה

Berakha gedolah
a great blessing

הברכה הגדולה

haberakha hagedola
the great blessing

In vielen Programmiersprachen muss man Variablen und Unterprogramme vor Gebrauch vereinbaren. Wenn sie dann wieder auftauchen, handelt es sich um den bestimmten Fall.

Programmiersprachen wie BASIC verzichten auf die explizite Vereinbarung von Variablen und Feldern. Damit entsprechen sie den natürlichen Sprachen ohne Artikel.

Auch Forth kennt, so betrachtet, den bestimmten Fall, nämlich bei Variablen und generell bei vom Nutzer definierten Worten. Diese müssen vor Gebrauch definiert werden. Man hat dann zwar keine Artikel, aber den bestimmten Fall auch ohne bestimmte Artikel. Damit kann man zum Beispiel lesen „die“ Variable (wie oben definiert) wird aufgerufen.

Aber wichtig ist, dass der unbestimmte Fall nicht im Programmtext auftritt. Eine Größe ist definiert oder sie existiert nicht, außer möglicherweise im Plan des Programmierers.

Kopula (Verknüpfungswort)

Noch erstaunlicher für Kenner z. B. der deutschen, englischen und französischen Sprache ist, dass viele Sprachen ganz ohne Kopula — im Deutschen „bin, bist, ist, sind, ...“ — existieren.

Die russische Sprache verzichtet auf Kopula in der Gegenwartsform, verwendet sie aber in der Vergangenheit und der Zukunft.

мой брат инженер
moy brat inzhener
My brother (is) (an, the) engineer

Weiteres aus natürlichen Sprachen

Neubildung von Wörtern

Die Methoden der Wortbildung in natürlichen Sprachen sind vielfältig.

Im Deutschen gibt es beeindruckend konstruierte *Zusammensetzungen von Wörtern* wie z. B. das Bundeswehr–Einsatzbereitschaftsstärkungsgesetz (BwEinsatzBerStG). Am Schluss, nach einer langen Liste von Hauptwörtern, steht der Hauptbegriff, hier: Gesetz.

Im Hebräischen gibt es, wie auch in anderen semitischen Sprachen, den Konstrukt–Fall. Der Hauptbegriff geht voran vor Adjektiven oder anderen Substantiven. Es gibt spezielle Kennzeichnungen dafür, dass der Hauptbegriff durch ein anderes Substantiv näher spezifiziert wird. Beispiel:

```
arucha (Mahlzeit)
```

wird zu

```
aruchat erev (Abendmahlzeit)
```

Hier sind weitere Beispiele:

ארוחה
arukha
meal

ארוחת ערב
arukhat erev
evening meal

Mit leichter Abwandlung ließe sich dies in einer auf Forth basierenden Spezialsprache einsetzen. Man verwendet nach einem Hauptwort eine Partikel, die wohl mit Leerzeichen umrahmt sein müsste, durch die signalisiert wird, dass noch eine Spezifikation durch ein weiteres Hauptwort folgt

Im Chinesischen sind viele Begriffe durch Paare von zwei Wörtern gegeben. So bedeutet „gehen–sehen" (来看) unser „besuchen". Dies bedeutet auch für das Schreiben eine Lern–Erleichterung, weil man nur zwei bekannte Zeichen zusammenfügen muss. Das ist im eigentlichen Sinne eine Wortneubildung aus alter Zeit.

Bemerkenswert ist auch die Methode, Akronyme und Abkürzungen aus Anfangsbuchstaben zu bilden. Im Englischen sind in vielen Gebieten der modernen Technik Akronyme wie ROM (Read Only Memory) oder Abkürzungen wie CPU (Central Processing Unit) zu finden.

Auch die ehrwürdigen Rabbiner aus alter Zeit wurden und werden durch Akronyme bezeichnet wie z. B. RaMBaM (Rabbi Moshe Ben Maimon = Maimonides). Weil Vokale in der hebräischen Schrift fehlen, hat man für das Akronym nur vier Buchstaben und eine „Tüttelchen"–Markierung zu schreiben, die die Buchstabenfolge als Akronym kennzeichnet.

רמב"ם
Rambam, auch RaMBaM
Rabbi Mosche Ben Maimon (auch Maimonides)

Die Colon–Definition in Forth lässt sich auch als eine Methode der Wortneubildung auffassen. Auch durch `create ... does` werden Wortneubildungen geschaffen. Damit kann Forth Worte auf einer höheren Ebene neu bilden.

Komplimente, Schimpfwörter und Metaphern

Diese drei basieren meistens auf Substantiven. Sie bilden keine eigene grammatische Kategorie. Sie sind weniger durch den semantischen Aspekt als durch den pragmatischen gekennzeichnet. Ihr Gebrauch erzeugt Sympathie, Ärger, Vertrauen und andere Regungen. Nach Meinung des Autors sind Kenntnisse dieser Wörter im menschlichen Zusammenleben notwendig.

Tier–Metaphern, die in Deutschland als Kompliment wirken, können in anderen Kulturkreisen als Beschimpfung aufgefasst werden. Beispielsweise gilt in Ostasien ein Fuchs nicht als ein schlaues, sondern als ein egoistisches mit bösen Zauberkräften ausgestattetes Tier.

Vermutlich wird erst der Einsatz von AI in der automatischen Interpretation und Übersetzung von Texten, geschrieben oder gesprochen, die Betrachtung von solchen Feinheiten möglich machen. Dann wird ein freundlicher Deutscher, der seine chinesische Partnerin in bester Absicht eine „Fähe" (Füchsin) nennt, gar nicht merken, dass sein Übersetzungsprogramm aus der „bösen Füchsin" flugs einen „sympathischen, eleganten Kranich" gemacht hat.

Ansonsten ist der Gebrauch von Schimpfworten noch nicht Sache des Computers.

Metaphern in Forth?

Die Poesie lebt von bildlichen Vergleichen. Aber gibt es so etwas in Programmiersprachen?

In Forth können wir beispielsweise bei Vereinbarungen von Datenstrukturen etwas realisieren, das dem nahekommt. Wir können `A` nach dem Muster von `B` vereinbaren, sofern wir den notwendigen Unterbau realisiert haben:

```
DECL A SAME AS B
```

Aussprache und Schrift

Im Englischen gibt es eine Fülle von Überraschungen im Verhältnis von Aussprache zur Schrift.

George B. Shaw empfahl des Wort „ghoch" zu betrachten: „gh" wie in „enough", „o" wie in „women" und „ch" wie in „machine" ergibt eine Aussprache wie „fish".

Im Chinesischen gibt es in der üblichen Schrift keine Ausspracheangaben. Es gibt aber eine offizielle Lautschrift, *Pinyin* genannt, die auch Dialektsprechern erlaubt, die chinesische Standardsprache (Mandarin) richtig auszusprechen. Zur Warnung sei gesagt, dass Pinyin zwar mit lateinischen Buchstaben und diakritischen Zeichen (Akzenten), die die Töne bezeichnen, geschrieben wird, aber trotzdem die Aussprache einige Wochen gelernt werden muss.

Im Hebräischen wird bei Bedarf ein weiterer Zeichensatz als Punktierung unter, über und in die Buchstaben des normalen Textes eingefügt. So wird beispielsweise der Text der Bibel mit Punktierung geschrieben, weil man beim Vorlesen falsch ausgesprochene Vokale, die auch sachliche Verwechslungen erzeugen könnten, unbedingt vermeiden will.

Überraschend einfach ist das Verhältnis von Aussprache zur Schrift im Russischen. Wenn man die kyrillischen Zeichen kennt und weiß, welche die betonte Silbe ist, kann man bei Kenntnis weniger einfacher Regeln jedes Wort richtig aussprechen.

Bemerkungen zur chinesischen Sprache

Es gibt in der chinesischen Sprache kein grammatisches Geschlecht, keine Flexion und kein Wort, das genau dem „Ja" oder „Nein" entspricht.

Viele Wortbildungen sind Paare aus einfachen einsilbigen Bestandteilen, die für sich eine Bedeutung haben — wie wir es weiter oben am Beispiel 来看 schon gesehen haben.

Weil nach den Lautregeln der chinesischen Sprache nur etwa 1600 verschiedene Silben (die vier Töne mitgerechnet) gebildet werden können, ist die Konstruktion aus Paaren von einsilbigen Wörtern und längeren Sequenzen eine Notwendigkeit.

Wer Chinesisch lernen möchte, wird dies vielleicht für besonders einfach halten, weil keine Flexionsregeln zu lernen sind. Meine chinesischen Freunde behaupten gerne, dass die chinesische Sprache ganz einfach sei, und sprechen Worte wie Akkusativ und Dativ, die sie erst in Deutschland kennenlernen mussten, mit Schaudern in der Stimme aus.

Aber andere Schwierigkeiten, wie die Idiomatik, sorgen für die Notwendigkeit zu lernen. Woher sollte man sonst wissen, dass die Sequenz der chinesischen Worte für

马马虎虎
(Pferd, Pferd – Tiger, Tiger)

etwa soviel wie `mittelmäßig` bedeutet (Abb. 1).

Ein Text für viele Sprachen

Die chinesische Kultur hat etwas hervorgebracht, was verwandt mit dem *objektorientierten Programmieren* ist.

Die chinesischen Schriftzeichen werden von allen Chinesen verstanden. Aber die Sprache in z. B. Süd Fujian unterscheidet sich vom Hochchinesischen (Mandarin) so sehr, dass eine mündliche Verständigung unmöglich ist. Man behilft sich mit der schriftlichen Verständigung, die auch darin bestehen kann, die Zeichen nur „in die Luft zu malen“.

Da die Zeichen keine Aussprache kodieren, sind sie universell (zumindest aber in Sprachen ähnlicher Struktur) einsetzbar. Früher war es auch für Angehörige gebildeter vietnamesischer Familien üblich, Briefe mit chinesischen Zeichen zu schreiben.

Das ist vergleichbar mit der Forderung, beim Programmieren auf höherer Abstraktionsebene zu kodieren: Die Bits und Bytes kommen später. Analog gilt: Die Kodierung der Aussprache im Pinyin ist eine spätere Schöpfung.

Was lässt sich für Forth verwenden?

Forth ist — abstrakt betrachtet — dem Chinesischen verwandt: Alle Worte bleiben unverändert und werden nach Regeln zusammengefasst. Modifikationen der Abarbeitung kann man durch Partikel wie `IF ... THEN` steuern. Selbstgeschriebenen Worten kann man eine Wortklasse wie z. B. Substantiv, Verb oder Adjektiv zuschreiben.

Es stellt sich mir die Frage, ob auch die grammatischen Kategorien Subjekt, Prädikat und Objekt in forth–basierten Systemen erscheinen können.

Es scheint mir, dass in objektorientierten Systemen derartige grammatische Regeln, die sich auf ganze Sätze beziehen, ihren Platz finden.

Zum Schluss: Was erwarten wir nicht in Programmiersprachen?

Unter Literaten geht die Einsicht um, dass Ironie nur von jedem zweiten Leser verstanden wird. Dann würde ich sie innerhalb von Programmiersprachen nicht erwarten. Die Übertreibung als literarisches Kunstmittel scheint mir in eine ähnliche Kategorie zu gehören.

Humor und Komik erwarte ich auch nicht in Computersprachen. Also werden wir weiterhin ohne Computer–Unterstützung selbst lachen müssen.

Nun habe ich hier Eigenheiten natürlicher Sprachen gesammelt und kommentiert. Vielleicht finden einige Gedanken Eingang in die Forth–Programmierung.

Danksagung

Ich danke Michael Kalus für viele Diskussionen, Suthida Thongnuch für ihre Hilfe mit der Thai–Sprache und meiner Frau Lu–Ping Tan–Storjohann für die Pferd/Tiger–Illustration.

Ich habe hoffentlich die Geduld meiner Leser nicht über die Gebühr beansprucht, und ich hoffe, dass man mir die sparsame Verwendung von Zitaten nachsehen wird.

Bei *www.quora.com* könnt ihr übrigens eine Diskussion verfolgen zur Frage: „What's the difference between natural languages and programming languages?“

Links

`https://de.wikipedia.org/wiki/Aorist`
Online; Stand 1. Juni 2020

`https://de.wikipedia.org/wiki/Status_constructus`
Online; accessed 7–June–2020

`https://www.quora.com/Whats-the-difference-between-natural-languages-and-programming-languages`

Quellen

[1] Thinking Forth; Brodie, Leo, 2004, Punchy Pub.

[2] Textbook of Israeli Hebrew; Rosén, H.B., Language study, 1962. University of Chicago Press.

`https://books.google.de/books?id=d3IqE5f455wC`

[3] The Corpora of Mandarin Chinese and German Animal Expressions: An Application of Cognitive Metaphors and Language Change; Hsieh, Shelley Ching–yu, 2003, Proceedings of the Corpus Linguistics 2003 main conference.

Abbildung 1: Chinesische Idiomatik: „Pferd, Pferd–Tiger, Tiger“ bedeutet „mittelmäßig“

Fortsetzung von Seite 6

Scanning Source Code

A scanner is the very first stage when parsing source code. The scanner breaks up the character input stream into tokens, which constitute atomic syntactical elements.

In Forth, scanning is extremely simple: Each token is delimited by whitespace (spaces, tabs, and " (quote)). It is even simpler than the scanner that works in our brain when we read this text. Besides ␣ (whitespace), `,` (comma), `.` (full stop), `:` (colon), and `;` (semicolon) are delimiters as well: Thisresultsintextthatismucheasiertoreadforahuman.

This is different from most other programming languages since the age of Fortran and Cobol, whose delimiters are whitespace, mathematical operators, and predefined keywords. Tokenizing their input stream is a lot more complicated compared to Forth, which gave rise to regular expression (regex) processors in the 1950s.

The Forth way of scanning the input stream has advantages beyond simplicity: Because only whitespace are reserved characters serving as delimiters, a name may consist of all other characters of the ASCII set including not only alphanumeric but also special characters and even control characters, although the latter ones are difficult to print (and pronounce). E.g. `r>` and `>r` would be scanned as the identifier `r` and the comparison operator `>` in most computer languages. And `-` can be used as separator for compound names rather than `_` as e.g. in `open-file`.

This opens up new dimensions for semantic indication, e.g. a rule that every constant name begins with `#` . I venture to say that this makes the source code more readable and therefore, easier to understand.

I asked myself: why the heck is this so?

In an age of limited processing power more complicated rules for scanning than in human reading had been chosen. Then not only processing power was limited, but memory as well and source code was stored on punched cards. Dropping seemingly superfluous spaces was a rational decision then. It seems that this created an unquestioned tradition until today, still making lexical analysis more complicated than really necessary. Klaus Schleisiek

Fortsetzung der Rubrik auf Seite 34

Eine Einführung in das VIS–System

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 (eine Ersetzung für VOCABULARY), ITEM und STICKY. Die beiden letzteren Begriffe sind Neuschöpfungen.*

Um VIS kennenzulernen, ist es hilfreich, sich des Wortes `order` *zu bedienen. Dazu später mehr. In der Quellenangabe steht, wo ihr VIS bekommt.*

VIS, `find` und das Dictionary

Forth–Worte sind in einem *Dictionary* abgelegt. Das Dictionary ist eine Liste, die alle bekannten Forth–Worte des Forth–Systems enthält. Es ist beliebig erweiterbar. Diese Worte kann man sich mit `words` anzeigen lassen, in manchen Systemen wird dafür auch `list` verwendet.

Wird im laufenden Forth–System ein Wort aufgerufen, so passiert Folgendes: Das Forth–Wort `find` durchsucht das Dictionary nach der Zeichenfolge des Wortes, bis es sie gefunden hat. Dann übergibt es die Adresse, an der die Zeichenfolge steht, an weitere Forth–Worte, die dann den dazugehörigen Code ausführen oder kompilieren (in neue Worte einbauen). Wesentlich dabei ist, dass `find` immer (nur) die neueste Version eines Wortes findet, falls gleichnamige Worte im Dictionary vorhanden sind.

Dieser Mechanismus sorgt also dafür, dass gleichnamige Worte nicht parallel benutzt werden können. Er erzwingt auch ein–eindeutige Wortnamen. Es kommt aber häufig vor, dass man äquivalente Funktionen für verschiedene Aufgaben programmiert. Wen es stört, dafür immer wieder neue Namen (Worte) zu ersinnen, für den ist das VOCABULARY gedacht, bzw. hier das VOC–System. Das Dictionary wird in beliebig viele Wortlisten aufgeteilt, die jede für sich aktiviert und durchsucht werden kann.

Wie funktioniert VIS?

Die Basis–Suchreihenfolge im Forth–System

Ich glaube, dass das Wort `order` sehr hilfreich ist, um das VIS–System zu verstehen. Bei einem jungfräulichen VIS–System zeigt es Folgendes an:[1]

```
order <ret>
 context: forth forth root
 current: forth compiletoram
```

ok.

- Die Zeile `context:` zeigt an, in welchen VOCs das `find` nach Wortnamen sucht (permanenter Kontext).
- Die Zeile `current:` zeigt das VOC an, in welches aktuell[2] kompiliert wird.

Damit ermöglicht `current` es auch, in ein anderes VOC als das `forth` zu kompilieren, wenn man so ein weiteres VOC[3] definiert.[4] (Das `compiletoram` ist eine Besonderheit vom *Mecrisp*. Es zeigt an, dass ins RAM kompiliert wird. Das Gegenteil wird mit `compiletoflash` erreicht.)

Also, `context` zeigt an, dass hier zwei (drei) VOCs durchsucht werden und zwar in der Reihenfolge `forth - forth - root`. Das bedeutet: `find` sucht zuerst im VOC `forth` nach dem Wortnamen. Wenn es dort nicht fündig wird, sucht es in dem nächsten VOC, dem `root` (das zweite `forth` wird ignoriert).

Noch einmal anders gesagt: Durch die Reihenfolge, die durch `context` gegeben ist, wird festgelegt, welche VOCs in welcher Reihenfolge durchsucht werden. Und es kann durchaus weitere Kontext–VOCs geben! Dagegen kann current immer nur *ein* VOC enthalten, nämlich das, in welches gerade kompiliert wird.

Beim Systemstart ist die oben gezeigte Suchreihenfolge aktiv. Man kann wie gewohnt arbeiten, ohne sich überhaupt um das VIS–System kümmern zu müssen.

Mehrere VOCs anlegen

Der Spaß fängt an, wenn ich gezielt ein oder mehrere VOCs erzeuge. Das kann verschiedene Gründe haben: Ich will die Übersicht behalten und in einer Wortliste nur wenige Worte haben. Oder ich will Wortnamen mehrfach verwenden. Oder ich will „tiefe“ Worte unsichtbar machen. Oder ich will den Aufruf „gefährlicher“ Worte ein wenig absichern.

Ein kleines Beispiel soll das zeigen. Sagen wir mal, ich definiere Farbworte in deutscher Sprache, dann komme ich schnell in einen Namenskonflikt, sobald es um das Farbwort `rot` geht. Denn das ist schon durch das

[1] Das <ret> meint hier, dass `order` im Terminal *interaktiv* ins Forth eingegeben wird, man also anschließend die Return– bzw. Enter–Taste drückt. In den Quelltext–Beispielen hingegen ist das <ret> am Zeilenende nicht explizit angegeben, das ergibt sich beim Quelltexteinlesen aus einer Datei ja von allein.

[2] aktuell: engl. “current”

[3] Im VIS–System ist tatsächlich `voc` das definierende Wort für ein Vokabular, und nicht `vocabulary`,wie im klassischen Forth.

[4] `forth` taucht im `context` zweimal auf, das hat historische Gründe und soll uns hier nicht verwirren.

[5] engl. “rot” für “rotation of items on a stack”.

stack–manipulierende Forth–Wort `rot`[5] belegt. Da hilft es, alle Farbworte in ein eigenes VOC zu stecken.

```
voc farbe
farbe also
farbe definitions
: rot ( -- ) ... ;
: gelb ( -- ) ... ;
: gruen ( -- ) ... ;
...
```

Mittels `voc farbe` habe ich ein neues VOC mit dem Namen `farbe` erzeugt. Durch `farbe also` wird dieses VOC in die Suchreihenfolge eingebaut und `farbe definitions` sorgt schließlich dafür, dass alle nun folgenden Definitionen im VOC `farbe` landen. Schauen wir uns an, was `order` nun erzählt.

```
order <ret>
 context: farbe forth forth root
 current: farbe compiletoram
```

Das bedeutet: Beim Kompilieren wird zuerst das VOC `farbe` durchsucht, dann das VOC `forth` usw. Und die neu definierten Worte werden nun alle im VOC `farbe` abgelegt.

Ich befinde mich im Moment also im VOC `farbe` — nur: Wie komme ich dort wieder heraus?

Dazu gibt es mehrere Wege:

```
only <ret>
order <ret>
 context: forth forth root
 current: farbe compiletoram
```

`only` stellt für die Suchreihenfolge, und *nur* für die Suchreihenfolge, die jungfräuliche Situation wieder her. Eine andere Möglichkeit:

```
previous <ret>
order <ret>
 context: forth forth root
 current: farbe compiletoram
```

`previous` entfernt *nur den obersten* Eintrag aus der Suchreihenfolge. Hier war es das VOC `farbe`. Dumm ist dabei nur, dass alle Definitionen noch immer im VOC `farbe` landen. Wenn ich auch das ändern will, muss ich es dem System erneut ausdrücklich mitteilen.

```
forth definitions <ret>
order <ret>
 context: forth forth root
 current: forth compiletoram
```

Sind nun die Worte aus dem Farbe–Vokabular wieder unsichtbar für Forth? Wo ist da der Spaß? Der fängt jetzt erst an!

VIS macht's möglich

Egal, in welchem Kontext ich mich beim Programmieren auch immer befinde, sobald ich ein „Farbwort“ verwenden will, brauche ich nur das VOC–Wort `farbe` voranzustellen.

```
... farbe rot ...
```

Das war's dann schon. Kein explizites Umschalten des Kontextes mehr nötig.

Beispiel

Ich habe ein VOC `farbe` für Farben und einen `cursor` definiert. Das Wort `cursor` setzt den Cursor im Terminal auf die Stelle x,y des Bildschirms. Ein von `emit` ausgegebenes Symbol erscheint dann dort in der voreingestellten Farbe.

```
\ Drei Positionen
\ y x
  3 3  2Constant oben
  4 3  2constant mitte
  5 3  2constant unten

: leuchte ( x y -- )
  cursor at    \ schreib an x,y
  2 spaces ;

: aus ( -- )
  farbe default \ Farbe wie Hintergrund
  oben leuchte
  mitte leuchte
  unten leuchte ;

: ampel ( -- )
  aus
  begin
  farbe grün \ nur das grüne Licht
  unten leuchte 5000 ms
  aus
  farbe gelb \ nur das gelbe Licht
  mitte leuchte 2000 ms
  aus
  farbe rot \ nur das rote Licht ...
  oben leuchte 7000 ms  \
  farbe gelb \ ... und nun gelb dazu!
  mitte leuchte 2000 ms
  aus
  key? \ user exit ...
  until ; \ ansonsten und das Ganze noch mal.
```

Wenn ich hier mit `farbe <farbwort>` ein Farbwort aufrufe, so wird das Forth–Wort `farbe` nur beim Kompilieren aktiv! Im Kompilat selbst ist es nicht mehr vorhanden. Das bedeutet auch, dass kein Speicherplatz durch die Nutzung von VOCs verschwendet wird.

Nun erklärt sich auch der Begriff *permanenter Kontext* oder *permanente Suchreihenfolge* (permanent seach order) als Gegensatz zur zeitweiligen (temporary) Suchreihenfolge. Das Nennen eines VOCs macht genau dieses VOC zur *zeitweiligen* Suchreihenfolge. Das bedeutet, `find` sucht jetzt nur *einmal* in diesem VOC nach dem Wort. Nach dessen Verwendung wird automatisch in die zuvor aktive *permanente* Suchreihenfolge zurückgeschaltet.

Und was sieht nun unser `order` davon?

```
order <ret>
 context: forth forth root
 current: forth compiletoram
farbe gelb <ret>
order <ret>
 context: forth forth root
 current: forth compiletoram
```

Gar nichts!

Dass `farbe` tatsächlich temporär in das VOC `farbe` umgeschaltet hatte, merkt man nur daran, dass das Wort `gelb` gefunden worden ist. Mit dem Dictionary–Browser[6] `??` kann man das überprüfen.

Statt:

```
farbe rot <ret>
```

schreiben wir nun ein `??` *dazwischen*:

```
farbe ?? rot <ret>
```

und erhalten so eine Liste aller Worte[7] aus dem VOC `farbe` und noch einige Zusatzinformationen darüber, wie die „order“ temporär ist:

```
... name: default
... name: rot
... name: gruen
... name: gelb
...
<< FLASH: farbe
>> RAM: farbe
context: farbe root
current: forth compiletoram
base: 16
Stack: [ 0 ] TOS: 2A *>
```

Im Prinzip arbeitet `??` wie eine Kombination aus `order` und `words`.

Zusammenfassung

Mittels `also` und `first` werden VOCs in die *permanente* Suchreihenfolge eingetragen. Das ist das, was `order` zeigt.

Anstatt die Farbworte mit einem Präfix zu versehen (`farbe-rot`, `farbe-gelb` usw.), sage (schreibe) ich nun die Phrase[8]

```
farbe rot ...
farbe gelb  ...
```

Das macht den Code übersichtlich und kompakt. Da für den Compiler und den Interpreter die VOCs ganz normale Forthworte sind, kann man sie sogar schachteln.

Dazu, und zu den Worten `ITEM` und `STICKY`, also dem I und S vom VIS, demnächst mehr.

... to be continued ...

Quellenangabe

VIS für *430eForth* und *Mecrisp* sind beschrieben und zu bekommen unter: `https://forth-ev.de/wiki/projects:forth-namespaces:start`

Download:

`https://forth-ev.de/wiki/res/lib/exe/fetch.php/projects:forth-namespaces:vis-mecrisp-quintus.tar.gz`

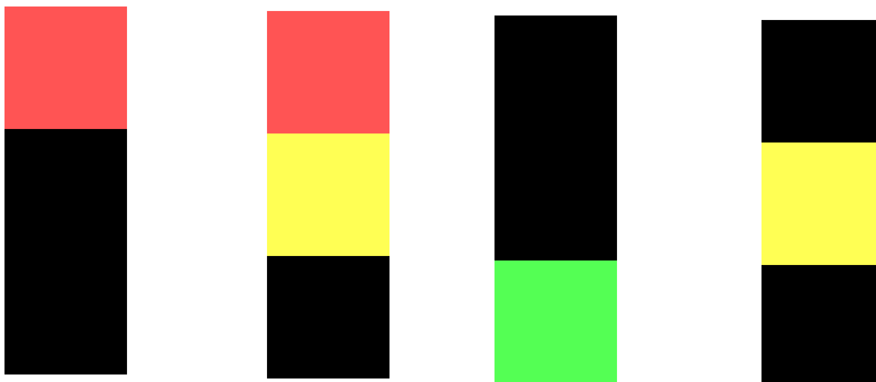


Abbildung 1: Ampel — Die vier zeitversetzten Screenshots vom e4thcom–Terminalfenster (Mecrisp mit VIS).

[6] ?? (zwei Fragezeichen) ist ein nützliches Forth–Wort zum Debuggen und im VIS–Paket enthalten.

[7] Die Liste ist sehr breit. Damit es in die Spalte passt, hier eine verkürzte Darstellung davon, die wohl den Sinn wiedergibt, aber Details wie den `wtag`, die `lfa` und das `xt` der Worte weggelassen hat.

[8] Natürlich, das VOC `farbe` mit seinen Farbworten muss vorher definiert worden sein.

VolksForth–Update

Carsten Strotmann

VolksForth ist das kleine 16–Bit–Forth der Forth–Gesellschaft. VolksForth gibt es für eine Reihe älterer und einige neuere Rechnersysteme. Es ist ein kleines, feines Forth–System, welches insbesondere Anhänger in der Retro–Computer–Szene hat.

Die aktuellen Quellen und Programmdateien des VolksForth befinden sich im GitHub–Repository des Projekts[1]. Im vergangenen Sommer des Jahres 2020 hat die Weiterentwicklung des VolksForth, insbesondere durch die Arbeiten von Philip Zembrod*, wieder Fahrt aufgenommen und es gibt so viele Neuigkeiten zu berichten, wie wahrscheinlich seit dem Ende der 1980er Jahre nicht mehr.*

Commodore 8–Bit

Philip Zembrod hat sich des VolksForth auf Commodore–8–Bit–Rechnern — C64 und C16, auch bekannt unter dem Namen *Ultra Forth* — angenommen und dort viele Neuigkeiten eingebracht, sodass diese Version nun die am weitesten vorgeschrittene darstellt.

File–Interface

Das Commodore–VolksForth arbeitet nun mit Stream–Files (regulären Dateien), das Block–IO–Interface ist noch optional verfügbar. Diese Änderung macht die Entwicklungsarbeit am VolksForth einfacher, da direkt auf den Quelldateien mit „großen“ Editoren auf modernen Betriebssystemen (Linux, macOS, Windows etc.) gearbeitet werden kann.

C16–Target gefixt

Die VolksForth–Version für den C16/Plus4–Computer war seit den 1990er Jahren defekt und funktionierte nicht mehr. Philip hat diese Version repariert und nun kann sie auf dieser Plattform wieder benutzt werden.

ANSI–Forth–kompatibel

Das Commodore–VolksForth hat als erstes Target die Versionsnummer 3.90 erreicht. Die von Philip in der Vierten Dimension 3/1995, Seite 11 vorgestellten Updates zu VolksForth 3.82 — `unloop 4+ DERROR? PENCOL` etc. — sind in diese Version eingeflossen.

Die Version 3.90 des VolksForth ist nun zum großen Teil ANS–Forth–kompatibel. Dies erleichtert die Übertragung von neuerem Quelltext.

Hayes–Test

VolksForth 3.90 kommt mit dem *ANS–Test von John Hayes* und besteht den Core–Test sowie viele Tests der Core–Extensions und Core–Plus–Tests. Durch die Portierung des Hayes–Test–Systems sind nun automatisierte Unit–Tests des VolksForth–Programmcodes möglich.

Automatisiertes Cross–Compiling mit VICE

Alle Commodore–Targets können nun automatisiert mithilfe des Commodore–Emulators *VICE*[2] auf modernen Betriebssystemen kompiliert werden. Die dazu benötigten Skripte finden sich im Quellcode–Repository.

Bugfixes

Im Wort `unloop` wurde ein Fehler gefunden und berichtigt. Das Wort `0>` hatte in allen 6502–CPU–Targets einen Fehler (Flag für den Eingabewert -32768 falsch); Stelle gefunden und gefixt. Dieser Fehler wurde durch den Hayes–Test (s. o.) aufgedeckt.

Atari ST

Das VolksForth–Programm für den Atari ST wurde von `4th.prg` in `volks4th.prg` umbenannt. Es wurde unter dem freien Betriebssystem *EmuTOS* für den Atari ST und Amiga und andere Motorola–68K–Systeme getestet. Dabei ist — eher durch Zufall — ein Fehler in der Textausgabe in dem Wort `STtype` entdeckt (High–Byte des ASCII–Werts != 0) und gefixt worden.

Quellcode–Updates

Neben den Arbeiten am VolksForth–Kern–System gibt es auch eine Reihe neuer Anwendungen und Portierungen von Forth–Quellcode.

MS–DOS–Meta–Compiler

Der Meta–Compiler für das MS–DOS–VolksForth ist nun Bestandteil des Projekts und die Verwendung des Meta–Compilers wurde dokumentiert.

Terminal–Programm mit Block–Interface

VolksForth kommt nun mit einem einfachen Terminal–Programm inkl. eines Block–IO–Interface für die serielle Schnittstelle. Dieses Programm war in den 1990er

[1] VolksForth–Quellcode und Programmdateien `https://github.com/forth-ev/VolksForth`

[2] VICE — the Versatile Commodore Emulator `https://vice-emu.sourceforge.io/`

Jahren Teil der Distribution des Forth für den *Zilog Super8*. Da dieses Programm aber auch für andere Anwendungen nützlich ist, wurde es in das VolksForth–Projekt aufgenommen.

MINIOOF

Die Forth–Erweiterung für objektorientierte Programme `MINIOOF`[3] von Bernd Paysan wurde auf VolksForth angepasst und ist nun Teil des Repositories.

RetroForth–Editor

Der RetroForth–Editor[4], ein moderner Line–Editor (wenn es so etwas gibt), wurde überarbeitet und einige Fehler wurden behoben.

Implementation von `N>R` und `NR>`

Die Worte `N>R` und `NR>` aus dem optionalen Entwickler–Tools–Wortset[5] wurde für VolksForth implementiert und ist nun im Repository zu finden.

Dokumentation

Bei all den Änderungen am Programm durfte auch die Dokumentation nicht zu kurz kommen, auch wenn noch viele Stellen einer Überarbeitung harren.

Die Dokumentation des CP/M–Targets wurde durch neu (wieder–) entdeckte Dokumentation vervollständigt und in moderne UTF–8–Textdateien übertragen, sodass kein altes CP/M mehr zum Lesen der Dokumentation notwendig ist. :-)

Der GitHub–Entwickler bodhi–baum hat die Lesbarkeit des VolksForth–Handbuchs verbessert.

Neue Target–Plattformen

Neben den etablierten Plattformen gibt es einige neue Target–Plattformen für das VolksForth.

py65

py65[6] ist ein Emulator für ein simples Computersystem mit einem 6502–Prozessor. Dieses System emuliert nur zeichenbasierte Ein– und Ausgabe und ist in Python geschrieben. Nicht schnell, aber dafür einfach anpassbar. Dieses Target wird benutzt, um generische Änderungen am 6502–VolksForth unter modernen Betriebssystemen automatisiert testen zu können. VolksForth unterstützt eine erweiterte Version des py65–Emulators mit Block–IO[7].

Commander X16

Der *Commander X16* ist ein neuer Commodore–8–Bit–kompatibler Computer[8]. Entwickelt wird dieses System von einem Team rund um David Murray (aka 8bit Guy). Bisher gibt es den Commander X16 nur als Emulator, die Portierung des VolksForth auf diesen „modernen Retro–Computer“ hat begonnen.

EMU2

EMU2 ist ein simpler textbasierter x86– und MS–DOS–Emulator[9]. Dieser Emulator versucht nicht, einen IBM–PC–kompatiblen Rechner mit aller Hardware zu emulieren, sondern stellt eine Emulation der 8086–CPU bereit. Alle MS–DOS–API–Aufrufe (Software–Interrupts) werden in Systemaufrufe für Linux umgesetzt. EMU2 läuft im Terminal–Fenster unter Linux und eignet sich hervorragend zur Entwicklung unter VolksForth–MS–DOS.

Damit VolksForth unter EMU2 korrekt funktioniert, wurde die zeichenbasierte Ein– und Ausgabe angepasst, sodass VolksForth auch mit diesem minimalen MS–DOS–System funktioniert.

tnylpo CP/M

Das Target mit dem schwierigsten Namen ist *tnylpo*[10]. Es ist ein CP/M–Textmode–Emulator für Linux/Unix und somit von der Idee vergleichbar mit dem oben beschriebenen EMU2. Es erleichtert die Arbeit mit VolksForth unter modernen Linux/Unix–Systemen.

Atari Portfolio (8086–MS–DOS)

Der Atari Portfolio ist ein kleiner und portabler 80x86–MS–DOS–kompatibler Minicomputer aus dem Jahre 1989.[11] Mit 240 x 64 Pixeln (40 x 8 Zeichen) hat der Portfolio einen sehr beschränkten Bildschirm. Im VolksForth befinden sich nun spezielle Bildschirmtreiber für das MS–DOS–VolksForth, welche die Ansteuerung des LCDs des Portfolio erlauben.

---

[3] MINIOOF `https://bernd-paysan.de/mini-oof.html`
[4] RetroForth–Editor `https://web.archive.org/web/20110811120814/http://retroforth.org/pages/?PortsOfRetroEditor`
[5] N>R NR> `https://forth-standard.org/standard/tools/NRfrom`
[6] py65–Emulator `https://github.com/mnaberez/py65`
[7] py65 mit Block–IO `https://github.com/cstrotm/py65`
[8] Commander X16 `https://www.commanderx16.com`
[9] Emu2–MS–DOS–Emulator `https://github.com/dmsc/emu2`
[10] tnylpo `https://gitlab.com/gbrein/tnylpo`
[11] Atari Portfolio `https://en.wikipedia.org/wiki/Atari_Portfolio`

CC64–C–Compiler

Philip Zembrod hat mit *CC64* einen C–Compiler für Commodore–Rechner mit 6502–CPU und 64 KB RAM geschrieben.[12] Dieses Projekt war die Motivation für die Weiterentwicklung des Commodore–VolksForth und ein Beweis, das in VolksForth umfangreiche Projekte (> 50K lines of code) möglich sind. Philip hat CC64 auf der *EuroForth–Tagung* im September 2020 vorgestellt, ein Video–Mitschnitt ist auf YouTube[13] erhältlich. cas

---

Fortsetzung von Seite 28

Gegen die Corona–Langeweile

Die Tagung im kommenden Jahr 2021 führt uns, ja, wohin? Das ist nun völlig offen. Wünscht euch was! Mein Tipp: Forth — der Ort! Dort kennt man keine Langeweile …

Jedenfalls ließ die Gemeinde Forth sich neulich was einfallen gegen ihre Corona–Langeweile: „Schnitzeljagd für die ganze Familie — durch Forth“. Veranstaltet von der Kinder– und Jugendfeuerwehr Forth. Der Eckentaler *wochenklick* wusste zu berichten.

Da hieß es:

> „Start und Zielpunkt ist das Feuerwehrhaus in Forth. Die Runde ist ca. 5,5 Kilometer lang und kann völlig flexibel durchgeführt werden. Gedruckte Karten und Fragebögen hängen wir auch an die Nebeneingangstüre am Feuerwehrhaus aus.
>
> Die Lösungen bitte notieren und danach in den Briefkasten am Feuerwehrhaus Forth einwerfen. Wir verlosen unter allen Teilnehmern 10 x den Plüschdrachen …“

Plüschdrache? Kommt uns das nicht bekannt vor?

So eine forthige Schnitzeljagd können wir doch auch, oder?

Also, wer hat Ideen für die 12 Stationen?

Einsendungen bitte an die Forth–Gesellschaft.

Im nächsten Heft gibt es dann, wenn ihr mitmacht, den Rundkurs. Also Weihnachten hat’s dann Forthiges zum Tüfteln.

Bis dahin, alles Gute euch.

Quelle: `https://www.wochenklick.de/eckental/c-lokales/schnitzeljagd-fuer-die-ganze-familie-durch-forth-von-der-kinder-und-jugendfeuerwehr-forth_a3720`

[12] CC64–C–Compiler in VolksForth `https://github.com/pzembrod/cc64`

[13] CC64–Vorstellung (EuroForth 2020) `https://www.youtube.com/channel/UC_mpkwOO_1ILd66GUTNVPQg/videos`

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

Jitsi `https://meet.forth-ev.de`

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

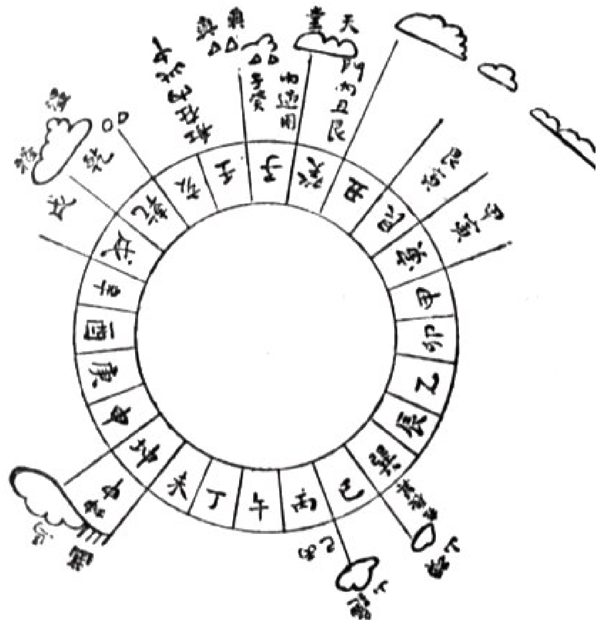
Donnerstags ab 20:00 Uhr
Forth–Chat net2o `forth@bernd` mit dem Key `keysearch kQusJ`, voller Key:
`kQusJzA;7*?t=uy@X}1GWr!+0qqp_Cn176t4(dQ*`

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.

Mitgliederversammlung und Tagung

Carsten Strotmann

Die Covid–19–Pandemie hat so manche Planung für das Jahr 2020 umgeworfen. Die Forth–Gesellschaft hatte die seit der Gründung regelmäßig jährlich abgehaltene Tagung und Mitgliederversammlung für den März 2020 geplant. Wir wollten uns im CocoNAT in Klein Glien bei Bad Belzig im schönen Naturpark „Hoher Fläming“[1] *treffen. Die Pandemie hat dies leider nicht zugelassen.*

So hat die Forth–Gesellschaft im März stattdessen kurzfristig, aber erfolgreich, eine Tagung online mit vielen interessanten Vorträgen organisiert. Die Videos der Vorträge sind online abrufbar.[2]

Das Organisationsteam der Tagung hat damals angekündigt, wenn möglich, die Tagung und Mitgliederversammlung im Herbst 2020 wie gewohnt „in persona“ nachzuholen. Auch wenn einige Bundesländer ein Treffen mit unter 50 Personen erlauben, ist das Direktorium der Forth–Gesellschaft nach Prüfung der Satzung, Gesprächen mit Mitgliedern und in Hinblick auf die Covid–19–Situation in Deutschland im August 2020 zu der Entscheidung gekommen, keine weitere Tagung und auch keine Mitgliederversammlung im Jahre 2020 auszurichten.

Keine Mitgliederversammlung in 2020, geht das?

Seit der Gründung der Forth–Gesellschaft wurden die Mitgliederversammlungen jährlich im Rahmen der Tagung durchgeführt, dieses hat sich bewährt und das aktuelle Direktorium möchte an dieser Regelung prinzipiell nichts ändern.

Die Satzung unseres Vereins schreibt aber keine Zeitintervalle für die ordentlichen Mitgliederversamlungen vor. Auch der Gesetzgeber gibt da keine Regelung vor.

Das Direktorium sieht nach Beratung derzeit keine zwingenden Gründe, welche eine Mitgliederversammlung (online oder in persona) in 2020 notwendig macht. Das derzeitige Direktorium führt gerne das Amt für eine um ein Jahr verlängerte Amtszeit aus.

Jedoch hat das Direktorium nicht alles im Blick. *Gibt es wichtige Themen, die keinen Aufschub bis zum Frühjahr 2021 erlauben?* Mitglieder der Forth–Gesellschaft haben die Möglichkeit, eine Mitgliederversammlung anzufordern. Sollte dies der Fall sein, bittet das Direktorium um eine schriftliche Meldung.

Mitgliederversammlung 2021

Im *Gesetz zur Abmilderung der Folgen der COVID–19–Pandemie im Zivil–, Insolvenz– und Strafverfahrensrecht*[3] hat der Gesetzgeber temporär einige Regeln für Vereine geändert, z. B. ist es Vereinen derzeit erlaubt, die Mitgliederversammlung online durchzuführen, auch wenn dies bisher in der Satzung nicht explizit erlaubt war.

Zum Glück erlaubte die Satzung der Forth–Gesellschaft schon immer eine rein elektronische Mitgliederversammlung:

> „Alle Beschlüsse der Vereinsorgane werden auf einer ordnungsgemäß einberufenen Sitzung gefasst. Solche Sitzungen können schriftlich, in persona oder fernmeldetechnisch (das heißt z. B. Videotext, Telefon, Computernetzwerk, Telex, über Satellit usw.) stattfinden. [. . .] Es entspricht dem Vereinszweck, fernmeldetechnische Möglichkeiten zu erproben und selbst weiterzuentwickeln und diese insbesondere für das Vereinsleben auch einzusetzen.“[4]

Das Direktorium wird die Covid–19–Pandemie im weiteren Verlauf beobachten und im Januar 2021 die Entscheidung treffen, ob im Frühjahr 2021 eine reguläre Mitgliederversammlung „in persona“ oder online durchgeführt werden kann. Es ist der klare Wunsch des Direktoriums, in 2021 eine Mitgliederversammlung durchzuführen. Die Mitglieder werden frühzeitig schriftlich im Frühjahr 2021 über den Termin, den Ort und die geplante Form der Versammlung informiert.

Bis zum Frühjahr 2021 wird das Direktorium, unterstützt durch interessierte Mitglieder, Systeme zur Durchführung von Online–Mitgliederversammlungen und Möglichkeiten der geheimen Wahl über Online–Medien testen. Wer Zeit und Interesse an dem Thema hat, ist herzlich zur Mitarbeit eingeladen.

---

[1] `https://www.hoher-flaeming-naturpark.de/`

[2] `https://www.youtube.com/playlist?list=PLR1LOfn4ZiNesbuONckWzQ1vqd1QhDloI`

[3] `https://www.bmjv.de/SharedDocs/Gesetzgebungsverfahren/DE/FH_AbmilderungFolgenCovid-19.html`

[4] `https://wiki.forth-ev.de/doku.php/infos:forthgesellschaftev`