Vierte Dimension

1/2021

Das Forth–Magazin

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

In dieser Ausgabe:

Dienstleistungen und Produkte

tematik GmbH
Technische
Informatik

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

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

Forth–Schulungen

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

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

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

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

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

Cornu GmbH
Ingenieurdienstleistungen
Elektrotechnik

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

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

KIMA Echtzeitsysteme GmbH

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

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

FORTecH Software GmbH

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

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

Ingenieurbüro
Klaus Kohl–Schöpe

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

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

Mikrocontroller–Verleih
Forth–Gesellschaft e. V.

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

Titelbild: Ohne Namen AUTOR mk

Quelle: Irgendwo gefischt und modifiziert.

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

Liebe Leser,

man nimmt das immer als so selbstverständlich an, dass ein Zeichen, das auf die Taste eines Keyboards gedruckt ist, dann auf dem Bildschirm auch so aussieht. Welch ein weiter Weg das aber ist, sowas zu verwirklichen, ahnt man kaum. Anton Ertl führt uns ein in diese zauberhafte Welt, und man beginnt zu verstehen, wie verhext der Umgang mit Schriftzeichen sein kann.

Was *parser* und *recognizer* aus solchen Zeichen dann machen, hängt ganz davon ab, in welcher Stimmung jene sind, ihre *states* sind von grundlegender Bedeutung. Klaus Schleisiek zeigt das einleuchtend auf. Und weil es zur Unterscheidung der Fälle vonnöten ist, lieferte er das passende *case* gleich dazu. Darüber hinaus kam für die Rätselecke seine Art, *conditional compiling* zu implementieren, dazu.

Matthias Koch und Robert Clausecker zeigen ihre ästhetische Seite mustergültig an den atmenden LEDs verschiedener MCUs, was je nach Board kniffelig sein kann. Das ist ja etwas, das mich immer wieder anspricht, diese minimalistische Kunst, mit einfachen Mitteln, nur eine LED, etwas Schönes zu machen.

Und Ulrich Hoffmann führt uns ein in die Kunst, mit *Docker* jedes Forthsystem auf jedem Betriebssystem laufen zu lassen. Fantastisch! Das will ich auch können! Und, wer kann das bereits und schreibt mal über seine Erfahrungen damit?

Und schließlich kommt Matthias Koch auf das kooperative Multitasking zurück. Verblüffend einfach, oder? Obschon, ich glaube, ich male mir das mal auf, um es so richtig zu verstehen.

Tja, und dann ist unser Forth–Jahrestreffen schon wieder nicht möglich gewesen. So um diese Jahreszeit herum war das ja bis 2019 immer. Wenn der Schnee geschmolzen war und man wieder reisen konnte. Diesmal ist es nicht der Schnee, der uns trennt.

Und die Kulturlandschaft ist so leer wie der Rest dieser Seite ... mk

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-2021-01

Die Forth–Gesellschaft e. V. wird durch ihr Direktorium vertreten:
Ulrich Hoffmann
Bernd Paysan
Carsten Strotmann
Kontakt: Direktorium@Forth-ev.de

Forth2020 Facebook–Gruppe

Im vergangenen Jahr hat sich die Gruppe *Forth2020* bei Facebook gegründet, ursprünglich, um Neuigkeiten rund um Tom Zimmers *Win32Forth* auszutauschen und die Weiterentwicklung dieses Systems zu organisieren.

Mittlerweile hat sich aus dieser Usergroup eine internationale Interessengruppe von Forth–Begeisterten des gesamten Erdballs mit *über 600 Mitgliedern* entwickelt. Teilnehmer kommen aus den USA, Brasilien, Argentinien, aus der Schweiz, Österreich, Deutschland, Frankreich, den Niederlanden, Schweden, Tschechien, Russland, Japan, China, Australien, u. a. und verwenden alle möglichen Forth–Systeme.

Auf dem Forum bei Facebook wird fleißig diskutiert und es werden dort viele Infos ausgetauscht, aber Facebook ist ja nicht jedermanns Sache. (Es lässt sich aber auch gut über Tor — und auch nur gelegentlich — verwenden.)

Wer sich nicht auf Facebook bewegen mag und dort nicht Mitglied der Gruppe werden will, kann dennoch an den monatlichen Zoom–Treffen teilnehmen, den spannenden Vorträgen zuhören und eifrig diskutieren. Die Treffen finden an jedem *2. Samstag im Monat* statt. Derzeit um 14:00 UTC (also 15:00 MEZ oder 16:00 MESZ). Neue Teilnehmer sind dort sehr willkommen. Genauso wie eigene Berichte über die persönlichen Forth–Aktivitäten.

Die Treffen werden mitgeschnitten und auf dem Forth2020–YouTube–Channel veröffentlicht, den man ebenfalls abonnieren kann.

Angekündigt werden die Zoom–Treffen über die Web–Seite `www.forth2020.org` und dort kann man sich auch zur Forth2020–Gruppe anmelden.

Auch wenn wir jetzt das Jahr 2021 haben, soll der Name *Forth2020* bestehen bleiben. Hat trotzdem was von Zukunft.

uho

Videotreffen „Forth lernen“

Jeden Montagabend um 20:30 Uhr trifft sich eine Gemeinschaft von Forth–Interessierten auf der Plattform `https://senfcall.de/`, um sich über Themen rund um Forth auszutauschen. Wie der Titel suggeriert, ist das Format gerade auch für Einsteiger geeignet. Der unterschiedliche Kenntnisstand der Teilnehmer sorgt dafür, dass es immer spannend bleibt. Es geht zwar primär darum, Forth zu lernen, aber die Runde ist auch offen für angrenzende Themen, die den Anwesenden auf den Nägeln brennen. Für die Teilnahme muss keine Software installiert werden. Ein Browser, Mikrofon und Kamera reichen. Eine Einladung mit dem Teilnahmelink bekommt man durch das Senden einer E–Mail an `wost@ewost.de`

Wolfgang Strauß

Videotreffen „Projekt Feuerstein“

Jeden Dienstagabend um 20:30 Uhr trifft sich das Team des Projektes Feuerstein auf der Plattform `https://senfcall.de/`. Bei Feuerstein geht es darum, Hilfen zu entwickeln, die beim Einstieg in das Thema „Forth auf Embedded–Systemen“ die Hürden und Fallstricke vermeiden sollen. Dokumentation, Werkzeuge und Experimentierhardware für RISC–V–Mikrocontroller unter dem Mecrisp–Quintus–Forthsystem sind in der Entwicklung. Interessenten sind auf unseren Treffen herzlich willkommen. Für die Teilnahme muss keine Software installiert werden. Ein Browser, Mikrofon und Kamera reichen. Eine Einladung mit dem Teilnahmelink bekommt man durch das Senden einer E–Mail an `wost@ewost.de`

Wolfgang Strauß

SVFIG „Fourth Saturday“ per Zoom

So wie es ausschaut, gibt es nun regelmäßige Zoom–Treffen der *Silicon Valley Forth Interest Group*. Kevin Appert organisiert das derzeit wohl von Palo Alto, Kalifornien, USA, aus; via Meetup. Am Samstag, 27. März 2021 von 09:30 bis 12:00 AM Pacific Time war das jüngste Treffen. Aus der Ankündigung:

> Details
>
> *** PLEASE NOTE 09:30 AM (PACIFIC) START TIME ***
>
> All duration and descriptions are approximate or perhaps entirely inconsistent with what will eventually transpire. No FIG agenda ever survives contact with the enemy ...

Leon Wagner, FORTH Inc., sollte sprechen. Er wollte *FORTH Inc's SwiftX® Tethered Forth* für eingebettete Prozessoren beschreiben und demonstrieren.

mk

Bit–Nr in Forth? 0...31 oder 1...32?

Neulich entstand diese Frage in der *Forth–Lernen*–Online–Runde[1] angesichts der Phrase:

```
1 <addr> bis!
```

mit der das erste Bit an der Adresse gesetzt wird. Oder es wird gelöscht mit:

```
1 <addr> bic!
```

Das sieht auf den ersten Blick so aus, als ob in Forth die Bits von 1...32 gezählt würden. Doch dem ist nicht so! Das macht man in Forth hübsch genauso wie im Datenblatt des Targets auch, von 0...31. Das erste Bit ist BIT 0, das zweite ist BIT 1, das dritte BIT 2 usw.

Der visuelle Effekt und damit der Irrtum entsteht, weil `bic!` und `bis!` gleich ganze 32–Bit–Masken setzen und rücksetzen.

```
bis! ( mask addr -- )
bic! ( mask addr -- )
```

[1] Eine Online–Zusammenkunft, die seit 2020 von Wolfgang Strauß an jedem Montagabend gemacht wird

Das sieht dann, wenn die Maske 1 ist, so komisch aus, so als ob wir bei 1 anfangen zu zählen. Gibt man ein Lineal hinzu, z. B. für die unteren 8 Bits, und schreibt die Maske binär auf, wird die Sache jedoch klar.

```
base @

2 base !

\   bit-mask  addr  word
\ 9876543210
  0000000001 <addr> bis!
  ...
  0000000001 <addr> bic!

base !
```

Da sieht man nun klar, dass in dem Beispiel mal das BIT 0 gesetzt und mal gelöscht wird.

Die alten Hasen kennen die umklammernde Phrase

```
base @ ... base !
```

noch. Sie diente dazu, zwischendrin in eine andere Zahlenbasis zu kommen. Von `decimal` nach `binary` und zurück.

Solche Zahlenbasismanöver sind nicht so komfortabel. Daher haben *Mecrisp* und viele andere der üppigeren Forth–Kernel Präfixe, die man vor die Zahlen geben kann. Selbst das kleine *noForth* kennt da einen raffinierten Trick. Aber das ist ein Thema für sich.

mk

Forth bei Heise Developer beliebt

Hi,

der Forth–Artikel vom 25.12. ist der zweitbeliebteste Artikel bei Heise Developer im Jahr 2020 gewesen.

```
<https://www.heise.de/news/Die-zehn-beliebtesten-
Beitraege-auf-heise-Developer-im-Jahr-2020-
5021641.html?seite=2>
```

Sicher, 25.12. und 26.12. war „Saure–Gurken–Zeit“, es gab fast keine anderen Meldungen auf Heise Online. Trotzdem bin ich überrascht.

Beste Grüße an alle Forther.

Carsten Strotmann

jeforth

Dr. Ting hat neulich sein kleines *eForth* als *Javascript* veröffentlicht. Es kam als `jeforth603.zip` zu mir. Die Anwendung ist denkbar einfach und im `readme.txt` hinterlegt:

```
This package contains two projects. Double
click these files

to run them:

jeforth503.html run jeforth system

jsBach.html play Bach polyphonic pieces

Documentations are shown in a text box.
```

Im Browser erscheint dann ein neues, zweigeteiltes Registerblatt. Der rechte Teil ist ein Text–Fenster mit 20 einführenden Forth–Lektionen. Die gewünschte Lektion markiert man und kopiert sie nach links in die Input–Box. Auf ein `enter` dort führt das jeforth diese Eingabe aus und das Ergebnis erscheint unterhalb in einer Output–Box.

Das jeforth macht auch Musik: `jebach.html` spielt einige Fugen. Der Grundwortschatz des jeforth ist Englisch, `words` zeigt ihn. Es kann aber auch chinesische Zeichen verarbeiten. Das Beispiel zeigt Tings Lektion Nr. 18. mk

```
https://github.com/hcchengithub/jeforth.3we

( 練習18 文字遊戲, a chinese word game )
( 這是一個用骰子玩的文字遊戲。用三顆骰子，
  每顆有六面各別寫上兩個字。)
( 一顆寫的是人物名稱，一顆寫的是地點，
  一顆寫的是動作。正常的六句詩是: )
( 公子章台走馬 少婦閨閣刺秀 寒士茅舍讀書 )
( 屠夫市井揮刀 妓女花街賣俏 乞丐墳墓睡覺 )
( 隨便擲這三顆骰子可以有216種不同的組合，
  有許多組合是蠻有趣的。)
( this is a chinese word game )
( you use 3 dices:
  one has 6 person,
  one has 6 places,
  and one has 6 actions )
( roll the dices and you get a line.
  many of them are funny )
( this shows that jeforth is a
  chinese programming language )
: 人物 ( n -- , 選一個人 select a person )
  dup 1 = if ." 公子"
  else dup 2 = if ." 少婦"
  else dup 3 = if ." 寒士"
  else dup 4 = if ." 屠夫"
  else dup 5 = if ." 妓女"
  else ." 乞丐"
  then then then then then
  drop
;
: 地點 ( n -- , 選一個地點 select a place )
  dup 1 = if ." 章台" drop exit then
  dup 2 = if ." 閨閣" drop exit then
  dup 3 = if ." 茅舍" drop exit then
  dup 4 = if ." 市井" drop exit then
      5 = if ." 花街" drop exit then
  ." 墳墓"
;
: 動作 ( n -- , 選一個動作 select an action )
  dup 1 = if ." 走馬" drop exit then
  dup 2 = if ." 刺秀" drop exit then
  dup 3 = if ." 讀書" drop exit then
  dup 4 = if ." 揮刀" drop exit then
      5 = if ." 賣俏" else ." 睡覺" then
;
: 骰子 ( -- , 印一句詩 print a line of poem )
```

```
  cr
  random 6 * int 人物
  random 6 * int 地點
  random 6 * int 動作
;
: dice 骰子 ;
dice dice dice dice
```

mk

noForth

Ich erhalte regelmäßig auch die Einladungen zu den Treffen der *niederländischen Forth–Gruppe*. Derzeit sind es Videokonferenzen statt leibhaftiger Treffen. Im Februar stand zum Beispiel Folgendes an:

- Neues vom Egel–Projekt–V2 — Vier Random–Generatoren in noForth, die von Brodie, Knuth, LFSR und Marsaglia.
- Stand der Dinge beim RISC–V — Mittlerweile gibt es einige einfache Beispiel–Platinen, auf denen noForth R(C)V läuft.
- Mesh–Netzwerk mit nRF24L01 — Sie sind jetzt bei Version 3.9 des Netzwerks angekommen. Der Code wurde weiter bereinigt und neue Optionen hinzugefügt.

Wunderbar die Idee, die Güte der Random–Generatoren auf dem kleinen Display des Projekts grafisch darzustellen. Da sieht man gleich, ob sich Muster ergeben oder es gut zufällig verteilt zugeht.

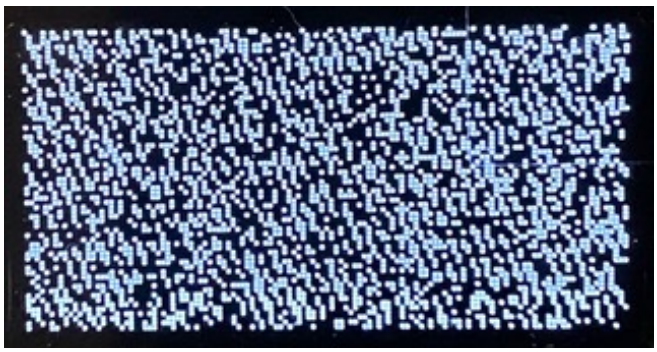


Abbildung 1: Güte des Brodie–Algorithmus

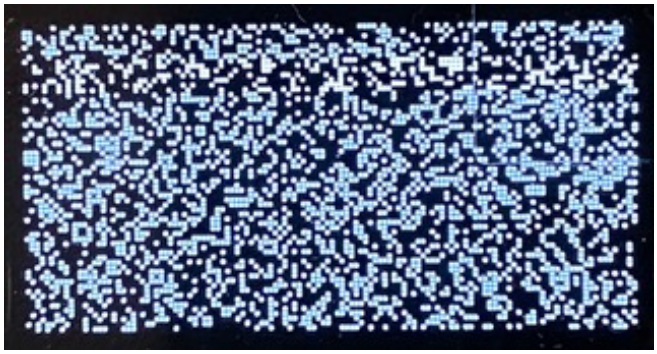


Abbildung 2: Güte des Marsaglia–Algorithmus

„Prof. Marsaglia is the developer of the diehard randomization test–suit. A generator invented by him is the 4th generator in the list. It generates normally distributed random numbers with low predictability. On top of that, all bits are equally random. As the method is simple and pretty fast, it is a true work–horse method.

The version shown here uses two 16b seeds resulting in a wrap–to–zero cycle of 2^32–1. So it can generate new numbers at full speed for almost 8 hours before it returns to the starting–point.“ (Aus: `https://home.hccnet.nl/willem.ouwerkerk/egel-for-msp430/egel%20for%20launchpad.html#e11x`)

Beeindruckend fand ich auch, wie weit das Funknetz schon gediehen ist. Verbindungen über mehrere Knoten hinweg sind da wohl kein Problem mehr.

`https://home.hccnet.nl/anij/index.html`

mk

PlanckForth — ein minimales Linux–Forth

In der letzten VD hatten wir mit *SectorForth* ein Beispiel für Forth–Minimalismus. Das SectorForth passt in die 512 Bytes des Bootsektors einer Diskette oder Festplatte eines PC–Systems. PlanckForth ist nun ein weiteres Beispiel für Forth–Minimalismus:

PlackForth ist ein Linux–Programm als handgeschriebene ELF–Linux–Datei[2]. D. h., PlankForth liegt nicht im Quelltext als Assembler oder Hochsprache vor, sondern wurde vom Autor Koichi Nakamura manuell in einer Hex–Datei geschrieben. Diese Hexdatei ist der „Quellcode“ und wird in Binärcode „übersetzt“.

Das Ergebnis ist ein minimales Forth mit 37 „Forth–Wörtern“, die jeweils aus nur einem Buchstaben, bei Unterscheidung von Groß– und Kleinschreibung, bestehen. Das resultierende Binärprogramm ist genau 1024 Bytes groß, also doppelt so groß wie SectorForth. In dieser Basis–Variante sehen die PlankForth–Programme noch sehr kryptisch aus, hier ein „Hello World“–Beispiel:

```
kHtketkltkltkotk tkWtkotkrtkltkdtk!tk:k0-tQ
```

Das minimale PlanckForth lässt sich via Forth–Quellcode in ein vollständiges Forth erweitern, und dieses Forth lässt sich wie gewohnt benutzen.

PlanckForth Binary Layout

Der Quellcode von PlanckForth, inkl. Beispielen und Core–Tests, findet sich auf GitHub: `https://github.com/nineties/planckforth`

PlanckForth ist Open Source unter der MIT–Lizenz.

cas

[2] ELF = Executable and Linking Format — Dateiformat für ausführbare Dateien unter Unix/Linux.

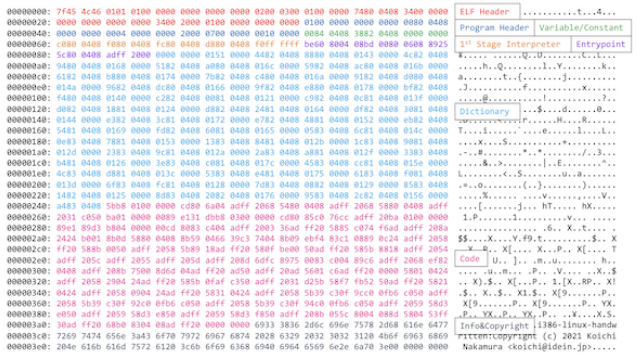


Abbildung 3: PlanckForth Binary Layout

CamelForth in C for RP2040 Raspberry Pi Pico

Ein Forth–Interpreter in C von Dr. Brad Rodriguez für den RP2040 und Raspberry Pi Pico. Es benötigt pico–sdk und ist mit Hilfe der Pico—Beispiele entstanden. Das `/dev/ttyACM0` ist die primäre Schnittstelle (USB). Der Referenz–Host–PC war amd64 mit Linux Debian.

Bei Verwendung der USB–zu–USART–Brücke CP2104 ist die sekundäre Schnittstelle (auf dem Linux–Host–PC) `/dev/ttyUSB0`. Man wird verbunden mit dem UART0 auf dem Pico. February, 2021.

`https://github.com/wa1tnr/camelforth-rp2040-a`

mk

Forth on Sinclair ZX Spectrum Next — vForth1.5

Im letzten Jahr hat Matteo Vitturi sein vForth1.5 auf den *ZX Spectrum Next* gebracht.

Sein Ziel war es, das `screen`– bzw. `BLOCK`–System mithilfe von ZxNextOS–APIs verfügbar zu machen. Er hatte das bereits für ZX–Microdrive und MGT–Disciple–Diskette gemacht. Diese Version sei nun stabil genug, um veröffentlicht zu werden — voila, seht selbst.

Die `screens` werden in einer Datei auf der SD–Karte mit dem Namen `!Blocks.txt` gespeichert, die 16 MB lang ist und 16,383 1–KB–Bildschirme aufnehmen kann (während die MGT–Version und Microdrive 512–Byte–screens — 1,560 bzw. 254 x 7 = 1,778 — aufnehmen können). In dieser neuesten Implementierung besteht ein `screen` aus zwei `BLOCK`s mit jeweils 1024 Byte (endlich) und direktem Lese– / Schreibzugriff über ZxNextOS–APIs.

Es gibt ein ASSEMBLER–Vokabular mit einer Forth–Notation, die auf seinen Wiki–Seiten erläutert wird. Dieser Assembler hat auch die neuesten Op–Codes für die Z80N–Erweiterung.

Über MMU7 wird ein größerer RAM von 1,792 KB zur Verfügung gestellt, auf dem jede 8–KB–Seite à la *Expanded Memory Specification* (EMS) eingebaut werden kann. Für die Speicherung von Zeichenketten gewährt eine HEAP–Einrichtung über FAR– und POINTER–Definitionen Zugriff auf 64 KB Speicherplatz.

Das Wiki in diesem Repository hat schon eine gewisse Form inzwischen. Ich schlage vor, einen Besuch abzustatten.

mk

`https://github.com/mattsteeldue/vforth-next`

`https://github.com/mattsteeldue/forth-next/wiki`

Spectrum Next

> „Who is it for? *The Spectrum Next* is aimed at any Retrogamer out there and Speccy enthusiast who prefers their games, demos and apps running on hardware rather than software emulators, but wants a seamless and simple experience contained within an amazing design.“ (Quelle: `https://www.specnext.com/about/`)

mk

fforth — modern minimal Forth

fforth ist eine minmalistische moderne Forth–Implementierung von Anton Ertl. Es handelt sich um ein 64–Bit–Forth–System mit nativem Code, das auf AMD64 ausgeführt wird. Die Portierung auf andere Architekturen sollte einfach sein. Das Hauptziel von fforth ist es, insbesondere moderne Forth–Implementierungstechniken zu demonstrieren. Dazu gehören

- Header–Strukturen, die beispielsweise die Kompilierungssemantik direkt darstellen
- Parser

Obschon diese Techniken in Gforth bereits demonstriert werden, ist Gforth sehr komplex und die Implementierung nicht einfach zu verstehen aufgrund seines Funktionsumfangs und Portabilität. Und es ist dauernd in Bewegung und daher nicht so gut, um Lehrmaterial darauf aufzubauen. Die Idee ist nun, dass fforth für das moderne Forth sein soll, was *eforth* oder *FIG–Forth* für das altmodische Forth war. Die Ziele von fforth sind also

- Klarheit der Darstellung
- Einfachheit (was zur Klarheit beiträgt)

`https://github.com/AntonErtl/fforth`

mk

Erratum 4d2020–04, S. 26 — Bild stand falsch.

Im vorherigen Heft ist uns leider ein Fehler unterlaufen. Das „Fingerabdruck–Scanner“–Bildchen auf S. 26 gehörte zur nächsten Meldung auf derselben Seite, aber nicht mehr zu Martin Bitters Beitrag. Wir bitten um Verzeihung. Ist beim Schlusslayout verrutscht. mk

Fortsetzung der Leserbriefe auf Seite 20

Warum ergibt `1 chars` den Wert 1?

Anton Ertl

Lange Zeit hatten Zeichen eine fixe Bit–Breite, und mit der Einführung von Unicode schien sich das zunächst auf 16–Bit–Zeichen fortzusetzen, doch dann kamen Unicode 2.0 und UTF–8. In Forth200x wurde für den nächsten Standard festgelegt, dass `1 chars` als Ergebnis 1 hat, was im Normalfall auf UTF–8 hinausläuft. Dieser Artikel erklärt, wie und warum es dazu kam.

Vor Unicode

Beim manuellen Telegraphieren wurden Codierungen mit variabler Länge verwendet, vor allem der internationale Morse–Code von Gerke. Bei der Weiterentwicklung zu automatischen Telex–Maschinen setzte sich dann die ITA2–Variante des Baudot–Codes mit fünf Bits pro Zeichen durch. Ich konnte den Grund dafür nicht herausfinden, aber ich vermute, es ist einfacher, einen Telex–Drucker zu entwickeln, der eine konstante Rate von Zeichen druckt.

1963 kam dann ASCII (sowohl für Telex als auch für Computer) mit 7 Bits pro Zeichen. ASCII kam zwar aus der Telegraphie, wurde aber schnell für Computer übernommen, weil beide Bereiche die gleichen Geräte verwendeten (Drucker und Lochstreifenstanzer). International wurden zunächst die als ISO 646 standardisierten 7–Bit–Codes verwendet, die auf ASCII basieren und einige Sonderzeichen durch sprachspezifische Zeichen ersetzen; wobei ich zwar in den 80ern ab und zu von ISO 646 gelesen habe, aber praktisch nur ASCII erlebt habe. Im Folgenden nenne ich einfach ASCII stellvertretend für alle ISO 646–Codierungen.

Computer der 1950er waren entweder wortadressiert, oft mit 36–Bit–Wörtern ohne von der Architektur vorgegebene Zeichen (z. B. IBM 701 ff.) oder zeichenadressiert ohne von der Architektur vorgegebene Maschinenwörter (z. B. IBM 1401). Auf beiden Arten von Geräten wurden meist 6–Bit–Codierungen für Zeichen eingesetzt, unter anderem BCDIC, auch wenn die wortadressierten Maschinen oft verschiedene Zeichenbreiten zu ähnlichen Kosten erlaubten.

Das IBM System/360 (angekündigt 1964) sollte diese beiden Computerarchitekturstile vereinigen (360 steht für den gesamten Kreis[1]). Es ist eine byte–adressierte Maschine mit 32–Bit–Wörtern. Ursprünglich sollte S/360 ASCII verwenden, aber da IBM noch keine eigene Peripherie für S/360 entwickelt hatten, verwendeten sie auch EBCDIC, ein auf 8 Bit erweitertes BCDIC, und das setzte sich auf S/360 durch.

In der breiteren Computerwelt setzte sich aber ASCII durch, auch z. B. auf dem IBM PC. Auch die Byte–Adressierung und das 8–Bit–Byte setzten sich durch, sodass heutzutage nur mehr Spezialprozessoren wortadressiert sind. Die Verwendung von 8–Bit–Bytes machte es leicht, ASCII zu erweitern, und alle dominanten späteren Codierungen basieren auf ASCII.

So ist KOI–8 (1974) eine 8–Bit–Codierung, bei der die ersten 128 Zeichen ASCII–Zeichen sind und der Rest kyrillische Zeichen enthält. Die ISO–8859–Serie (standardisiert ab 1987) verwendet die Codes 128–255 auch für Zeichen, die in ASCII nicht vorhanden sind. Da ISO–8859 richtig ASCII–kompatibel ist, setzte es sich anders als ISO 646 bei uns durch.

Auch die ostasiatischen Sprachen Chinesisch, Japanisch und Koreanisch, die weit mehr Zeichen haben als in 8 Bits passen, wurden ASCII–kompatibel codiert. Dabei wurde üblicherweise ein Doppel–Byte–Code für die Ideogramme definiert, bei denen das erste Byte im Bereich 128–255 liegt (und meist auch das zweite). Diese Codierung wurde mit ASCII kombiniert; da das erste Byte des Doppel–Byte–Codes sich nicht mit ASCII überschneidet, ist beim Verarbeiten von vorne nach hinten klar, ob es ein ASCII–Zeichen oder ein Doppel–Byte–Zeichen ist. Wenn das zweite Zeichen auch im Bereich 128–255 ist, ist das auch beim Verarbeiten von hinten nach vorne klar. Diese Struktur ist zwar in all diesen Sprachen sehr ähnlich, es gab aber mindestens eine, manchmal mehrere Codierungen pro Land (z. B. GB2312 (China), Big5 (Taiwan), Shift JIS (Japan), EUC–KR (Südkorea)).

Unicode

Was mit den diversen ISO–8859–Codierungen und den diversen ostasiatischen Codierungen aber nicht möglich war: mehrere Sprachen in einem Dokument zu vereinen; und es war auch wenig praktisch bei der Verarbeitung der Dokumente einer Sprache in einem anderen Land. Daher gab es ab 1978 Arbeiten bei Xerox, die dann 1987 in Arbeiten an Unicode mündeten, an dem sich Teilnehmer aus diversen amerikanischen (v. a. kalifornische) Computerfirmen beteiligt haben, die dann das Unicode Consortium bildeten. So entstand 1991 der erste Unicode–Standard.[2]

Die ursprüngliche Idee hinter Unicode [Bec88] ist ein auf 16–Bit verbreitertes ASCII, in dem alle aktuell verwendeten Zeichensätze codiert werden sollten, mit jeweils 16 Bit pro Zeichen. ASCII–Kompatibilität bezüglich der Werte 0–127 wurde als wichtig angesehen, die feste Anzahl an Bits ebenfalls. Da 256 Zeichen nicht reichen und angenommen

[1] Beim Nachfolger S/370 hatte die Marketingabteilung den Ursprung des Namens S/360 offensichtlich vergessen.

[2] Zeitgleich gab es auch eine ISO–Arbeitsgruppe, die am Universal Character Set (UCS) ISO 10646 arbeitete, und die sich dann mit Unicode vereinigte. Ich schreibe im Folgenden nur von Unicode, auch wenn ich den Eindruck habe, dass die Änderungen in Unicode 2.0 den Einfluss der UCS–Gruppe widerspiegelt.

wurde, dass 65536 Zeichen reichen würden (schließlich kamen die gebräuchlichen ostasiatischen Codierungen damals mit <10000 Zeichen aus, wobei sie meist dieselben Zeichen codierten), ergab sich damit automatisch eine 16–Bit–Codierung.

Dieses Grundkonzept hielt bis Unicode 2.0 im Jahr 1996. Dann holte die Komplexität der Schriften in der weiten Welt Unicode ein. Es stellte sich heraus, dass Unicode nicht mit 65536 Zeichen auskommen würde (in Unicode 13.0 sind 143859 „Zeichen“ definiert). Das hätte man noch mit einer Erweiterung auf 32 Bits hinbekommen und das Konzept der festen Anzahl an Bits pro Zeichen gerettet.

Aber als weitere Anforderung kam dazu, dass Zeichen aus mehreren Teilen zusammengesetzt werden können, z. B. ä aus a und ¨. Damit müssen wir zwischen dem (zusammengesetzten) Zeichen[3] und den *Codepunkten (Code point)* unterscheiden, aus denen sie zusammengesetzt sind.[4]

Für unsere Umlaute sind die fertig zusammengesetzten Formen schon in Unicode vorhanden, aber offenbar war diese Strategie auf Dauer nicht gut genug, und daher wurde die Möglichkeit hinzugefügt, Zeichen aus einem normalen Zeichen (z. B. a) und (eventuell mehreren) darauffolgenden *kombinierenden* Codepunkten zusammenzusetzen.

Damit codiert auch eine auf 32 Bits aufgeblasene Codierung ein Zeichen nicht mehr in einer festen Länge von 32 Bits, sondern ein fertiges Zeichen braucht u. U. auch mehr.

Unicode–Codierungen

Die ursprüngliche Codierung in feste 16–Bit–Zeichen (UCS–2) war mit Unicode 2.0 hinfällig. Um die feste Anzahl von Bits pro Codepunkt zu retten, kann man nun auf 32 Bits erweitern (UCS–4 alias UTF–32). Und folgt man den Argumenten, die Unicode 1.0 zu den festen 16–Bit–Codepunkten gebracht hat, müsste man das auch machen. Das wird zwar auch manchmal gemacht, aber eher selten.

Denn es ist sehr aufwändig, ein existierendes Softwareökosystem auf breitere *Codeeinheiten (code unit)* umzustellen, während der Vorteil einer festen Breite sehr gering ist (schon ohne, aber noch mehr durch zusammengesetzte Zeichen), wie wir später noch sehen werden. Daher blieben Systeme wie Windows NT und Java bei 16–Bit–Codeeinheiten, wobei ein Codepunkt aus einer oder zwei Codeeinheiten bestehen kann. Damit wurde aus UCS–2 mit fester Anzahl an Bits pro Codepunkt das UTF–16 mit variabler Anzahl an Bits pro Codepunkt.

Das Gleiche gilt allerdings auch für Software, die mit 8–Bit–Zeichen (ASCII, ISO 8859) hantiert: Der Vorteil einer festen Zeichenbreite ist sehr gering, und die Umstellung auf eine größere Breite (zuerst 16, dann 32 Bits) ist schwierig und kostenintensiv. Daher gab es auch schon vor Unicode 2.0 Bemühungen, Unicode in 8–Bit–Codeeinheiten zu codieren, und der Turingpreisträger Ken Thompson (bekannt für Unix) erfand 1992 UTF–8[5].

UTF–8 hat gewisse Ähnlichkeiten zu den Codierungen für ostasiatische Sprachen vor Unicode: Bytes mit den Werten 0–127 repräsentieren ASCII–Werte, sodass ein ASCII–String automatisch ein UTF–8–String ist. Andere Zeichen werden durch Sequenzen von 2–4 Bytes dargestellt, wobei das erste Byte im Bereich 192–255 ist und die weiteren im Bereich 128–191, sodass man das erste Byte eines Codepunkts immer erkennen kann, sowohl beim Lesen von vorne als auch von hinten. Aus denselben Gründen, warum die UCS–2–basierten Systeme sich ab Unicode 2.0 für UTF–16 entschieden haben, entschieden sich praktisch alle auf 8–Bit–Zeichen basierenden Systeme für UTF–8.

Um die Beziehung zwischen Zeichen, Codepunkten, und Codeeinheiten darzustellen, hier die zwei oben genannten Darstellungen von ä, mit Codeeinheiten für UTF–8:

Zeichen	Codepunkte Unicode	Codeeinheiten UTF–8	zusammengesetzt
ä	U+00E4	`$c3 $a4`	fertig
ä	U+0061 U+0308	`$61 $cc $88`	zerlegt

Praktische Auswirkungen von UTF–8

Der meiste Code arbeitet mit Strings statt mit einzelnen Zeichen, und kann daher genauso gut mit UTF–8–Strings arbeiten wie mit ASCII–Strings oder ISO–8859–Strings. Zum Beispiel ist die Erfahrung, dass Forth–Systeme ohne besondere Anpassung an UTF–8 (z. B. Gforth–0.6, SwiftForth und VFX) zum Großteil richtig funktionieren (auch z. B., wenn man Wortnamen mit Nicht–ASCII–Zeichen definiert und verwendet), mit einigen wenig problematischen Ausnahmen: Das Editieren innerhalb einer Kommandozeile funktiniert nicht richtig; die Fehlermeldungen zeigen den Fehler nicht an der richitigen Stelle im Quellcode; und Case–Insensitivität funktioniert nur für ASCII–Zeichen, wie schon in der vorigen Ausgabe erläutert.

Code, der mit einzelnen Zeichen arbeitet, arbeitet oft mit bestimmten ASCII–Zeichen (z. B. die erlaubten Zeichen bei der Konversion eines Strings zu einer Zahl), und auch dieser Code funktioniert wie gehabt.

Es gibt nur wenig Code, bei dem die variable Breite eines Zeichens eine Rolle spielt, und solchen Code kann man noch seltener machen, indem man, soweit möglich, auf die Verwendung von Strings setzt; z. B. in Forth, indem man ein String mit `type` ausgibt, statt ein Zeichen mit `emit` bzw. `xemit`.[6] Oder indem man in einem String ein

[3] Tatsächlich redet Unicode in diesem Zusammenhang über *extended grapheme cluster*, und verwendet *Zeichen/character* nicht in normativem Text

[4] `https://manishearth.github.io/blog/2017/01/14/stop-ascribing-meaning-to-unicode-code-points/` enthält mehr über zusammengesetzte Zeichen und ihre Handhabung.

[5] Unicode (or Universal Coded Character Set) Transformation Format

[6] In Forth–2012 gibt `emit` ein rohes Byte (bzw. eine Codeeinheit) aus, und `xemit` ein Zeichen (bzw. einen Unicode–Codepunkt).

Zeichen als String mit `search` sucht statt als Zeichen mit `scan`. Als Konsequenz daraus sind die besonderen Wörter für erweiterte Zeichen (wie `xemit`) praktisch nie nötig: `Xemit` (seit 2005 verfügbar) kommt bis jetzt genau einmal im Gforth–Source–Code vor, im Vergleich zu 151 Vorkommen von `type` und 114 Vorkommen von `emit`.

Mir sind nicht viele Anwendungen eingefallen, in denen Einzelzeichen eine Rolle spielen: Überprüfung auf Palindrome oder Anagramme. Es gab auch noch den Vorschlag eines Histogramms der Zeichenhäufigkeit. In all diesen Fällen hilft wegen der Möglichkeit von kombinierenden „Zeichen“ auch UTF–32 nicht.

Unicode bietet Angreifern einige Möglichkeiten, die sie mit ASCII noch nicht hatten, und mit den UTFs kommen noch ein paar dazu. Daher sollte man in Programmen, die nicht vertrauenswürdige Eingaben verarbeiten, diverse Vorsichtsmaßnahmen treffen. Andererseits ist das Abbrechen bei Eingaben, die nicht den Spezifikationen entsprechen (z. B. einer Datei aus einer vertrauenswürdigen Quelle mit einem Mix aus ISO 8859 und UTF–8), vom Benutzer oft auch unerwünscht. Leider ist das alles relativ komplex, und ich habe das Problem noch nicht ernsthaft bearbeitet und daher auch keine Lösung.

Programmiersprachen und –interfaces

Programmiersprachen wie Java und APIs wie das von Windows NT, die beide in den frühen 1990ern entwickelt wurden, setzten voll auf die damalige Idee von Unicode (also UCS–2) und definierten Zeichen als 16–Bit. Als dann Unicode 2.0 kam, waren aber auch sie festgelegt, und erkannten, dass es leichter ist, mit variablen Zeichenbreiten umzugehen als Codeeinheiten breiter zu machen; im Ergebnis unterstützen sie jetzt UTF–16. Weiters unterstützen sie auch noch 8–Bit–basierte Codierungen in verschiedenen Formen, und in *UTF–8 Everywhere*[7] argumentiert ein Windows–Programmierer für die Benutzung von UTF–8. Insbesondere verwenden sowohl Java als auch Windows in Dateien üblicherweise etwas 8–Bit–basiertes.

Die meisten älteren Systeme verschleppten das Thema vorerst, was sich in diesem Fall als sinnvoll erwies, weil 16–Bit–Zeichen nicht die Lösung waren, und es sich dann herausstellte, dass mit UTF–8 der meiste existierende Code ohne oder mit wenigen Änderungen funktioniert.

C's `char` ist ein Byte[8], aber schon C89 standardisierte `wchar_t`, um breitere Zeichen zu unterstützen; Funktionen für diesen Typ fangen üblicherweise mit `wc` an oder enthalten `wc`. Und C89 standardisierte auch variabel breite Zeichendarstellungen (*multi–byte characters* in der C–Terminologie) in Form von Multi–Byte–Funktionen, deren Name üblicherweise mit `mb` anfängt.

Die sinnvolle Abbildung dieser Konzepte auf Unicode ist, `wchar_t` 32 Bits breit zu machen, und/oder Unicode–Codepunkte in Form von UTF–8 als multi–byte characters zu repräsentieren.

Unter Unix ist `wchar_t` tatsächlich 32–Bit breit, wird aber kaum verwendet. Stattdessen wird mit UTF–8–Strings gearbeitet, und an den wenigen Stellen, wo das nötig ist, mit den Multi–Byte–Funktionen.

Windows legte sich, wie oben beschrieben, breitflächig auf 16–Bit–Codeeinheiten fest, und daher ist `wchar_t` in Windows 16 Bits breit. Es gibt allerdings in C keine standardisierte Unterstützung für variabel breite Codierungen, die mit breiteren Codeeinheiten als Bytes arbeiten, also keine Unterstützung für UTF–16.

Einen ungewöhnlichen Weg ging Python: Während Python 2 noch mit Byte–Strings hantierte, wurden in Python 3 Unicode–Strings eingeführt, die aus Unicode–Codepunkten bestehen, die in Python Characters genannt werden. Diese Strings werden intern entweder als Arrays von Bytes, Arrays von 16–Bit–Codepunkten oder Arrays von 32–Bit–Codepunkten dargestellt, wobei ein einziger Codepunkt ab 65536 zu dem 32–Bit–Array führt, und ein einziger Codepunkt ab 256 zu mindestens dem 16–Bit–Array. Allerdings führte das zu Problemen, die auch über zehn Jahre nach der Einführung von Python 3 zu langen Diskussionen[9] führen und dieses Modell ist daher wohl eher als Beispiel zu sehen, wie man es nicht macht.

Obwohl ich nicht Python programmiere, haben diese Probleme von Python–Programmen auch mir schon praktische Schwierigkeiten gemacht: In Dateien, die mehrere Vorgängerdateien mit diversen Daten kombinierten, tauchten sowohl ISO–8859–1–Strings als auch UTF–8–Strings auf, und das von jemand anderem geschriebene Python–Programm verweigerte den Dienst. Ich habe das dann mit einem Forth–Programm gelöst, das solche Misch–Dateien in UTF–8 umgewandelt hat, bevor sie an das Python–Programm verfüttert wurden.

Und in Forth?

Während Forth–83 noch auf byte–adressierte 16–Bit–Maschinen beschränkt war, führte Forth–94 (alias ANS Forth) Abstraktionen ein, die diese System–Einschränkung aufhoben: *address units* repräsentieren die kleinste adressierbare Einheit (es war auch von Hardware die Rede, die 4–Bit–Einheiten adressierte), und *characters* können aus einer festgelegten Anzahl an address units bestehen, mindestens aber 8 Bits. Das war auch dazu gedacht, um die damals scheinbar heranstürmende Welt der 16–Bit–Zeichen zu unterstützen, und Jack Woehr hat mit jaxForth für Windows NT auch ein System mit 16–Bit–Zeichen implementiert, in dem `1 chars` 2 produziert; allerdings wurde jaxForth nicht breit verwendet. Der Großteil der Forth–Programmierer ignorierte diese Entwicklung und arbeitete weiter mit der Annahme, dass `1 chars` 1 produziert; auf allen verbreitet genutzten Forth–Systemen ist diese Annahme wahr, sodass die

[7] `https://utf8everywhere.org/`, auch sonst eine gute Quelle, um sich über die Unicode–Problematik zu informieren

[8] auf byte–adressierter Hardware

[9] `https://gregoryszorc.com/blog/2020/01/13/mercurial%27s-journey-to-and-reflections-on-python-3/`
`https://lwn.net/Articles/809336/`

Forth–Programmierer auch nicht durch Testen herausfinden können, ob sich ihr Code auf diese Annahme verlässt, selbst wenn sie es wollen.

Immer wieder wurde das Thema Unicode diskutiert, und schließlich nahm ich mich 2005 des Themas an, erkannte, dass die feste Zeichenbreite, der ursprünglich auch ich anhing, nicht wichtig ist, und machte zusammen mit Bernd Paysan einen Vorschlag für die Unterstützung variabel breiter Zeichen [EP05], der nach einigen Änderungen 2010 in Forth 200x aufgenommen wurde und in Forth–2012 als *Extended–Character word set* erschien.

Das Prinzip dieser Erweiterung ist, dass im Speicher ein Codepunkt mehrere Forth–chars einnehmen darf, wobei ein Forth–char[10] nicht einem Zeichen oder Codepunkt entspricht, sondern einer Codeeinheit. Z. B. könnte ein Forth–char ein Byte sein, und die Codierung UTF–8. Auch ein 16–Bit Forth–char mit UTF–16 ist Forth–2012–konform.[11]

Auf dem Stack ist ein Codepunkt eine Zelle, üblicherweise als `xchar` bezeichnet, und auch da müssen ASCII–Zeichen ihre Werte behalten. Forth–2012 schreibt jenseits von ASCII nichts vor, sodass die verschiedensten Codierungen mit den xchar–Wörtern von Forth–2012 verwendet werden können, natürlich ASCII, Latin–1, und UTF–8, aber auch z. B. ältere Codierungen für ostasiatische Sprachen.[12] Ein interessanter Ansatz von Peter Fälth ist, die Bytes eines Codepunkts aus der UTF–8–Codierung einfach undekodiert in einer Stack–Zelle zu haben, was auch konform ist, wenn das einzelne Byte eines ASCII–Zeichens auf dem niederwertigsten Byte der Zelle landet.

Umgekehrt garantiert Forth–2012 auch nichts jenseits von ASCII. Es ist daher noch nicht standardisiert, wie der Programmierer dem System sagt, wie das Programm codiert ist, und eventuell, in welcher Codierung es die Eingabe erwartet; genauso für den Benutzer des Programms. Gforth verwendet dafür, wie unter Unix üblich, das Environment. Interessanterweise sind dazu bisher kaum Anfragen gekommen; da aufgrund des langsamen Aussterbens von anderen Codierungen als UTF–8 das Problem im Laufe der Zeit wohl eher kleiner als größer wird, kann man vielleicht in ein paar Jahrzehnten einfach standardisieren, dass die Programme und Daten alle in UTF–8 sind.

Das xchar–Wordset standardisiert 10 neue Wörter im Kern, 8 weitere neue Wörter in den optionalen Erweiterungen, und definiert 3 Wörter (`char [char] parse`) neu, sie arbeiten jetzt mit xchar statt char. Dabei haben wir es vielleicht etwas mit der Anzahl an Wörtern übertrieben, um nur ja nicht ein wichtiges Wort auszulassen, und wohl, weil wir damals noch davon ausgegangen sind, dass man diese Wörter häufig brauchen würde. Tatsächlich braucht man sie selten, weil man eben selten mit einzelnen Codepunkten arbeiten will und für die Arbeit mit Strings die herkömmlichen Wörter reichen. So gibt es bisher folgende Anzahlen von Verwendungen dieser Wörter in Gforth:

xchar		String–Alternative	
3	`x-size`		
1	`xc!+`	43	`move`
3	`xc!+?`	43	`move`
5	`xc,`	9	`mem,`
5	`xc-size`	114	`nip`
5	`xc@+`	43	`move`
4	`xchar+`		
1	`xemit`	151	`type`
2	`xkey`		
1	`xkey?`		
xchar ext		**String–Alternative**	
2	`+x/string`	71	`/string`
0	`-trailing-garbage`		
1	`ekey>xchar`		
6	`x-width`		
3	`xc-width`	6	`x-width`
6	`xchar-`		
0	`xhold`	4	`holds`
1	`x\string-`	71	`/string`

Viele der Benutzungen der xchar–Wörter sind im Kommandozeilen–Editor von Gforth. Die String–Alternativen zeigen, was man statt der xchar–Wörter verwenden kann; die String–Verwendungen sind in den meisten Fällen nicht dazu da, um einen einzelnen xchar in String–Form zu verarbeiten, aber sie geben einen Eindruck, wie selten die xchar–Wörter im Vergleich zu anderen Wörtern im selben Bereich verwendet werden.

Von Verwendungen anderswo habe ich nichts gehört, obwohl es durchaus prominente Beispiele für international verbreitete Forth–Anwendungen gibt. Das kann bedeuten, dass die Wörter keine Probleme machen, oder dass die Programmierer diese Wörter gar nicht brauchen, weil String–Wörter gut genug sind, oder weil sie schon vorher ihre eigenen Wörter entwickelt haben und jetzt weiter verwenden.

Was derzeit im xchar–Wordset fehlt, ist: Unterstützung jenseits von Codepunkten, also für zusammengesetzte Zeichen. Unterstützung für Dinge wie die Konversion verschiedener Darstellungen des gleichen Zeichens in dieselbe Darstellung (Normalisierung), und die Validierung von Byte–Strings als gültige Strings der Codierung. Allerdings gab es bisher auch keine Anfragen oder Eigenbedarf dazu.

Mit den xchar–Wörtern gibt es jetzt noch weniger Grund als zuvor, dass auf byte–adressierten Maschinen ein char größer als ein Byte sein sollte. Maschinen mit kleineren Adress–Einheiten spielen für Standard–Forth–Systeme

[10] Im Standard–Text an manchen Stellen auch pchar genannt; anderswo im Standard wird aber überall char verwendet, und ich mache das genauso; wenn etwas anderes gemeint ist, steht explizit *xchar*.

[11] Allerdings ist ASCII–Kompatibilität vorgeschrieben, sodass 8–Bit–chars mit UTF–16 nicht gehen, weil ein 8–Bit–ASCII–String für `abc` nicht das Gleiche ist wie ein UTF–16–String für `abc`.

[12] Das optionale `xchar-` funktioniert nur auf Encodings, in denen der Anfang eines Zeichens auch bei der Verarbeitung im Rückwärtsgang erkannt wird.

auch keine Rolle (und auch sonst nicht mehr). In Forth–Systemen ist es weiterhin allgemeine Praxis, dass `1 chars` 1 produziert, und in vielen Programmen wird absichtlich angenommen, dass das der Fall ist (und in den meisten anderen ist diese Annahme wohl unabsichtlich an der einen oder anderen Stelle durchgerutscht). Daher hat das Forth–200x–Komitee 2016 den Vorschlag[13] angenommen, das zu standardisieren.

Zusammenfassung

Auch mit Unicode kann man dank UTF–8 weiterhin mit 8–Bit–chars arbeiten, und der Großteil des Codes funktioniert unverändert. Für die wenigen Stellen, an denen eine Änderung nötig ist, stellen die xchar–Wörter Funktionalität zur Verfügung, um mit Codepunkten umzugehen. Diese Wörter reduzieren den ohnehin schon geringen Bedarf nach chars mit mehr als einem Byte weiter, und der nächste Forth–Standard legt daher `1 chars`=1 fest.

Referenzen

[Bec88] Joseph D. Becker. Unicode 88. Reprint, Unicode Consortium, 1988.

[EP05] M. Anton Ertl and Bernd Paysan. Xchars or Unicode in Forth. In M. Anton Ertl, editor, *21st EuroForth Conference*, pages 16–20, 2005.

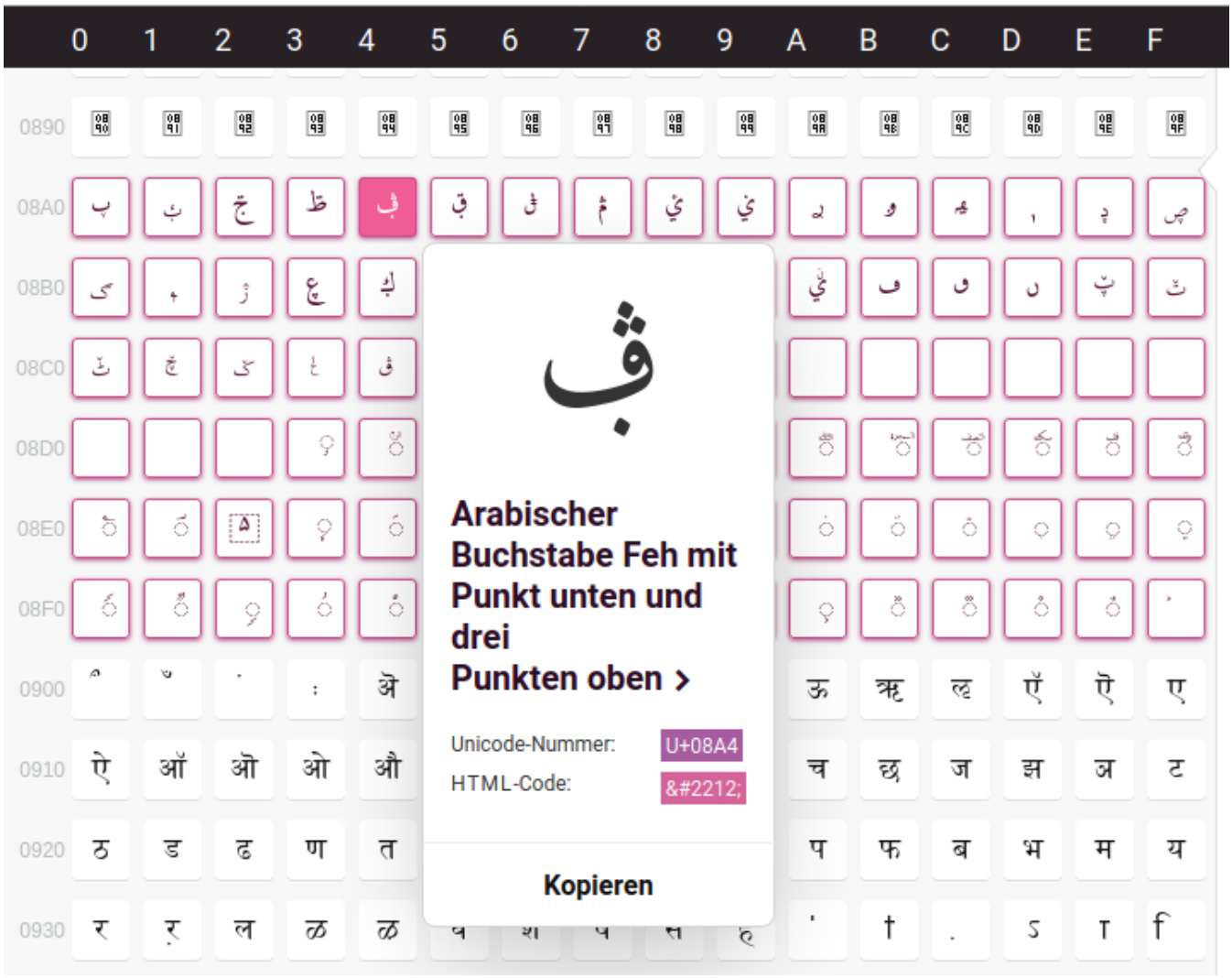


Abbildung 1: Unicode–Tabelle, zum Beispiel: Arabisch, erweitert–A

Quelle: `https://unicode-table.com/de/#08A4`

[13] `http://www.forth200x.org/char-is-1.html`

[brackets], Parser and Recognizers

Klaus Schleisiek

It happened when I developed volksForth in the late 1980's after serving on the pre–ANSI 1983 standards team. State smart words had fallen into disgrace, because the circumstances under which they would fail did become more and more clear. Therefore, I wanted to be smart and I introduced the new deferred word `Parser`.

`Parser` was supposed to either *interpret* or *compile* the source code depending on the system's state and the definitions for `[` and `]` looked like this:

```
: [ ( -- )   State off
   ['] interpreter IS Parser
; immediate

: ] ( -- )   State on
   ['] compiler IS Parser
;
```

So I did not need `State` any more (except for legacy code) and therefore, at least `find` and `interpret` would no longer have to look at `State`. And because *Gforth* was based on volksForth, Gforth inherited this very construct. In retrospect, that was a terrible mistake, which did not become apparent some 30 years later when I wrote the cross–compiler for microCore based on Gforth.

In the cross–compiler, two *totally different sets* of interpreter/compiler are needed depending on whether I am producing code for the target system, or whether I am extending the cross–compiler on the host. In principle, this problem could be solved with `Parser` elegantly. But whenever I would execute `[` or `]` in the target context, this would throw me back into the host's interpreter/compiler.

Therefore, I propose that Gforth adopts the following way of doing things:

```
: [ ( -- ) State off ; immediate

: ] ( -- ) State on ;

: host-parser ( -- )   State @
   IF compiler ELSE interpreter THEN
; ' host-parser IS Parser
```

Now `Parser` may actually be used to change the interpretation/compilation context as needed.

But "State should not be used" is one of the strongly held beliefs in the Forth community today. But what is wrong with the use I make of it above? I am sure: nothing. This is just a situation where the use of State does not lead to unexpected results.

What does this have to do with *recognizers*, you may ask. It is definitely a structure that is above recognizers. These come into play when defining the compiler/interpreter. And therefore, besides introducing recognizers, we should first get the super structure right.

As I learned on occasion of the virtual Forth meeting alongside the *r3C Chaos Computer Congress 2020*, the development of recognizers is not yet finished. My simple approach as proposed in *Poor Man's Recognizer*[1] was rejected on the grounds that it would not be capable of all the manipulations one needs to perform on and with recognizers. Therefore, I propose that we agree on a *set of requirements* for recognizers. These are the ones that come to my mind:

- Make the interpreter/compiler loop more readable.
- Make it easier to modify the interpreter/compiler loop.
- Recognizers should support the use of multiple code-fields associated with a colon definition depending on wheter we are interpreting (State = 0), compiling (State = 1), or postponing a word.

I think that so far we are in agreement. But how important is it to modify the interpreter/compiler itself on the fly e.g. in order to include floating point number formats for number input when loading a floating point package.

I venture to say that "on the fly" modification of the sequence of recognizers is not really needed. Instead, another static set of recognizers could be compiled and activated by assigning it to `Parser` when using the structure discussed above.

[1] see VD3/2020

Poor Man's Case

Klaus Schleisiek

I invented "my" case construct many years ago when I had to decode ASCII keys to trigger specific actions of a simple appliance. At the time I looked at the proposals for `CASE ... OF ... ENDOF ... ENDCASE` *type constructs and they all seemed too complicated to me. Again, making use of the return stack simplifies things considerably.*

Instead of compiling the code sequentially having to jump from each `ENDOF` to `ENDCASE`, we just split up the code hierarchically, and replace the branch to `ENDCASE` by a simple `EXIT`. As a consequence, we get away without having to introduce any new compiling word at all!

Example

Here is a very dumb terminal for testing e.g. an *RS232 interface*: First we define a few control keys.

```
$08 Constant #BS
$0A Constant #LF
$0C Constant #CR
$1B Constant #ESC
```

Followed by the key decoder. For simplicity, it only takes care of *backspace*, *linefeed* and *carriage return* for special actions. All other keys will be displayed. And we maintain a keystrokes counter.

```
Variable Keystrokes

: backspace ( --)
  #BS emit space #BS emit
  -1 Keystrokes +! ;

: decode ( key -- )
  #BS case? IF backspace EXIT THEN
  #CR case? IF EXIT THEN
  #LF case? IF cr EXIT THEN
  emit ;
```

Please note that at the end of each single case we return to `dumb-terminal` via `EXIT`, increment the keystrokes counter and wait for the next key.

```
: dumb-terminal ( -- )
  BEGIN key
        #ESC case? IF EXIT THEN
        decode
        1 Keystrokes +!
  AGAIN ;
```

The phrase `1 Keystrokes +!` is the code that would traditionally be placed after `ENDCASE` and `decode` handles all of the traditional `OF ... ENDOF` cases as a separate word. And when we hit the escape key, we leave `dumb-terminal` and return to Forth's interpreter.

CASE?

And here comes the magic word, which we need to define in order to make our code readable:

```
: case? ( n1 n2 -- n1 ff | tf )
  over = dup IF nip THEN ;
```

When `n1` equals `n2`, we just leave a `true flag`. Otherwise, the key's value `n1` remains on the stack and we push a `false flag`.

I venture to say that this is the simplest case type construct that one can possibly come up with. I never felt the need to use any other case type construct.

Switch

```
$day = 3
    switch ( $day )
    {
        0 { $result = 'Sunday'    }
        1 { $result = 'Monday'    }
        2 { $result = 'Tuesday'   }
        3 { $result = 'Wednesday' }
        4 { $result = 'Thursday'  }
        5 { $result = 'Friday'    }
        6 { $result = 'Saturday'  }
    }
    $result
    # Output
    'Wednesday'
```

Da begegnete mir doch bei der Suche nach etwas Brauchbarem für den verbliebenen Rest auf dieser Seite nebenstehende merkwürdige Passage im Web. Ist das nicht ganz ähnlich wie unser `case` in Forth?

> Switch statement — The switch statement allows you to provide a variable and a list of possible values. If the value matches the variable, then it's scriptblock will be executed.
>
> For this example, the value of `$day` matches one of the numeric values, then the correct name will be assigned to `$result`. We are only doing a variable assignment in this example, but any PowerShell can be executed in those script blocks.

Was mag das wohl sein? mk

`https://powershellexplained.com/2018-01-12-Powershell-switch-statement/`

Ein Forth–Rätsel: [IF] . . .

Michael Kalus

Hoffentlich findet ihr Vergnügen an diesem Rätsel.

Angeregt durch den Artikel *Conditional Compilation*[1] hat Klaus Schleisiek sein `[if] [else] [then]` und das `[notif]` freundlicherweise auch publik gemacht. Sein `case?` spielt darin eine zentrale Rolle. Ihr habt es ja soeben kennengelernt. Der Code ist doch übersichtlich und nachvollziehbar, oder?

Aber was genau passiert denn da, was mutet Klaus uns da zu?

- Warum werden `[if] [else] [then]` als *deferred words* angelegt?
- Wieso kommt die Suchschleife von `[if]` oder `[else]` denn zum Ende?
- Und wieso werden `\` und `(` und `(*` nochmal *postponed*, nachdem sie im Rahmen dieser Suche an so einer bedingten Stelle in einem Quellcode vorgekommen sind?

Fragen über Fragen. Und vielleicht kommen euch ja noch ganz andere. Zuschriften dazu bringen wir gern.

Und nun auf, auf, hier kommt es . . .

Das rätselhafte Listing

```
1   \ ------------------------------------------------------------------------
2   \ Conditional compilation, Version 3
3   \ ------------------------------------------------------------------------
4
5   Defer [IF]    ( flag -- )  immediate
6       : [NOTIF] ( flag -- )  0= postpone [IF] ; immediate
7   Defer [ELSE]  ( -- )       immediate
8   Defer [THEN]  ( -- )       immediate
9
10  : <eof> ( -- ) ; \ used to signal "end of file"
11
12  : forth-word  ( -- xt )
13     BEGIN BEGIN  parse-word ?dup
14           WHILE  forth-wordlist search-wordlist ?EXIT
15           REPEAT
16           drop   refill 0=
17     UNTIL  ['] <eof>
18  ;
19  : *)   ( -- )  ; \ used to signal "end of multi-line comment"
20
21  : (*   ( -- )   BEGIN  forth-word dup ['] <eof> = swap ['] *) = or UNTIL ; immediate
22
23  Variable Level   0 Level !
24
25  : level-1  ( -- )  Level @ 1 - 0 max Level ! ;
26
27  : [if]-decode  ( xt -- flag )
28     ['] [IF]    case? IF  1 Level +!     false  EXIT THEN
29     ['] [NOTIF] case? IF  1 Level +!     false  EXIT THEN
30     ['] [ELSE]  case? IF  Level @ 0=             EXIT THEN
31     ['] [THEN]  case? IF  Level @ 0=   level-1  EXIT THEN
32     ['] \       case? IF  postpone \     false  EXIT THEN  \ needed to be able to e.g. comment out [THEN]
33     ['] (       case? IF  postpone (     false  EXIT THEN  \ needed to be able to e.g. comment out [THEN]
34     ['] (*      case? IF  postpone (*    false  EXIT THEN  \ needed to be able to e.g. comment out [THEN]
35     ['] <eof> = abort" [THEN] missing"
36     \ all oter xt's are just ignored
37     false
38  ;
39
40  :noname  ( flag -- ) ?EXIT BEGIN  forth-word [if]-decode UNTIL ; IS [IF]
41
42  :noname  ( --      )       BEGIN  forth-word [if]-decode UNTIL ; IS [ELSE]
43
44  :noname  ( --      )       level-1 ;                             IS [THEN]
45
```

[1] Heft 4/2020

Angenehmes Blinken

Matthias Koch, Robert Clausecker

Wer kennt es nicht, das hektische An–Aus–Blinken einer Leuchtdiode. Was den Programmierer beim ersten gelungenen „Hallo Welt“ auf einem neuen Mikrocontroller freut, ist für die meisten Menschen aber nicht schön anzusehen. Aus diesem Grund haben wir in den letzten Tagen einmal über angenehmes Blinken nachgedacht. Es sollte ein sanftes, von der Helligkeit her langsam steigendes und wieder fallendes Blinken sein, ein Effekt, der ein bisschen wie das Atemholen wirkt.

Das menschliche Auge nimmt den Logarithmus der Lichtintensität wahr — um also einen scheinbar gleichmäßig an– und absteigenden Helligkeitsverlauf zu erhalten, muss die Helligkeit einer exponentiellen Kurve folgen.

Das angenehme Blinken besteht also aus einem Rampengenerator, einer Exponentialfunktion und einer Möglichkeit, die so berechneten Werte als Intensität auszugeben.

ARM

Beginnen wir mit einer Implementierung für einen ARM Cortex M0 — Listing 1.

Einen Aufwärts–Zähler über den Absolutwert in eine Dreiecksfunktion zu verwandeln ist ein alter Trick, der einen festen Platz im Nähkästchen eines Zahlenjongleurs haben sollte. Drumherum wird einfach nur passend geschoben: davor nach links, um die Geschwindigkeit anzupassen — denn bis ein 32–Bit–Register von selbst überläuft, dauert es eine Weile — und anschließend mit Vorzeichen nach rechts, um die Amplitude der Dreiecksfunktion in eine passende Größenordnung für die Skalierung zu bringen.

Die Skalierung bildet die Werte von 0 bis 65536 auf den gewünschten Bereich von 117 bis 231 ab, wobei die Größe aller Werte und die Schubweiten so beschaffen sind, dass alles bequem in ein 32–Bit–Register passt. Die Skalierung ist nötig, da eine 32*32=64 Multiplikation (`umull`, unsigned multiply long) erst ab dem ARMv7–M–Befehlssatz, und damit nicht auf dem Cortex M0 bzw. M0+ zur Verfügung steht.

In der fürs Blinken vereinfachten und heruntergebrochenen Bitexp–Funktion werden die untersten drei Bits maskiert, mit einem führenden 1–Bit versehen und soweit nach links geschoben, wie es die fünf folgenden Bits in dem Eingabe–Byte angeben. Das Format der Eingabe–Bytes ist dabei `%EEEEEMMM`, und es wird `%1MMM << %EEEEE` berechnet. Der Maximalwert `231 = $E7 = %11100_111` ergibt sich aus `%1_111 << %11100 = 15 << 28` und beträgt `$F0000000`. Noch größere Eingabewerte führen zum Überlauf und ergeben dann wieder kleinere Ausgabewerte.

Die vollständige Bitexp–Funktion liegt Mecrisp–Stellaris und Mecrisp–Quintus jeweils in `common/bitlog.txt` bei. Der Unterschied ist, dass diese Funktion sich bei kleinen Eingabewerten unter 16 linear verhält, was als „Anlaufphase“ oft praktisch ist, und bei zu großen Eingabewerten in Sättigung geht.

So sieht sie aus:

```
: bitexp ( u -- u )
  \ Returns an integer value equivalent to
  \ the exponential. For numbers > 16,
  \ bitexp(x) approx = 2^(x/8 + 1)
  \ B(E(x)) = x for 16 <= x <= 247.
  dup 247 u> \ Overflow ?
  if drop $F0000000
  else dup 16 u<=
    if 1 rshift
    else
      dup ( u u )
      7 and 8 or ( u b )
      swap ( b u )
      3 rshift 2 - lshift
    then
  then
  1-foldable ;
```

Nur fürs Blinken aber brauchen wir diese Feinheiten nicht und können so mit sehr wenigen Opcodes auskommen.

Bei einem Sigma–Delta–Modulator wird der gewünschte Steuerwert in ein Register addiert. Bei einem Überlauf dieses sogenannten Phasenakkumulators wird der Ausgang für einen Zyklus lang gesetzt. Je größer der Steuerwert ist, desto schneller und häufiger läuft der Phasenakkumulator über. Im Vergleich zur Pulsbreitenmodulation besteht der Vorteil darin, dass der Ausgang bei einer Änderung des Steuerwertes nicht springt oder hakt, und dass die einzelnen, schmalen Pulse gleichmäßig über die Zeit verteilt sind.

Spontan wäre eine Addition in den Phasenakkumulator die erste Wahl — dann wäre das Carry–Flag gut verständlich beim Überlauf gesetzt, und ansonsten nicht. Nun gibt es allerdings den praktischen Trick `sbcs r3, r3` — also Subtraktion eines Registers von sich selbst, aber mit Carry. Das ergibt saubere Forth–Flags: `0`, falls Carry gesetzt gewesen ist oder `-1`, falls Carry gelöscht gewesen ist. Da in diesem Falle aber die Leuchtdiode gerade bei einem Überlauf des Phasenakkumulators für einen Moment leuchten und ansonsten dunkel sein soll, wird hier vom Phasenakkumulator abgezogen, was (zumindest in der ARM–Architektur) die Bedeutung des Carry–Flags umkehrt. Und ob der Phasenakkumulator nun vorwärts oder rückwärts zählt, spielt für diesen Zweck keine Rolle. Sollte die Leuchtdiode mit einer `0` in ihrem Register leuchten, dann ist einfach nur `subs` durch `adds` auszutauschen.

Eine noch offene Frage ist bestimmt, wieso mit `117 = %1110_101` als Eingabewert für die Exponentialfunktion begonnen wird, was also einem Steuerwert von `%1_101 << %1110 = 13 << 14 = $00034000` entspricht, obwohl die Exponentialfunktion doch auch sauber bis runter auf Null reichen würde. Der Grund ist ganz einfach: So eine Leuchtdiode an einem Sigma–Delta–Modulator–Ausgang hat einfach nicht so wirklich 32 Bit Dynamikumfang. Werden die Steuerwerte zu klein, dann liegen die einzelnen Zyklen, in denen die Leuchtdiode eingeschaltet ist (Überlauf!) zeitlich soweit auseinander, dass der Mensch die einzelnen Blitze separat sehen kann. Und das sieht nicht schön aus: Runterdimmen, Blitz, Blitz, Blitz, Raufdimmen … Deshalb muss mit einer „Mindesthelligkeit“ begonnen werden. Bei einer Pulsbreitenmodulation mit 32 Bit Auflösung würde übrigens ein anderes Problem auftauchen: Da wären die An– und Aus–Zeiten so lang, dass es bei realistischen Taktfrequenzen des Mikrocontrollers mit jedem Steuerwert einzeln sichtbare „An“ und „Aus“ geben würde.

Das Schöne an dieser Routine insgesamt ist, dass sie sehr einfach ist und prima in Warteschleifen eingefügt werden kann, wenn zum Beispiel auf einen Knopfdruck gewartet wird und der Mensch am anderen Ende sanft davon in Kenntnis gesetzt werden soll. Sie zeigt aber auch, wie viele Ideen in so wenigen Instruktionen versteckt sein können.

RISC–V

Lasst uns diese Routine noch einmal auf einem RISC–V Prozessor in der Geschmacksrichtung RV32IM betrachten. Dort ist es sogar noch mit einigen Opcodes weniger möglich — Listing 2.

Die Funktionsweise ist der bereits besprochenen Variante sehr ähnlich, aber doch gibt es einige Unterschiede. Zunächst einmal beginnt es mit dem geschwindigkeitsbestimmenden Linksschub, aber es wird die volle Amplitude der Dreiecksfunktion verwendet. Im RISC–V steht nämlich `mulhu` (multiply high unsigned) zur Verfügung, also der obere Teil von `um*`, und so lässt sich die Skalierung der Amplitude, die beim M0 mit zwei zusätzlichen Schiebebefehlen bewerkstelligt wurde, gleich in einem Rutsch erledigen. Das ist übrigens ein Trick, der sich bestens für die Skalierung von Werten aus einem Analog–Digital–Wandler eignet: Passende Konstante, `um*`, Offset draufaddieren, fertig.

Diesmal hat der Sigma–Delta–Modulator den intuitiv richtigen Addierbefehl, aber da es in RISC–V *keine* Flags gibt, wird der Überlauf eben mit einem `sltu` (set less than, unsigned) erkannt. Dieser ergibt dann `1` bei Überlauf des Phasenakkumulators und `0` sonst. Wer ein wohlgeformtes Ergebnis möchte, kann noch ein `sub x13, zero, x13 (0, 1 --> 0, -1)` oder ein `addi x13, x13, -1 (0, 1 --> -1, 0)` hinzufügen, je nachdem, welche Polarität gewünscht wird.

Und siehe da: Es blinkt so richtig angenehm.

Mecrisp

Zum Ausprobieren hier gleich noch eine Variante, die direkt in Forth geschrieben ist und eine Zellengröße von 32 Bits benötigt — Listing 3. Vergleicht und experimentiert selbst.

Listing 1

```
1   @ Assembler Quellcode - ARM Cortex M0
2
3     ldr r2, =leds @ Dies sei der LED-Register
4
5   @ ------------------------------------------------------------------------
6   breathe_led: @ Generate smooth breathing LED effect
7   @ ------------------------------------------------------------------------
8
9       @ Register usage:
10
11      @ r0 : Unused
12      @ r1 : Scratch
13      @ r2 : Initialised with IO address for GPIO
14      @ r3 : Scratch
15      @ r4 : Unused
16      @ r5 : Delay counter
17      @ r6 : Slow triangle counter, clipped
18      @ r7 : Phase accumulator for sigma-delta modulator
19
20      adds r5, 1                  @ Delay counter
21
22      lsls r6, r5, 12             @ Select blinking speed here.
23      asrs r6, r6, 15             @ Triangle generator between 0 and 0x10000.
24      bpl  1f                     @ Idea:
25      negs r6, r6                 @   abs((upcounter << 12) >>> 15)
26  1:
27      movs r1, 231-117            @ Scale range:
28      muls r6, r1                 @ (high-low) * x / 65536 + low
29      lsrs r6, 16                 @
30      adds r6, 117                @ Baseline minimum brightness
```

```
31
32      movs r3, 7                  @ Simplified bitexp function.
33      ands r3, r6                 @   Valid for inputs up to 231
34      adds r3, 8                  @   Gives too small values above 231
35      lsrs r1, r6, 3              @   Input  in r6 is kept
36      lsls r3, r1                 @   Output in r3
37
38      subs r7, r3                 @ Sigma-Delta phase accumulator
39      sbcs r3, r3                 @ Sigma-Delta output through carry, which is inverted for subs
40
41     str  r3, [r2]                @ Output in all bits at once
42     b.n breathe_led
43
```

Listing 2

```
1   # Assembler Quellcode - RISC-V RV32IM
2
3     li x8, leds # Dies sei der LED-Register
4
5   # ------------------------------------------------------------------------
6   breathe_led: # Generate smooth breathing LED effect
7   # ------------------------------------------------------------------------
8
9       # Register usage:
10
11      # x8  : Initialised with IO address for GPIO
12      # x9  : Phase accumulator for sigma-delta modulator
13      # x10 : Delay counter
14      # x11 : Slow triangle counter, clipped
15      # x12 : Scratch
16      # x13 : Scratch
17
18      addi x10, x10, 1            # Delay counter
19
20      slli x11, x10, 12           # Select blinking speed here.
21      bge  x11, zero, 1f          # Triangle generator between 0 and 0x80000000
22      sub  x11, zero, x11         # Idea: abs(upcounter << 12)
23  1:
24      li   x12, 2*(231-117)       # Scale range:
25      mulhu x11, x11, x12         # (high-low) * x * 2 / $80000000 + low
26      addi x11, x11, 117          # Baseline minimum brightness
27
28      andi x13, x11, 7            # Simplified bitexp function.
29      addi x13, x13, 8            #   Valid for inputs up to 231
30      srli x11, x11, 3            #   Gives too small values above 231
31      sll  x13, x13, x11          # Input in x11, output in x13
32
33      add  x9, x9, x13            # Sigma-Delta phase accumulator
34      sltu x13, x9, x13           # Sigma-Delta output on overflow
35
36     sw  x13, 0(x8)               # Output in least significant bit
37     j breathe_led
38
```

Listing 3

```
1   \ Forth Quellcode - Mecrisp
2
3   0 variable sdm-phase
4
5   : >sdm ( u -- )
6     sdm-phase @ ( u alt )
7     tuck +      ( alt neu )
8     dup sdm-phase !
9     u> led-out-gpio !
10  ;
11
12  : atemblinker ( -- )
13
14    0 \ Startwert für den Zähler: LED beginnt dunkel
15
16    begin
17      1+ dup         \ Zähler laufen lassen
18
```

```
19      12  lshift     \ Hiermit wird die Blinkgeschwindigkeit eingestellt
20
21      15 arshift abs \ Ergibt eine Dreiecksfunktion, die von 0 bis $10000 läuft
22
23      231 117 - *    \ Die Dreiecksfunktion
24      16 rshift      \   auf Werte zwischen
25          117 +      \   117 und 231 skalieren
26
27      dup 7 and 8 +  \ Vereinfachte Bitexp-Funktion,
28      swap 3 rshift  \   gültig für Eingabewerte
29      lshift         \   von 0 bis einschließlich 231
30
31      >sdm
32
33    key? until
34    drop
35
36    0 led-out-gpio ! \ Leuchtdiode wieder ausschalten
37  ;
38
```

Fortsetzung der Leserbriefe von Seite 8

Unser SWAP–Drache sucht einen neuen Drachenhüter.

Tja, hier sitze ich nun bei Michael im Regal auf 'ner Dose Schiffszwieback und studiere fleißig … die Geschichte der Seefahrt.

Hier steht nämlich neben mir, auf unserer Schatzkiste, ein Modell der „Sciabecco Mediterraneo“, ein Dreimast–Segelschiff mit lateinischem Segelsystem. Mit einer Länge von 40 m schon ein imposanter Typ. So ab 1700 n. Chr. wurden sie mehr und mehr gebaut, und bewaffnet, gegen die Piraten im spanischen Raum. Hat gedauert, bis die Magherbina–Piraterie endlich gezähmt wurde.

Eigentlich ist die *Sciabecco* ein Schiffstyp arabischen Ursprungs, der seit 1300 im Mittelmeerraum eingesetzt wurde. Laut arabischen Quellen erscheint dieses Schiff bereits im 10. Jahrhundert unter dem Namen „Shabak“. Der erste Beleg für das Vorhandensein von Sciabeccos in westlichen Yachthäfen stammt aus dem 14. Jahrhundert. Doch weil es nur noch wenige solcher Belege gibt, glauben die Gelehrten, es sei ein selten gebautes Schiff gewesen. Aber dann tauchten wohl in Sizilien und Kalabrien immer mehr auf, dann in Spanien und in Portugal.

Im 16. Jahrhundert waren noch die Galeeren, geruderte Boote, das Standardkampfschiff in den europäischen Flotten. Kann man sich nur noch schwer vorstellen heute, oder? In den ersten Jahrzehnten des achtzehnten Jahrhunderts gaben die westlichen Marinen den Einsatz von Galeeren dann aufgrund ihrer Unwirksamkeit bei Zusammenstößen mit den schnelleren und inzwischen größeren Segelschiffen, mit quadratischen Segelsystemen und Artillerie, allmählich auf.

So begann der verbreitete Einsatz der schnellen und bewaffneten Sciabeccos. Das Arsenal von Neapel beschloss ein bemerkenswertes Bauprogramm: 1749 wurde der erste militärische Sciabecco gebaut, die „San Gennaro“, gefolgt von etwa zwanzig ähnlichen Einheiten, von denen einige auf der königlichen Werft in Palermo gebaut wurden. Für einen Großteil des neunzehnten Jahrhunderts noch war es einer der am meisten vertretenen Typen. Aber auch kaufmännisch wurden sie verwendet. Um gut zu segeln, erforderte dieses Schiff aber erfahrene Besatzung, um die riesigen Lateiner–Segel zu handhaben.

Und 1707 baute dann Denys Papin ein *Dampfschiff*, mit dem er auf der Fulda von Kassel nach Münden fuhr. Und nun fahren wir Atom–U–Boote. Irgendwie gings dann schnell, oder?

So, und nun ist mir eigentlich mal nach einem anderen Bücherregal. Bin neugierig, was es dort so gibt …

(Nach: Bellabarba S., Guerrieri E., Lo Sciabecco, in Vele italiane della costa occidentale, HOEPLI, Milano 2011, pp. 61–64 und Wikipedia: Dampfer.)

mk

Fortsetzung der Leserbriefe auf Seite 26

Mit Docker–Forth viele Forth–Systeme probieren

Ulrich Hoffmann

Möchte man verschiedene Forth–Systeme ausprobieren, etwa um zu prüfen, ob ein Programm, das man geschrieben hat, auch dort läuft, so ist es gewöhnlich nötig, dass man all diese Systeme vor der Benutzung auf seinem PC installieren muss. Mit Hilfe von Containervirtualisierung, wie sie etwa durch Docker zur Verfügung gestellt wird, ist es möglich, die Installation soweit vorzubereiten, dass der Start eines beliebigen Forth–Systems auf einem beliebigen Computer mit nur einem Kommando erfolgen kann.

Forth–Systeme auf Knopfdruck

Immer wieder kommt es vor, dass man mal einen Blick in ein Forth–System werfen möchte, das nicht das Leib–und–Magen–Forth ist. Das macht man etwa, um zu erfahren, wie sich ein anderes System in einer bestimmten Situation verhält oder vielleicht auch, um spezielle Features zu beobachten, die dieses System bietet. Wie war das noch mit den Wahrheitswerten in FIG–Forth? Oder wie gehen die interpretativen Kontrollstrukuren in cforth?

Hmm — FIG–Forth installieren?! Wie, wo, was? Ach egal.

Aber wir wollen die Flinte nicht gleich ins Korn werfen.

Mit Forth–Systemen, die in Containern schon installiert sind, lassen sich solche Experimente schnell machen. Wollen wir also FIG–Forth (hier die 32–Bit–Variante von Albert van der Horst [2]) starten, so können wir das mit

```
docker run -i -t -rm rundockerforth/figforth32
```

direkt machen und schon startet das System:

```
80386 IBM-PC $RCSfile: fig86.gnr,v $ $Revision: 2.148 $
10 10 = .
1 OK
10 20 = .
0
```

Ach ja — in FIG–Forth war `true` ja 1 und `false` ja 0. Gut zu wissen.

Und interaktive Kontrollstrukturen?

```
10 0 DO I . LOOP
DO?  MSG # 17 : COMPILATION ONLY, USE IN DEFINITION
```

OK — die gehen also nicht interpretativ, sondern nur in Definitionen.

Wie sieht das Ganze nun in Mitch Bradleys cforth [5] aus?

```
docker run -i -t -rm rundockerforth/cforth
```

und schon läuft es:

```
C Forth  Copyright (c) 2008 FirmWorks
ok 10 10 = .
ffffffffffffffff
ok 1 OK
ok 10 20 = .
0
ok 10 0 DO I . LOOP
0 1 2 3 4 5 6 7 8 9 a b c d e f
ok
```

Oh — das System startet in `HEX`, `true` hat alle (64) Bits gesetzt, und es kann wirklich interaktive Kontrollstrukuren.

Das Schöne an Containern ist, dass sie nicht schon auf meinem Computer vorhanden sein müssen. Stehen die zugehörigen Images in einem Repository zur Verfügung, werden sie unmittelbar heruntergeladen — hier vom Docker–Repository des Projekts Docker–Forth, das den Namensraum `rundockerforth` verwendet — und direkt gestartet. Das funktioniert auf jedem Computer, der für die Containervirtualisierung vorbereitet ist.

Standardprogramme

Um Programme mit möglichst vielen Leuten teilen und gut über diese Programme diskutieren zu können, haben wir in Forth Standards. Wenn von Forth–94 die Rede ist, dann ist klar, dass erwähnte Worte wie im 1994er ANS–Forth–Standard [6] zu verstehen sind. Es ist also beispielsweise klar, was das Wort `LSHIFT` ist und wie es sich verhalten soll.

Für Diskussionen um den entstehenden Forth–200x–Standard gibt es die Web–Seite `forth-standard.org`. Dort kann man auch die als Dokument verfügbare Version Forth–2012 einsehen.

Insbesondere wenn Standard–Programme oder Standard–Bibliotheken entstehen sollen, ist es wichtig, sie auf einer Vielzahl von Standard–Systemen prüfen zu können, um möglicherweise noch vorhandene Inkompatibilitäten aufzudecken.

Der Forth–Mund sagt *If you've seen one Forth, you've seen one Forth* und der Teufel steckt auch hier im Detail. Schnell ist ein Wort verwendet, dass einem zwar lieb geworden ist, das aber dann doch kein Standard–Wort ist und auf anderen Standard–Systemen gar nicht verfügbar ist.

Verschiedene Forth–Systeme auf Knopfdruck starten zu können, ist also nützlich, auch wenn man sich selbst auf ein festes Forth–System für sein aktuelles Projekt festgelegt hat.

Containervirtualisierung

Container sind virtuelle, gekapselte Programmablauf–Umgebungen, in denen ein (Gast–)Betriebssystem zusammen mit einer installierten Anwendung abläuft. Auf einem Wirts–Betriebssystem lassen sich Container wie Prozesse starten und können dann ihre Aufgaben verrichten: z. B. Daten etwa über Netzwerk oder per Eingaben einlesen, verarbeiten und geeignet wieder ausgeben. Container sind dabei voneinader isoliert, d. h. ohne Erlaubnis können sie nicht auf die Ressourcen (Dateisystem, Netzwerkverbindungen, Speicher, ...) des Wirts–Betriebssystems oder andere Container zugreifen. Das Wirts–Betriebssystem hingegen kann durchaus die Container–Ressourcen direkt erreichen.

Im Gegensatz zu virtuellen Maschinen teilen sich Container den Betriebssystemkern mit dem Wirts–Betriebssystem. Dadurch sind sie ressourcen–schonender und lassen sich einfacher und schneller verwalten.

Derzeit sehr beliebt ist die Containervirtualisierung mit *Docker* [3]. Hier wird die Konfiguration des Wirts–Betriebssystems und Installation der Anwendung mit Hilfe sogenannter *Dockerfiles* beschrieben, die die nötigen Konfigurations– und Installations–Schritte enthalten. Docker erzeugt aus einem Dockerfile ein *Image*, dass das konfigurierte Wirts–Betriebssystem und die installierte Anwendung enthält. Das Image dient als Blaupause für Container. Startet man einen Container, so wird das zugehörige Image verwendet, um den Anfangszustand des Containers festzulegen. Die Anwendung darin läuft ab und kann dabei den isolierten Container (nicht aber das Image) verändern. Wird der Container angehalten und gelöscht, werden alle zwischenzeitlichen Änderungen verworfen. Daten, die aufbewahrt werden sollen, müssen also geeignet aus dem Container herauskopiert werden. Das geschieht über Netzwerk oder über *Volumes*. Wie diese zu konfigurieren sind, lässt sich in den vielfach zu Docker erhältlichen Tutorials nachlesen, würde aber den Rahmen dieses Artikels sprengen.

Das Projekt Docker–Forth

Um möglichst viele Forth–Systeme auf diese Weise auf Knopfdruck starten zu können, ist das Projekt Docker–Forth [1] entstanden. Es stellt auf die beschriebene Weise eine ganze Reihe (derzeit, Frühjahr 2021, rund 20) verschiedene Forth–Systeme (und auch unterschiedliche Versionen des gleichen Systems) zur Verfügung.

Das `Dockerfile` etwa von cforth sieht dabei so aus, wie im Listing auf der nächsten Seite zu sehen ist. Das Gast–Betriebssystem ist die zur Image–Erstellungszeit aktuelle Debian–Buster–(10.x)–Distribution, die die Zeilen 6 bis 10 auf den aktuellen Stand bringen und notwendige Hilfspakete installieren. Das Docker–Kommando `RUN` lässt Befehle im Gast–Betriebssystem ablaufen.

Die Zeilen 3 und 4 erzeugen ein Verzeichnis `/src` als Volume, in das beim späteren Start des Containers ein Wirts–Verzeichnis eingebunden werden kann, damit man auf eigene Dateien zugreifen kann.

Die Installation von cforth erfolgt nun in den Zeilen 12 bis 14: Zuerst wird das zugehörige `git`–Repository geladen und dann das darin befindliche cforth mit `make` installiert.

Das Kommando `ENTRYPOINT` in Zeile 16 legt dann schließlich fest, wie die Anwendung, das cforth, gestartet wird. Das generierte cforth–64–Bit–System für Linux liegt unter `/cforth/build/host-serial-linux64/forth`.

Um nun aus diesem Dockerfile lokal ein Image zu erzeugen, wird das Kommando

```
docker build -t cforth
```

verwendet. Dem generierten Image wird der Name `cforth` zugeordnet, mit dem dann später Container gestartet werden können. Mit `docker push` können lokal erzeugte Images in ein Repository hochgeladen werden, damit auch andere dieses Image verwenden können.

All das muss einen als Benutzer von Docker–Forth aber nicht interessieren. Alle Docker–Forth–Images sind im Docker–Repository im Namensraum `rundockerforth` bereits verfügbar und können wie eingangs beschrieben gestartet werden.

Ausblick

Docker–Forth ist eine Basistechnik, die das Starten von vorbereiteten Forth–Systemen auf beliebigen Computern erlaubt. Diese Basistechnik kann nun in vielfältiger Weise in Projekte integriert werden. Wäre es nicht gut, gäbe es eine Web–Seite, bei der man einzelne Forth–Systeme probieren könnte, ganz ohne Docker selbst installiert zu haben?

Tatsächlich ist geplant, Docker–Forth einzusetzen, um forth-standard.org so zu erweitern, dass man dort auch Programme direkt mit unterschiedlichen Forth–Systemen testen kann.

Derzeitige Container basieren meist auf 64–Bit–Linux. Alles was unter Linux läuft, kann auch in einen Container. Das können mit Hilfe von Emulatoren auch Forth–Systeme sein, die eigentlich unter anderen Betriebssystemen laufen. Mecrisp–Stellaris [7] beispielsweise ist ein ARM–Forth, das in Docker–Forth durch `qemu` abläuft. Das eröffnet einen großen Bereich an historischen Forth–Systemen, die ebenfalls von Docker–Forth unterstützt werden könnten.

Wer gerne zum Docker–Forth–Projekt beitragen möchte, ist herzlich willkommen. Vielleicht gibt es ja noch nicht unterstützte Forth–Systeme, die unbedingt dabei sein sollten?

Pull–Requests auf GitHub, aber auch jede andere Form von Kommentaren und Verbesserungsvorschlägen tragen dazu bei, das Projekt nützlicher zu machen.

Fragen, Hinweise, Anregungen bitte gerne an `uho@xlerb.de`.

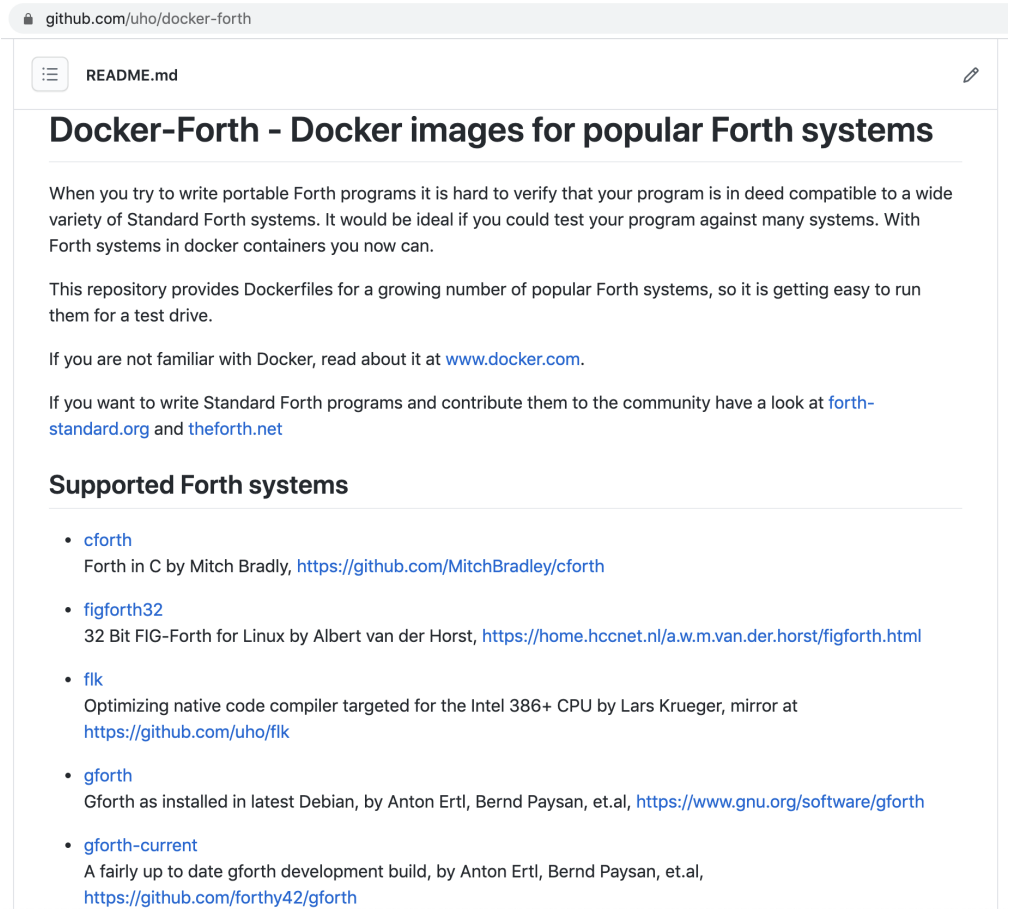
Referenzen

1. `https://github.com/uho/docker-forth/` GitHub–Repository des Docker–Forth–Projekts
2. `https://home.hccnet.nl/a.w.m.van.der.horst/figforth.html` 32–Bit–FIG–Forth für Linux
3. `https://docker.com` Docker
4. `https://theforth.net` Quellcode Repository TheForth.net
5. `https://github.com/MitchBradley/cforth` Mitch Bradleys cforth
6. `http://lars.nocrew.org/dpans/dpans.htm` Forth–94 online
7. `http://mecrisp.sourceforge.net/` Mecrisp Forth

Listings

`Dockerfile` für cforth

```
1   FROM debian:buster-slim
2
3   RUN mkdir /src
4   VOLUME /src
5
6   RUN apt-get update -q \
7     && DEBIAN_FRONTEND=noninteractive \
8     apt-get install -qy automake libtool libtool-bin gcc make git \
9     && apt-get clean \
10    && rm -rf /var/lib/apt
11
12  RUN git clone https://github.com/MitchBradley/cforth.git
13  WORKDIR /cforth/build/host-serial-linux64
14  RUN make
15
16  ENTRYPOINT ["/cforth/build/host-serial-linux64/forth"]
```



github.com/uho/docker-forth

README.md

Docker-Forth - Docker images for popular Forth systems

When you try to write portable Forth programs it is hard to verify that your program is in deed compatible to a wide variety of Standard Forth systems. It would be ideal if you could test your program against many systems. With Forth systems in docker containers you now can.

This repository provides Dockerfiles for a growing number of popular Forth systems, so it is getting easy to run them for a test drive.

If you are not familiar with Docker, read about it at www.docker.com.

If you want to write Standard Forth programs and contribute them to the community have a look at forth-standard.org and theforth.net

Supported Forth systems

- cforth
  Forth in C by Mitch Bradly, https://github.com/MitchBradley/cforth
- figforth32
  32 Bit FIG-Forth for Linux by Albert van der Horst, https://home.hccnet.nl/a.w.m.van.der.horst/figforth.html
- flk
  Optimizing native code compiler targeted for the Intel 386+ CPU by Lars Krueger, mirror at https://github.com/uho/flk
- gforth
  Gforth as installed in latest Debian, by Anton Ertl, Bernd Paysan, et.al, https://www.gnu.org/software/gforth
- gforth-current
  A fairly up to date gforth development build, by Anton Ertl, Bernd Paysan, et.al, https://github.com/forthy42/gforth

Multitasking für Stack–Prozessoren

Matthias Koch

Kooperatives Multitasking zählt zu den äußerst nützlichen und seit langem bewährten Werkzeugen in Forth. Das Grundprinzip ist immer das Gleiche: An vordefinierten Stellen werden die Stacks ausgetauscht, und ein anderer Task übernimmt.

Aber was ist, wenn es keine Stackpointer gibt, die umgebogen werden könnten, weil die Stacks, wie in Mecrisp–Ice, in Hardware implementiert sind? Es hat eine Weile gedauert, bis ich auf die Idee gekommen bin: Die Stackelemente können auch nacheinander rausgeschrieben und zurückgeholt werden — da in Forth nur selten wirklich tiefe Stacks verwendet werden, dauert es auch gar nicht mal so lange.

Der Speicherbereich für einen Task enthält dann — jeweils für den Daten– und Returnstack — zunächst die Zahl der gesicherten Elemente, und einen Speicherbereich für die maximal mögliche Zahl der Elemente in dem Hardwarestack, die bei Mecrisp–Ice auf 32 Elemente festgelegt ist.

In `(pause)` wird zunächst die Tiefe des Datenstacks notiert, anschließend werden die Elemente in einer Schleife rausgeschrieben. Gleiches wird für den Returnstack wiederholt, bevor der nächste Task gewählt und die Elemente aus dessen Task–Struktur zurückgeholt werden.

Die Vorbereitung eines neuen Tasks in `preparetask` wird auf diese Weise sehr einfach: Es muss nur die Startadresse als erstes und einziges Element in dem gesicherten Returnstack hinterlegt werden, während der Datenstack mit null Elementen startet.

Listing

```
1   \ ------------------------------------------------------------
2   \   Cooperative Multitasking
3   \ ------------------------------------------------------------
4
5   \ Configuration:
6
7   32 cells constant stackspace
8
9   \ Internal stucture of task memory:
10  \    0: Pointer to next task
11  \    2: Task currently active ?
12  \    4: Saved data stack depth
13  \    6: Parameter stack space
14  \  n+6: Saved return stack depth
15  \  n+8: Return stack space
16  \ 2n+8: Complete size of data structure.
17
18  create boot-task
19    here , \ Boot Task
20    true , \   is active.
21       0 , \ Saved depth: 0
22    stackspace allot
23       0 , \ Saved rdepth: 0
24    stackspace allot
25
26  boot-task variable up \ User Pointer
27  : next-task   ( -- task )    up @ ;
28  : task-state  ( -- state )   up @ 1 cells + ;
29  : task-data   ( -- data )    up @ 2 cells + ;
30  : task-return ( -- return )  up @ 3 cells stackspace + + ;
31
32  : (pause) ( stacks fly around )
33
34    \ ------------------------------------------------------------
35
36    depth task-data !  \ Number of elements
37    task-data 2 + >r    \ Begin with top of stack
38
39    begin
40      depth
41    while
42      r@ !
```

```
43      r> 2 + >r
44    repeat
45
46    rdrop
47
48    \ ------------------------------------------------------
49
50    rdepth task-return !  \ Number of elements
51    task-return 2 +        \ Begin with top of return stack
52
53    begin
54      rdepth
55    while
56      r> over !
57      2 +
58    repeat
59
60    drop
61
62    \ ------------------------------------------------------
63
64    begin
65      next-task @ up !     \ Switch to next running task
66    task-state @ until
67
68    \ ------------------------------------------------------
69
70       task-return @  \ Number of elements
71    2* task-return +   \ Begin with end of stack
72
73    begin
74      dup task-return <>
75    while
76      dup @ >r
77      2 -
78    repeat
79
80    drop
81
82
83    \ ------------------------------------------------------
84
85       task-data @  \ Number of elements
86    2* task-data +   \ Begin with end of return stack
87    >r
88
89    begin
90      r@ task-data <>
91    while
92      r@ @
93      r> 2 - >r
94    repeat
95
96    rdrop
97
98    \ ------------------------------------------------------
99  ;
100
101 : wake ( task -- ) 1 cells +  true swap ! ; \ Wake a random task (IRQ safe)
102 : idle ( task -- ) 1 cells + false swap ! ;  \ Idle a random task (IRQ safe)
103
104 \ ------------------------------------------------------
105 \  Round-robin list task handling - do not use in IRQ !
106 \ ------------------------------------------------------
107
108 : stop ( -- ) false task-state ! pause ; \ Stop current task
109 : multitask  ( -- ) ['] (pause) 2/ ['] pause ! ; \ Generate jump opcodes
110 : singletask ( -- ) [']  nop    2/ ['] pause ! ;  \ to be stored in pause
111
112 : task-in-list? ( task -- ? ) \ Checks if a task is currently inside of round-robin list (do not use in IRQ)
113   next-task
114   begin
115     ( Task-Address )
116     2dup = if 2drop true exit then
117     @ dup next-task = \ Stop when end of circular list is reached
118   until
```

```
119     2drop false
120   ;
121
122   : previous ( task -- addr-of-task-before )
123     \ Find the task that has the desired one in its next field
124     >r next-task begin dup @ r@ <> while @ repeat rdrop
125   ;
126
127   : insert ( task -- ) \ Insert a task into the round-robin list
128     dup task-in-list?  \ Is the desired task currently linked into ?
129     if drop else next-task @ over ! next-task ! then
130   ;
131
132   : remove ( task -- ) \ Remove a task from the round-robin list
133     dup task-in-list?  \ Is the desired task currently linked into ?
134     if dup @ ( task next )
135        swap previous ( next previous ) !
136     else drop then
137   ;
138
139   \ ----------------------------------------
140   \ Create a new task - do not use in IRQ !
141   \ ----------------------------------------
142
143   : task: ( "name" -- )  create stackspace 2* 8 cells + allot ;
144
145   : preparetask ( task continue -- )
146     swap >r ( continue R: task )
147
148       \ true  r@ 1 cells + ! \ Currently running
149            0  r@ 2 cells + ! \ Empty data stack
150            1  r@ 3 cells stackspace + + ! \ One element in return stack
151               r@ 4 cells stackspace + + ! \ Store the desired entry address at top of the tasks return stack
152
153     r> insert
154   ;
155
156   : activate ( task --   R: continue -- )
157     true over 1 cells + ! \ Currently running
158     r> preparetask
159   ;
160
161   : background ( task --   R: continue -- )
162     false over 1 cells + ! \ Currently idling
163     r> preparetask
164   ;
165
166
167
```

Fortsetzung der Leserbriefe von Seite 20

BeLUG Pico Sizecoding Competition

Für all jene, die sich gerne mit trickreichen, winzigkleinen Programmen beschäftigen, könnte der „Picocomp“ interessant sein: `https://picocomp.belug.de/` Es geht darum, etwas ganz besonders Schönes oder Nützliches mit dem neuen Raspberry Pico anzustellen — und das in 256, 1024 oder 4096 Bytes. Natürlich läuft auch Mecrisp–Stellaris bereits auf dem Chip, dem RP2040, aber Forth dürfte bestimmt die 4096 Bytes sprengen. Oder? Mag jemand die Herausforderung annehmen, kreativ so viel Forth wie möglich in 4096 Bytes auf einem ARM Cortex M0 zu quetschen? Ansonsten sind aber auch „ganz normale“ Assemblerprogramme herzlich willkommen. In die 256 Bytes passen übrigens außer dem „angenehmen Blinken“ noch ein Mandelbrot– und ein Tricorn–Fraktal in ASCII–Art !

```
          ---
       .-*)) ‘*-.
      /*  ((*   *’.
     |   *))  *   *\
     | *  ((*   *  /
      \  *))  *  .’
 jgs  ’-.((*_.-’
```

Frohe Ostern gehabt zu haben.[1]

Matthias Koch

[1] Das Heft noch vor Ostern herauszubringen, hat nicht ganz geklappt. (der Sätza)

Forth–Gruppen regional

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

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

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

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

Dienste der Forth–Gesellschaft

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

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

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

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

Spezielle Fachgebiete

Forth-Hardware in VHDL microcore (uCore)	**Klaus Schleisiek** Tel.: **(0 58 46) – 98 04 00 8**$_p$ kschleisiek@freenet.de
KI, Object Oriented Forth, Sicherheitskritische Systeme	**Ulrich Hoffmann** Tel.: **(0 41 03) – 80 48 41** uho@forth-ev.de
Forth–Vertrieb volksFORTH ultraFORTH RTX / FG / Super8 KK–FORTH	Ingenieurbüro **Klaus Kohl–Schöpe** Tel.: **(0 82 66)–36 09 862**$_p$

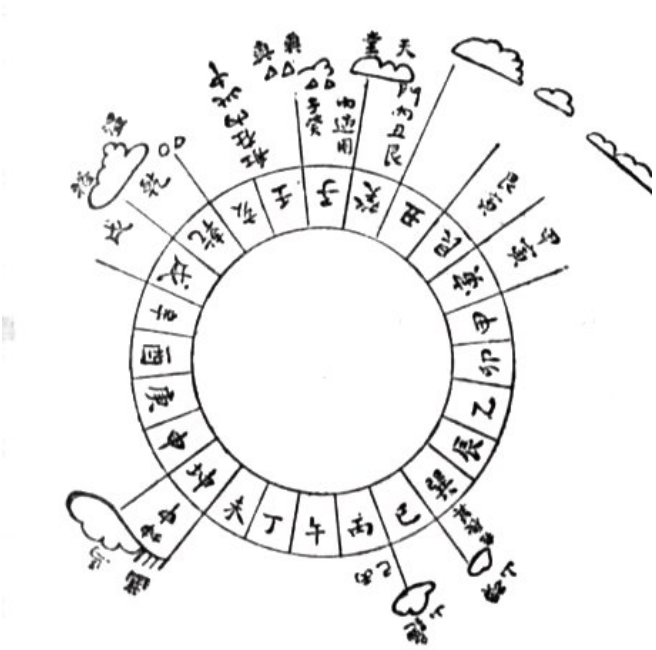
Termine

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

Montags ab 20:30 Uhr
Forth lernen
Videotreffen (nicht nur) für Forthanfänger
Info und Teilnahmelink: E–Mail an wost@ewost.de

Jeder 2. Samstag im Monat
ZOOM–Treffen der Forth2020 Facebook–Gruppe
Infos zur Teilnahme: www.forth2020.org

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.

TAGUNG UND MITGLIEDERVERSAMMLUNG 2021

Forth–Gesellschaft Direktorium

Liebe Mitglieder,

die Situation rund um die COVID–19–Pandemie macht es dem Direktorium schwer, eine Tagung mit Mitgliederversammlung zu planen. Das Direktorium der Forth–Gesellschaft möchte im Jahr 2021 eine Tagung mit Mitgliederversammlung durchführen. Jedoch lebt die Tagung und die MV von Euch, der Teilnahme der Mitglieder. Ohne eine rege Teilnahme ist die Tagung und die MV wenig interessant.

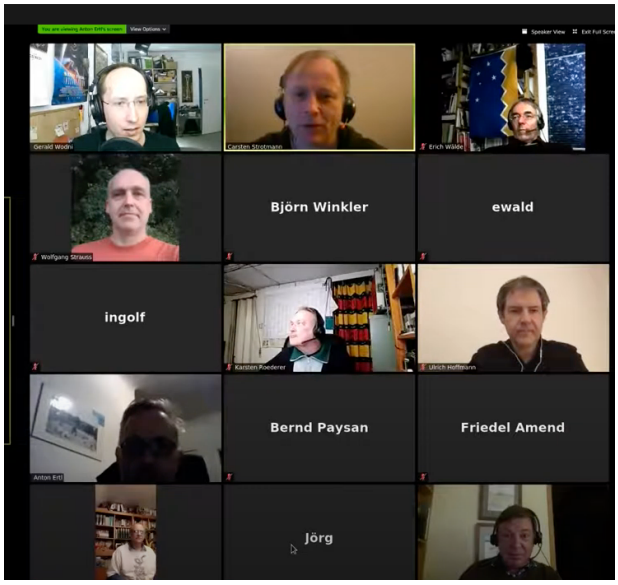
Über eine Abstimmung möchten wir, das Direktorium, erfahren, welche Art einer Tagung mit MV unsere Mitglieder wünschen.

Als Zeitpunkt für die Tagung/MV haben wir den Spätsommer (August bis Anfang September) geplant.

Es gibt vier Optionen zur Auswahl (wenn wir eine Option übersehen haben, schreibt uns bitte, wir können die Abstimmung anpassen):

- Klassische „In Person“–Tagung mit MV (wenn zulässig): Für diese Option werden wir das schon für die Tagung 2020 geplante Tagungs–Haus „CocoNAT“ in Bad Belzig anmieten. Das CocoNAT hat großzügige Außenflächen, so dass wir bei gutem Wetter die Tagung teils an der frischen Luft durchführen können. Vielleicht sind im Spätsommer 2021 auch viele Mitglieder schon geimpft, so dass sich die Infektions–Lage entspannt.

- Kombinierte Online– + „In Person“–Tagung mit MV (wenn zulässig): Je nach Situation wird nicht jedes Mitglied das Risiko eines Treffens in Person tragen wollen. Andere Mitglieder sehnen sich jedoch schon länger nach direkten Gesprächen mit Freunden. Um beiden Gruppen gerecht zu werden, ist eine kombinierte Online– und „In Person“–Tagung möglich. Dabei werden die Online–Teilnehmer per Video–Chat in den Tagungsraum eingebunden und können per Audio, Video und Chat an der Tagung teilnehmen.
- Tagung mit MV nur Online via Videokonferenz: Wie im Jahr 2020 werden wir eine reine Online–Tagung durchführen, nur mit der Erfahrung von über einem Jahr Online–Konferenzen und Home–Office–Arbeit (also besser). Es wird auch eine Online–Mitgliederversammlung geben, inkl. der Möglichkeit einer Online–Abstimmung.



- Keine Tagung/MV in 2021, wir warten auf 2022: „In Person“–Treffen sind auch im Spätsommer 2021 zu riskant, und eine Online–Tagung ist kein Ersatz. Besser ist es, das Ende der Pandemie abzuwarten und dann eine klassische Tagung in 2022 durchzuführen.

Die Abstimmung läuft bis zum 15.06.2021 unter:

`https://nuudel.digitalcourage.de/cKsm26KXhRvdrIu2`

Auch wenn die Abstimmung bis Mitte Juni läuft, möchten wir die Mitglieder bitten, so schnell wie möglich abzustimmen, so dass das Direktorium ggf. schon früh einen Trend bekommt und mit der Planung der Tagung starten kann.

Die Abstimmung ist öffentlich, auf der Webseite können öffentlich Anmerkungen eingetragen werden. Wer nicht öffentlich abstimmen möchte oder direkt an das Direktorum einen Kommentar senden möchte, kann dieses per E–Mail–Adresse `direktorium@forth-ev.de` oder ganz klassisch per Briefpost tun.

Wir freuen uns auf rege Rückmeldungen

Euer Direktorium

Ulrich Hoffmann, Bernd Paysan, Carsten Strotmann