Vierte Dimension

4/2009

Das Forth–Magazin

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

In dieser Ausgabe:

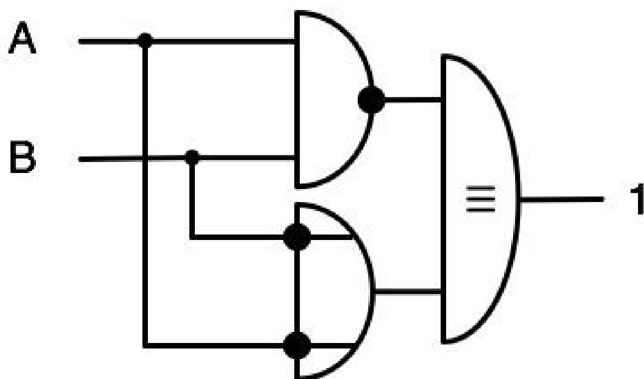
fJACK – Echtzeit-Audio in Forth

Minimalbasis

Forth im FPGA — Teil 2, b16

Gleitkommazahlen

FFT–Bit–Umkehrung



Dienstleistungen und Produkte fördernder Mitglieder des Vereins

tematik GmbH
Technische Informatik

Feldstrasse 143
D–22880 Wedel
Fon 04103 – 808989 – 0
Fax 04103 – 808989 – 9
mail@tematik.de
www.tematik.de

Gegründet 1985 als Partnerinstitut der FH–Wedel beschäftigten wir uns in den letzten Jahren vorwiegend mit Industrieelektronik und Präzisionsmeßtechnik und bauen z. Z. eine eigene Produktpalette auf.

Know–How Schwerpunkte liegen in den Bereichen Industriewaagen SWA & SWW, Differential–Dosierwaagen, DMS–Messverstärker, 68000 und 68HC11 Prozessoren, Sigma–Delta A/D. Wir programmieren in Pascal, C und Forth auf SwiftX86k und seit kurzem mit Holon11 und MPE IRTC für Amtel AVR.

LEGO RCX–Verleih

Seit unserem Gewinn (VD 1/2001 S.30) verfügt unsere Schule über so ausreichend viele RCX–Komponenten, dass ich meine privat eingebrachten Dinge nun Anderen, vorzugsweise Mitgliedern der Forth–Gesellschaft e. V., zur Verfügung stellen kann.

Angeboten wird: Ein komplettes LEGO–RCX–Set, so wie es für ca. 230,– € im Handel zu erwerben ist.

Inhalt:

1 RCX, 1 Sendeturm, 2 Motoren, 4 Sensoren und ca. 1.000 LEGO Steine.

Anfragen bitte an
Martin.Bitter@t-online.de

Letztlich enthält das Ganze auch nicht mehr als einen Mikrocontroller der Familie H8/300 von Hitachi, ein paar Treiber und etwas Peripherie. Zudem: dieses Teil ist „narrensicher“!

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

Hier könnte Ihre Anzeige stehen!

Wenn Sie ein Förderer der Forth–Gesellschaft e.V. sind oder werden möchten, sprechen Sie mit dem Forth–Büro über die Konditionen einer festen Anzeige.

`Secretary@forth-ev.de`

KIMA Echtzeitsysteme GmbH

Tel.: 02461/690–380
Fax: 02461/690–387 oder –100
Karl–Heinz–Beckurts–Str. 13
52428 Jülich

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

FORTecH Software GmbH
Entwicklungsbüro Dr.–Ing. Egmont Woitzel

Bergstraße 10 D–18057 Rostock
Tel.: +49 381 496800-0 Fax: +49 381 496800-29

PC–basierte Forth–Entwicklungswerkzeuge, comFORTH für Windows und eingebettete und verteilte Systeme. Softwareentwicklung für Windows und Mikrocontroller mit Forth, C/C++, Delphi und Basic. Entwicklung von Gerätetreibern und Kommunikationssoftware für Windows 3.1, Windows95 und WindowsNT. Beratung zu Software–/Systementwurf. Mehr als 15 Jahre Erfahrung.

Hier könnte Ihre Anzeige stehen!

Wenn Sie ein Förderer der Forth–Gesellschaft e.V. sind oder werden möchten, sprechen Sie mit dem Forth–Büro über die Konditionen einer festen Anzeige.

`Secretary@forth-ev.de`

Ingenieurbüro
Klaus Kohl–Schöpe

Tel.: 07044/908789
Buchenweg 11
D–71299 Wimsheim

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

Leserbriefe und Meldungen 5

Gehaltvolles 6
zusammengestellt und übertragen von *Fred Behringer*

fJACK – Echtzeit-Audio in Forth 7
Hanno Schwalm

Minimalbasis 10
Fred Behringer

Forth im FPGA — Teil 2, b16 17
Ulrich Hoffmann

Lebenszeichen 21
Berichte aus der FIG Silicon Valley: *Henry Vinerts*

Gleitkommazahlen 22
Michael Kalus

FFT–Bit–Umkehrung 28
Fred Behringer

Impressum

Name der Zeitschrift
Vierte Dimension

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

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

Anzeigenverwaltung
Büro der Herausgeberin

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

Erscheinungsweise
1 Ausgabe / Quartal

Einzelpreis
4,00€ + Porto u. Verpackung

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

Liebe Leser,

willkommen zur vierten Ausgabe unseres Forth-Magazins. Ich hoffe, Ihr seid alle gut in das neue Jahr gekommen. Ihr haltet dieses Heft erst jetzt in den Januartagen in den Händen. Zum Ende des letzten Jahres war es dann doch zu trubelig, als dass wir die Vierte Dimension in der gewohnten Sorgfalt hätten fertigstellen können. Das ist wahrscheinlich ein Zeichen, dass die Redaktion Verstärkung braucht. Wer also Lust hat, an der Erstellung künftiger Ausgaben mitzuwirken, und auch vor LaTeX oder Subversion nicht zurückschreckt, der kann sich gerne an mich unter uho@forth-ev.de wenden. Aber nicht nur helfende Hände werden gesucht. Wie eigentlich ständig, brauchen wir neue Artikel. Wenn Du also beim Surfen etwas Interessantes im Netz findest oder gerade selbst an einem spannenden Forth-Projekt arbeitest, dann schreib doch was für die Vierte Dimension.

So - was steht also im neuen Jahr an? Nun - zunächst mal die Forth-Tagung in Warnemünde/Rostock (26.-28. März 2010). Die Einladung liegt diesem Heft bei. Im Juni findet dann in Berlin wieder der Linuxtag (9.-12. Juni 2010) statt und ich kann jedem nur empfehlen, dort mal vorbeizuschauen. Günstige und gute Unterkünfte in Berlin kann ich gerne nennen. Und — vielleicht auch Lust, auf dem Stand mitzuwirken?

Was gibt es in dieser Ausgabe Spannendes zu finden?
Hanno Schwalm berichtet uns über fJACK, die Anbindung der libjack Audio-Biliothek an iForth und VFX. Michael Kalus erklärt uns, was jeder Forth–Programmierer über Gleitkommazahlen wissen muss, wenn er sie denn wirklich einsetzen will oder muss. Fred Behringer greift die Diskussion von Willi Strecker über eine minimale Instruktions–Basis für ein Forth–System auf und setzt sich erschöpfend mit den logischen Operatoren auseinander. In einem weiteren Artikel erläutert er, wie man die für die schnelle Fourier-Transformation benötigte Bitumkehr effizient realisieren kann, auch wenn das verwendete Forth-System eine geringere Zellbreite besitzt als das umzukehrende Wort. Ich selbst erzähle, wie man Bernd Paysans b16–Prozessor auf dem Altera–FPGA–Board ans Laufen bringt und wie man mit Hilfe der dort vorhandenen 7–Segment–Anzeige Debugging macht.

Oft wird viel beklagt, dass die jungen Leute (ich sag jetzt mal $20 <) überhaupt kein Interesse mehr an Technik oder am Programmieren haben. Das stimmt nicht! Ein Blick auf Treffen Commputerbegeisterter wie etwa den Linuxtag oder der Chaos Communication Congress zeigt, dass es viele Junge gibt, die mit Technik gerne, viel und kreativ umgehen. Nur — Forth spielt hier keine Rolle; es ist weitgehend unbekannt; *hip* ist anders. Auch Felder, in denen Forth vorgibt, stark zu sein, (eingebettete Systeme, interaktive Programmierung) werden anders beackert: Assembler, C, C++, Python, Lisp, Erlang, Haskell. Wir sollten in diesen Kreisen Forth bekannt machen und unseren Weg aufzeigen.
Wie können wir junge Leute von Forth begeistern?

Ulrich Hoffmann

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
`http://www.forth-ev.de/filemgmt/index.php`

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

Ulrich Hoffmann Kontakt: Direktorium@Forth-ev.de
Bernd Paysan
Ewald Rieger

Leserbrief zum Josephspfennig

Johannes Brooks hat in seinem Leserbrief aus dem VD–Heft 3/2009 zum besagten Thema eine Rechnung in echten Brüchen unter LISP durchgeführt und das unten stehende Ergebnis (c) erzielt. Meine Berechnung auf dem *Taschenrechner* unter Windows–XP, mit Logarithmieren, Multiplizieren–mit–2003 und dann Delogarithmieren (keine exakte Arithmetik, aber) stimmt bis auf erstaunliche 31 Zehnerstellen mit dem Ergebnis von Johannes überein. Ich habe im VD–Heft 2/2009 über meine Bemühungen berichtet. Von Michael stammt das Ergebnis (b).

(a) Fred: XP–Taschenrechner
27.679.996.896.157.634.534.421.221.977.464.000.000.000.00 DM

(b) Michael: Forth–Programm
27.679.996.896.051.261.677.068.884.476.135.650.875.110,12 DM

(c) Johannes: LISP–Programm
27.679.996.896.157.634.534.421.221.977.463.776.885.435,84 DM

Was sagt uns das? Mit großer Wahrscheinlichkeit liefern die Beträge (a) und (c) das Ergebnis auf 31 Dezimalstellen genau, auch wenn das Logarithmieren und das Delogarithmieren alles andere als exakte Rechenweisen darstellen. An sich sollte zu jeder in Physik und Technik gelieferten Zahlenangabe eine Fehlereingrenzung mitgeliefert werden! Es ist gar zu verführerisch, hinter einer Zahl mit vielen Stellen hinter dem Komma mehr zu vermuten, als das, was tatsächlich dahintersteckt.

Über das Rechnen mit Bruchzahlen in Forth gibt es ein Forth–Programm, `VULGAR.MF` von Gordon R. Charlton (*Vulgar Numbers* (Gemeine Brüche)), das 1996 als Forth Scientific Library Algorithm #46 herauskam. Man beachte, dass schon die extrem einfache Bruchzahl 1/3 auf dem Computer im üblichen Positionssystem nicht endlich darstellbar ist, weder im Binär– noch im Hexadezimal– noch im Dezimalsystem. Man sollte wirklich dem Aufruf von Johannes Brooks folgen und für die Bereitstellung eines akzeptierbaren Bruchzahlen–Forthpaketes sorgen. Schwierig wird es, wenn Zähler und Nenner beim Verarbeiten übermäßig stark anwachsen, ohne dass der Bruch gekürzt werden kann (*relativ prim*).

Vielleicht wäre es in diesem Zusammenhang angebracht, (endlich) einmal den Bericht von Christoph Pöppe aus dem VD–Heft 3/2001, S.25–30, aufzugreifen und für Forth umzusetzen: *Rechnen mit garantierter Genauigkeit.* Es geht um Arbeiten von Professor Ulrich Kulisch und seiner Schule, die sich seit 1966 intensiv mit der Fortentwicklung der Intervallarithmetik beschäftigen. Interessant der im VD–Artikel zu findende Ausspruch *Besser gut iteriert als falsch exakt.*

Fred Behringer

Errata

Im Listing vom Artikel *Minimaler Basis–Befehlssatz für ein Forth–System* von Willi Stricker in der *Vierten Dimension* 3/2009 Seite 17, heißt es in den Zeilen 92 und 105:

```
92 : * (data0data1->prod) M* D->S ;

105 : /MOD ( data1 data2 -> mod quot ) >R S->D R> M/MOD ;
```

Hier werden die Worte `D->S` und `S->D` verwendet. Richtig muss es `D>S` und `S>D` heißen. Die Worte werden gerade vorher in Zeile 81 und 78 definiert.

von den Elektor-Machern entgegengenommen von Fred Behringer

Ab dem 18. November gibt es an den Kiosken ein neues Elektor-Mikrocontroller-Sonderheft mit bisher noch nicht veröffentlichten Schaltungen und weiteren interessanten Themen. Das Sonderheft wendet sich an alle, die sich privat oder beruflich mit Mikrocontrollern hard- oder/und softwaremäßig beschäftigen. Aus dem Inhalt:

- Mehr RFID - Ein RFID-System aus Standardkomponenten; nicht nur zum Identifizieren
- Sparpumpe - Umbau einer ECM- zur Systempumpe
- Operation am offenen Herzen - Programmierung des Mikrocontrollers im Lego-Mindstorms
- HaiTech - BF537LANcore: Mini-DSP-Modul mit Blackfin-DSP und Xilinx-FPGA
- ON/OFF - USB-Lineswitch
- Media-Cube - Aufbau eines kleinen Media-PCs zum Surfen, für das Heimkino und als Web-Radio

Gehaltvolles

zusammengestellt und übertragen von *Fred Behringer*

VIJGEBLAADJE der HCC Forth–gebruikersgroep, Niederlande Nr. 76, Oktober 2009

Ervaringen met een Arduino bordje — Ron Minke

Ron hatte im letzten Vijgeblaadje mit seinem Erfahrungsbericht begonnen. Er setzt ihn jetzt fort. Dem Benutzer wird eine gut ausgerüstete Entwicklungsumgebung zur Verfügung gestellt. Bei den Benutzergruppen gibt es viele schon entwickelte Programme - aber noch kein Forth. Was das Aufladen eines Forth-Systems betrifft, so könnte man das mit dem AVRDude (AVR Downloader-UploaDEr) erledigen. Auf der Arduino-Platine sitzt ein ATmega328P. Dieser Prozessor wird aber vom AVRDude noch nicht anerkannt und muss durch einen ATmega168 ersetzt werden. Ron berichtet über Kommunikationsprobleme (57600 Baud statt 19200), die mit dem Prozessor-Ersatz zusammenhängen, und wie er diesem Problem auf die Spur gekommen ist. Dann berichtet er über die Konfigurations-Datei und schließlich kommt er auf Forth zu sprechen: Ron hat sein 'Forth von der Pike auf' an den ATmega328P angepasst.

Forth is overal — Frans van der Markt

Zwei Fotos mit einem Straßenschild *FORTH ST.*, die Frans von seinem Urlaub in Newcastle (GB) mitgebracht hat.

VIJGEBLAADJE der HCC Forth–gebruikersgroep, Niederlande Nr. 77, Dezember 2009

Ervaringen met een Arduino bordje III — Ernst Kouwe

Vor dem Treffen der Forth–gg am 12.Dezember 2009 waren die zu erwartenden Teilnehmer aufgefordert worden, ihre Laptops mitzubringen und sich am Experimentieren mit den 12 vorhandenen Arduino–Platinen zu beteiligen. Ernst Kouwe und Ron Minke sind offensichtlich die aktivsten unter den Aktiven in Sachen Arduino–Ausprobieren. In den Vijgeblaadjes 75 und 76 hatte Ron über seine Erfahrungen berichtet, diesmal ist Ernst dran.

Ernst dankt Ron für seine beiden Artikel über Arduino und AVRDude, sagt aber auch Willem Ouwerkerk Dank für dessen Tips über AVRF.CFG und für eine weitere verbesserte Version von AVR–ByteForth.

Der Artikel enthält folgende Abschnitte:

Einleitung (Arduino für Anfänger gedacht, vergleichbar mit dem Egelwerkboek der Forth–gg, aber in mancher Hinsicht besser),

Wie programmiert man einen Arduino? (via RS232 über USB, unter Zuhilfenahme von AVRDude über Rx, Tx und die DTR–Signalleitung),

DOS von ByteForth aus aufrufen (hier werden die nötigen Befehle dafür angegeben),

DOS–Puffer (das neue ByteForth stellt für das Durchreichen der DOS–Kommandos 256 Bytes zur Verfügung, beim alten ByteForth waren es nur 64. Um auch mit dem alten ByteForth arbeiten zu können, muss die AVR–Zeile zur Verkürzung in eine Batch–Datei gesetzt werden),

Die einzelnen Schritte (ein paar Angaben für die genannte Batch–Datei),

Als Abschluss–Beispiel bringt der Artikel ein Forth–Progamm von 11 Zeilen zum Blinkenlassen einer LED an Pin 13.

SPI spy — Ron Minke

Bei der Programmentwicklung für den AVR–Prozessor entstand der Wunsch, den Programm–Ablauf verfolgen zu können. Mit einem zweiten AVR–Prozessor geht das (im ersten Prozessor Forth, im zweiten das Monitor–Programm). In jedem AVR–Prozessor gibt es ein SPI (serial programming interface), über das der Prozessor (im Reset–Zustand) mit einem Spezial–Programmierer (z.B. STK500 von Atmel) neu programmiert werden kann. Der Mechanismus ist auch für die Beobachtung eines laufenden Systems geeignet. Dafür zuständig ist im Slave–Prozessor ein per Interrupt angeschlossenes Programm, das über einen MISO–MOSI–Ringpuffer arbeitet. Ein Schaltbild im Artikel zeigt den (leicht nachvollziehbaren) Aufbau. Die Schaltung besteht aus einem AT90S8115, einem ATmega162, zwei MAX232, einem SPI–Interface, einem Reset–Knopf, dem Reset–Hochzieh–Widerstand und dem obligatorischen Leitungsmaterial.

Das Vijgeblaadje und die HCC–Forth–gg

Das Vijgeblaadje erscheint um den Ersten eines jeden geraden Monats herum. Neues von der Forth–gebruikersgroep erfährt man per `http://www.forth.hcc.nl/nieuws`.

fJACK – Echtzeit-Audio in Forth

Hanno Schwalm

Die Zeiten, in denen ein dezent–nervendes Piepen aus dem Inneren des PCs als *Sound–Erfolg* gelten konnte, sind für immer vorbei. Heute haben wir es mit einer unüberschaubaren Vielfalt von Hardware und mindestens 3 Betriebssystemen zu tun und ein hochqualitatives Audioerlebnis wird von allen Benutzern erwartet.

Aber was hat das denn mit Forth zu tun? Ich habe doch keine HiFi–Zeitung sondern ein Magazin für Forth–Anwender in der Hand — ein kleiner Blick auf den Umschlag wird dies bestätigen. Im Frühjahr 2009 habe ich auf der Tagung der Forth–Gesellschaft einen Vortrag über ein Echtzeit–Audio–System in Forth – iJACK – gehalten. Inzwischen ist iJACK gründlich renoviert und auch erwachsen geworden, es nennt sich jetzt fJACK und läuft unter allen aktuellen iFORTH–Systemen sowie VFX Forth von MPE.

Ulli Hoffmann fragte mich damals, ob es denn spezielle Gründe für die Verwendung von Forth gab — jetzt würde ich sagen: Ich kenne kein anderes System zur Audio–Echtzeit–Verarbeitung, mit dem sich interaktiv besser spielen lässt. Ich höre mit Forth Musik von der Konserve, vom Radio oder auch aus dem Live–Mischpult, schreibe an den Audio–processing Plugins und kann sie unmittelbar testen und wieder aus dem System entfernen. Also FORTH pur!

Was solls denn können?

Schon auf meinem guten alten Acorn–Rechner habe ich mich mit Audio–Programmierung beschäftigt, damals zu großen Teilen in Assembler und als Interruptroutine, die Entwicklungsumgebung war natürlich Forthmacs. Heute ist das komplizierter und gleichzeitig einfacher: Interrupts sind in allen Betriebssystemen für Anwendungen quasi tabu und heutige Compiler sollten Assemblercode eigentlich überflüssig machen – und das ist in der Intel–Welt auch gut so.

Vor dem iForth–3.0–Release hatten Marcel (Hendrix) und ich länger über unsere Unzufriedenheit mit dem Audiosystem diskutiert. Es gab für 3 Betriebssysteme (Linux, Windows und Mac OS X) jeweils 32– und 64–Bit–spezifischen Code, zum Teil in Forth, zum Teil in C geschrieben. Die Wartung war gelinde gesagt schwierig. Außerdem waren wir mit der Leistung des vorhandenen Systems unzufrieden: eine viel zu hohe CPU–Last, kein wirklicher Duplexbetrieb, schlechtes Zeitverhalten, keine Schnittstelle zu anderen Programmen . . . Also eine unhaltbare Situation. Ein völlig neu entwickeltes Audio–System war erforderlich.

Als Vorgaben hatten wir festgelegt:

1. Alle iFORTH–Versionen sollten unterstützt werden können (Also 3*2)
2. Niedrige CPU–Last und damit Performance wie andere erstklassige Systeme, Echtzeitfähigkeit und niedrige Input–Output–Latenz → Also kein Polling in threads sondern ein interruptgesteuertes System war gefordert
3. *Pflege* der Hardwarebasis durch andere fleißige Menschen :-)
4. Hohe Stabilität und damit auch für Bühne, Musikstudio und Langzeitbetrieb nutzbar
5. Gut definierte und gut dokumentierte Schnittstelle → weniger Arbeit für mich
6. .wav–Dateien/Daten in allen Sampleraten müssen unabhängig von der Hardware abgespielt und aufgezeichnet werden können, und das ohne Knistern oder Aussetzer.
7. Weiterleitung von Audiodaten zwischen verschiedenen Programmen muss vollständig unterstützt sein

Ganz schön happig, oder? Da auch ich nicht gerne Räder neu erfinde, habe ich nach vorhandenen Projekten gesucht und durch den Vorgaben–Filter geschickt. Unten fiel eigentlich nur ein Projekt raus: das JACK–Audiosystem mit dem JACK–Sound–Demon *jackd* und der Bibliothek *libjack*. Es ist für alle wichtigen Betriebssysteme verfügbar, es wird aktiv dran gearbeitet und es gibt eine relativ große und aktive Entwicklergruppe.

Ich habe dann die *libjack*–Dokumentation durchgelesen und mich in den einschlägigen Internet–Foren umgehört, was Benutzer anderer Programme dazu sagen. Zumindest für Linux wurde klar: Es gibt für anspruchsvolle Umgebungen keine wirkliche Alternative, damit war die Entscheidung gefallen.

Eine Forth–Audio–Umgebung sieht in etwa so aus:

- Ein JACK–System (`http://jackaudio.org`) mit dem Sound–Demon *jackd* und der Bibliothek *libjack* ist installiert. Bei korrekter Installation wird der Demon von anderen Programmen automatisch gestartet, ansonsten gibt es auch graphische Frontends wie *qjackctl*.
- Die komplette Audiohardware wird unter allen Betriebssystemen vom *jackd* kontrolliert und alle Audioanwendungen benutzen die *libjack*. Audiodaten liegen intern und in den Schnittstellen immer im gleichen Datenformat vor (sfloats).
- Forth hat eine Schnittstelle zur *libjack*, das Forth–System fJACK ist aus der Sicht des *jackd* ein ganz normaler Client inclusive Callbacks und Datenstrukturen. Forth kann Audiodaten des *jackd* aufnehmen, verarbeiten oder liefern.

- Alle Funktionen des fJACK werden ganz normal in Forth programmiert und können von allen Forth–Programmen genutzt werden.

Im Folgenden werde ich zeigen, wie fJACK implementiert wurde – ohne dabei mit endlosen Listings Papier und Geduld übermäßig zu strapazieren. In dem nächsten Artikel werde ich den Signalfluss der Audiodaten und die fJACK–Forth–Bibliothek vorstellen und zeigen, wie eine ganz einfache Audioanwendung aussehen könnte. Ein abschließender Artikel stellt dann fJACK–Plugins im Detail vor, wir werden dann gemeinsam ein Echtzeit–Echo/Hall–System entwickeln und benutzen.

IForth 4.0 gibt's jetzt auch in einer Evaluation–Version und diese lege ich hiermit allen Lesern ausdrücklich ans Herz. Zwischen Installation und erstem Audioerlebnis werden kaum 10 Minuten verstreichen! Für diejenigen, die Anleitungen nicht lesen: iForth starten und auf der Kommandozeile

```
in fjack
```

eingeben. Ein kleiner Hinweis an dieser Stelle: Wer ein technisches Programmer's Reference Manual sucht, wird unter *includes/fjack/fjack.html* fündig werden.

Das Forth–System

Es kommt nur ein stabiles ANS–Forth–System mit hochoptimierendem Compiler für Integer und Floating–Point–Daten in Frage. Relativ komplexer Code mit massenweise Berechnungen in Floating Point wird z. B. 50/sec in einem Callback ausgeführt – daraus ergibt sich der Bedarf nach einem möglichst guten Compiler. iForth und VFX sind für diesen Anwendungsfall die mit Abstand schnellsten Systeme und werden beide unterstützt. Für andere Forth–Systeme wäre mit schweren Performance–Einbußen zu rechnen.

In den letzten fJACK–Versionen wird außerdem unter iForth auf entsprechenden CPUs auf optimierte 64–Bit–SSE2–Filter–Routinen umgeschaltet, die Leistung verdoppelt sich erwartungsgemäß nahezu.

Zusätzlich sind eine Reihe von Forth–Erweiterungen essentiell:

1. Einbinden von Funktionen aus beliebigen dynamischen Bibliotheken
2. Spezielle Datentypen mit `TO +TO` müssen machbar sein
3. Eine Schnittstelle zur Ausführung von Forth als Callback aus externen Programmen
4. Threading–Unterstützung

Funktionen in dynamischen Bibliotheken

Die *libjack* Funktionen werden über die aus VFX bekannte Syntax eingebunden, also

```
Library: libjack.so.0
Extern: int "C" jack_client_close ( jack_client_t * client);
Extern: float "C" jack_cpu_load   ( jack_client_t * client);
```

Beide Funktionen erwarten die ID des Clients auf dem Stack, `jack_client_close` beendet den Client und `jack_cpu_load` liefert die relative Last. Auf diese Weise werden alle benötigten *libjack* Funktionen verfügbar gemacht[1]. Der Aufwand und die nötige Cleverness für einen solchen Funktionsaufruf ist erheblich:

- die Forth–internen Register der CPU werden gesichert
- die Parameter werden typ–abhängig von den Forth–Stacks in die Parameter des C–Interface *übersetzt*. Dazu werden primär diverse CPU–Register und sekundär der Prozessor–Stack verwendet, die Zuordnung und Sortierung der Register ist jedoch abhängig vom OS und 32/64–Bit–Version. Da wird viel Historie herumgeschleppt — so ein Interface korrekt zu implementieren, ist wirklich aufwendig.
- Der Prozessor–Stack muss bei manchen Betriebssystemen zur Laufzeit *aligned* werden
- Die jeweilige C–Funktion muss in den Bibliotheken gesucht, gefunden und zuletzt aufgerufen werden. Falls die ganze Bibliothek oder auch nur die Funktion nicht vorhanden ist, sollte das System nicht im Nirwana enden.
- Die Forth–internen Register werden aus den gesicherten Werten wiederhergestellt.
- Zuletzt wird das Ergebnis je nach Typ aus dem wiederum OS–spezifischen Register auf den jeweiligen Forth–Stack gelegt.
- Und das Ganze muss auch noch reentrant–fähig sein

Es hat für iForth eine ganze Weile gedauert, bis das Ganze rund und stabil lief. Jetzt stellt das *dynlibs*–Modul die `Extern:`–Syntax zur Verfügung, *fjack/jacklib.frt* enthält alle benötigten Funktionen.

Spezielle Datentypen

Im fJACK–Client werden eine Reihe von speziellen Datentypen und Strukturen benutzt, dieses führt wie üblich zur besseren Lesbarkeit, kompakterem Code und weniger Bugs. Wir alle wissen inzwischen durch A. Ertls wiederholte Vorträge und unzählige Postings in c.l.f: *State smart is evil!* Auch ich bin schon in vorhandene Fallen getappt und habe stundenlang gegrübelt, warum interaktiv *alles gut* war und im fertigen Programm nicht mehr. Offenbar muss jeder mal durch diese Lektion :-) Auf jeden Fall findet sich in *fjack/types.frt* ein Beispiel, wie um diese Hürde herum gearbeitet werden muss, da der Standard da noch zu wenig präzise ist.

[1] Selbstverständlich unterstützten `Extern:` definierte Worte auch *turnkey Anwendungen*.

Die Callbacks

Callbacks sind sozusagen eine Umkehrung der Funktionsaufrufe; aus beliebigen Funktionen heraus werden die Parameter in die Forth–Umgebung übersetzt, ein Forth–Wort wird aufgerufen und das Ergebnis wieder zurückgegeben. Dummerweise kann dazu aber nicht das normale Forth–System genutzt werden. Es muss eine neue Forth–Umgebung mit zumindest eigener `USER`–Area und privaten Stacks verwendet werden.

Die Callbacks aus dem *jackd* werden letztendlich interrupt–kontrolliert gestartet und müssen innerhalb kürzestmöglicher Zeit – also im µsec–Bereich – die Daten zurückliefern, da ja die Audio–Hardware darauf wartet. Also: langsame Speicherverwaltung, Dateizugriff, Semaphore, Bildschirmzugriff und ähnliche Aktionen mit nicht vorhersehbarer Laufzeit sind für *jackd*–Callbacks absolut tabu! Glücklicherweise sind diese Details für den Benutzer verborgen – ob eine Applikation stabil auf einem Rechner läuft, kann an den XRUNs des *jackd* abgelesen werden, aus der schon oben erwähnten Last geschlossen werden oder der Klang fängt heftig an zu stottern. Ein Callback wird folgendermaßen definiert, hier das Beispiel der XRUN–Zählerei:

```
: XT-COUNT-XRUNS ( parameter -- 0 )
   DROP 1 +TO JACK-XRUNS FALSE ;

' XT-COUNT-XRUNS   CB( _int )CB-int COUNT-XRUNS

fJACK-ID COUNT-XRUNS 0 jack_set_xrun_callback
  ABORT" Can't set xruns callback"
```

`( xt ) CB( ... )CB-int` definiert letztendlich die Callback–Adresse. `_int` sagt, dass zur Laufzeit ein Integer als Parameter übergeben wird, `)CB-int` legt den Rückgabewert des Callback–Aufrufs als einen Integer fest. Die Adresse sowie die ID bekommt dann zuletzt *libjack* mitgeteilt.

Threading

nicht im forthigen (in)direct–threaded–code Sinn sondern wie in multithreading wird im Forth–Standard nicht ausreichend gewürdigt. Fast alle modernen Prozessoren enthalten seit ein paar Jahren mindestens zwei Kerne und diese Tendenz wird sicher weiter zunehmen. Die Taktrate des Prozessors bleibt quasi konstant und die Zahl der Kerne nimmt zu. Im Standard gibt es für den ganzen Bereich des *parallel processing*s quasi keine Unterstützung, die aktuellen großen PC–Forth–Systeme lassen einen da aber alle nicht ganz allein im Regen stehen.

Die relativ umfangreiche Unterstützung im iFORTH–threads–Modul wird in diesem Projekt allerdings gar nicht benutzt, lediglich das Starten eines Forth–Wortes als thread wird vorausgesetzt.

Der fJACK–Client

wird schon beim Laden des Moduls per `include fjack.frt` oder `in fjack` gestartet und ist danach permanent im Hintergrund aktiv. Die Audiodaten werden permanent vom *jackd* an alle verbundenen Programme per Callback weitergereicht, darin werden dazu Zeiger auf interne Speicherbereiche erfragt. Pro Callback werden dann z. B. 512 Audiosamples *präsentiert*. Diese liegen jeweils für einen Kanal als einzelne 32–Bit–sfloats vor. Jeder Client muss in einem Callback

1. Für jeden verbundenen Kanal alle einzelnen Audiosamples einlesen und eventuell bearbeiten
2. die verarbeiteten Audiodaten in Ausgabepuffer schreiben.

Der Callback ist in Wirklichkeit noch deutlich komplexer, da auch andere Daten parallel verarbeitet werden. Es können beliebig große Blöcke von Audiodaten in diversen .wav–Formaten in beliebiger Geschwindigkeit in den Audiodatenstrom eingemischt oder auch aufgezeichnet werden. Dies geschieht so präzise, dass nicht ein einzelnes Sample verloren geht. Einfache Funktionen zum Aufzeichnen oder Abspielen kompletter Wav–Dateien sind vorhanden z. B.:

```
.WAV       ( c-addr u -- )
RECORD-WAV ( time name len -- )
.SOUND     ( samplerate channels 8/16/32/-32 c-addr u -- )
```

`.WAV` spielt eine Standard–WAV–Datei ab und sieht im Header der Datei nach, wieviel Kanäle, welche Samplerate usw. benutzt werden sollen, `RECORD-WAV` zeichnet den laufenden Audiodatenstrom in Stereo und 16–Bit–Format auf. `.SOUND` bekommt dagegen die Samplerate, die Anzahl der Kanäle, den Datentyp und den Filenamen mitgeteilt.

Eine kleine Demo

Unter Linux–iForth lässt sich eine kleine Demo starten, es werden eine Reihe von kurzen Audiodateien in verschiedenen Formaten und Abtastraten abgespielt und in einer graphischen Umgebung gezeigt. Doppelklicks in der Filterleiste (knapp oberhalb des Analysefensters) setzen Filter – links setzt den Hochpass, re einen Tiefpass und mid einen Notch. Im Analysefenster können die Skalen per *Drag* verschoben werden. Abbildung 1 auf Seite 16 zeigt einen Screen–Shot. Einfach ausprobieren ...

Fortsetzung folgt...

Drei Primitives weniger in Willi Strickers Forth–Minimalbasis

Fred Behringer

Die drei Logik–Primitives `INVERT AND OR` in Willi Strickers Minimalsystem (VD–Heft 3/2009) lassen sich einsparen: Man kann schon ohne sie das *Hilfswort* `NAND` als Colon–Definition aufbauen, und aus `NAND` allein lassen sich Willis `INVERT` und `AND` (und auch gleich sein `OR`), wiederum als Colon–Definitionen, herleiten. Damit hat man dann die gesamte Forth–Logik (mit -1 und 0 als Wahrheitswerte, zur Verknüpfung von Flags), aber auch sogar im üblichen Sinne (bitweise, mit 1 und 0 als Bit–Wahrheitswerte, zum Aufbau von Bit–Masken) zur Verfügung. — Der eigentliche Inhalt des vorliegenden Artikels steckt in der Colon–Definition `NAND` am Schluss. Alles andere sind Überlegungen zur Logik.

Logik nur mit `NAND`

Der interessante Artikel von Willi wirft die Frage auf: Kommt man mit weniger Primitives auch noch hin? Antwort: Ja, auf jeden Fall. Ob sinnvoll oder nicht, soll hier nicht diskutiert werden. In einem Lern–System z. B. spielt Zeit für gestiegenen Aufwand keine Rolle: Bei meinem `NAND`–Vorschlag weiter unten sind bei interaktiver Eingabe keine Verzögerungen zu erkennen. — Die Bezeichnung *minimal* verwende ich im Sinne von *je kleiner, desto besser*, nicht unbedingt im wörtlichen Sinne von *kleiner geht es nicht*.

Willi benötigt 26 Primitives. Drei davon sind Logik–Bausteine: `INVERT` und `AND` und `OR`. Bekanntlich kann die zweiwertige zweistellige Logik mit einer einzigen Operation aufgebaut werden: Beispielsweise mit `NAND` (oder aber auch mit `NOR`). Ich entscheide mich im Folgenden für `NAND`. (`NAND` ist uns allen vom IC 74SN00, dem ersten seiner Reihe, her vertraut. Ein aktuelleres Beispiel für den Begriff `NAND` ist das des `NAND`–Flash–Chips bei SSDs [CT]. Führt man `NAND` als zusätzliches Primitive–Wort ein, Dann kann man also den im Stricker–System benötigten Primitives–Wortschatz ohne weiteres Zutun auf 24 Worte reduzieren. (26 ist eine *krumme* Zahl, 24 dagegen lässt sich durch 8 teilen und entspricht damit eher der Bit/Byte–Einteilung im Computer.)

Logik auf Bit– und Byte–Ebene

Zunächst einmal ist zu beachten, dass der Computer, und mit ihm Forth, die übliche zweiwertige Logik auf zwei Ebenen behandelt: Auf Bit– und auf Byte–Ebene. (Ich spreche im vorliegenden Artikel von *Byte*, gehe aber davon aus, dass sich alles in diesem Zusammenhang Interessierende nahtlos auf Doppelbyte et cetera übertragen lässt. Beim Ausprobieren habe ich Turbo–Forth und ZF verwendet. Die *einfachgenaue* Operandenlänge beträgt also bei mir 16 Bit.) Genau genommen, gehört das ANS–Wort `INVERT` nicht zu Turbo–Forth oder ZF. Mit der kleinen Hilfsmaßnahme `: INVERT NOT ;` steht es aber auch in den letzteren Systemen sofort zur Verfügung.

`TRUE` = alle Bits gesetzt?

Im ANS–Dokument [AF] X3J14 (1994) liest man:

> Flags may have one of two logical states, true or false. Programs that use flags as arithmetic operands have an environmental dependency. A true flag returned by a standard word shall be a single–cell value with all bits set. A false flag returned by a standard word shall be a single–cell value with all bits clear.

`TRUE` = ungleich null?

Das auch in ANS–Forth unter Common–Usage zugelassene Wort `NOT` dagegen (in Turbo–Forth und ZF enthalten) ist der rein byteweisen Verwendung zuzuordnen: Es ist äquivalent zum Ausdruck `0=` und macht aus jedem Bytewert ungleich 0 den Bytewert `TRUE`, also -1, eine Zahl, in welcher alle Bits gesetzt sind. Es ist verführerisch, beispielsweise den Bitvektor $AA55$ (= 1010101001010101, little endian) aus der *Magic Number* eines FAT16–Bootsektors für einen anschließenden Branch–Befehl, ohne den Umweg über `0= 0=`, als `TRUE` zu werten. Er würde aber per `INVERT` nicht in `FALSE` übergehen. Die Verwendung von `NOT` im ANS–Sinn, als `: NOT 0= ;` , wäre hier eher angebracht.

`5555 AAAA AND` → `FALSE`

(hexadezimal gesehen), obwohl $5555 \neq 0$, also `TRUE` im üblichen Verständnis, und $AAAA \neq 0$, also ebenfalls `TRUE` ist. Richtig überlegt, müsste man `5555 0= 0= AAAA 0= 0= AND` → `TRUE` argumentieren. Aber so penibel geht ja keiner vor. Nur, wenn es unbedingt nötig ist. Bei meinen weiter unten besprochenen Worten `BOR` und `BNAND`, die in Bit–Logik mit den Wahrheitswerten 1/0 (und nicht mit $-1/0$) arbeiten, wäre mir auch nie eingefallen, `0= 0=` einzusetzen. Hier, bei 5555 und $AAAA$ im Zusammenwirken mit `AND`, würde aber ein Fall vorliegen, bei dem der Einsatz von `0= 0=` nötig wäre.

-1 und Arithmetik

Dass auf Byte–Ebene `TRUE` mit dem arithmetischen Wert -1 (statt etwa mit 1) in Verbindung gebracht wird, ist weniger schlimm: Auf Bit–Ebene könnte man ja auch dazu übergehen, ein gesetztes Bit als (Bitwert) -1 zu interpretieren, und dabei ein gelöschtes Bit weiterhin als

(Bitwert) 0. Man müsste dann die gesamte Arithmetik–Darstellung umwerfen — Logik (auf Bit–Ebene) betreiben (mit den Bit–Wahrheits–Werten −1 und 0) könnte man damit aber auch. Reine Interpretationssache.

INVERT AND OR XOR bitweise

Fest steht: Die in Forth (in ANS–Forth, im Core–Word–Set) zur Verfügung gestellten Logik–Operationen INVERT AND OR XOR arbeiten bitweise, also streng parallel immer auf ein und dieselbe Bitstelle eines Bytes (Komponente eines Bitvektors) bezogen, ohne von einem Bit zum anderen überzugreifen, ganz im Gegensatz zur Arithmetik, wo (im Positionssystem) ständig Übergriffe in Form von Überträgen (*Carrys*) stattfinden.

Mit anderen Worten, wir sprechen beim Computer, auch in Forth, immer von Logik auf Byte–Ebene, meinen aber stets eine in parallelen Bitsträngen (also komponentenweise) stattfindende Logik–Bearbeitung der einzelnen Bits im Byte (oder im Doppelbyte, usw.).

Möglichst wenige Primitives?

Und noch eines scheint mir bei der Frage nach Minimalsystemen von Primitives wichtig: Soll es nur darauf ankommen, ein Forth–System mit möglichst wenigen Primitives hochzuziehen, oder sollen darüber hinaus die verwendeten Primitives gleichzeitig auch noch zum Grundwortschatz des hochzuziehenden Forth–Systems gehören?

Das Letztere tritt wohl bei jedem Minimalsystem in den Hintergrund: Bei Willi Stricker gehören die Systembefehle einerseits und die Lade– und Speicherbefehle für die Parameter– und Return–Stack–Pointer andererseits (obwohl sehr nützlich, wenn man am System experimentieren möchte) nicht zum Sprach–Grundschatz. Er benötigt beispielsweise -PICK als Primitive. Das gibt es aber wohl in kaum einem Forth–System (?), schon gar nicht im ANS–Standard X3J14 (1994) [AF].

So aber kann man den hier zu besprechenden Logik–Befehl NAND auch ansehen, wenn man ihn als Bestandteil der minimalen Primitive–Basis einbringen möchte: Als eigenständiges Forth–Wort zunächst einmal nicht unbedingt nötig, aber als Dreingabe über alle *ANS–Word–Sets* hinaus durchaus nicht zu verachten.

NAND als Primitive

NAND ließe sich in jedem gängigen Forth–System, von jedem Einsteiger, auf High–Level–Ebene sofort als neues Forth–Wort (: NAND AND INVERT ;) programmieren, gehört aber nicht zum ANS–Forth–Reservoir — weder zum Core–Word–Set noch zu irgendeinem anderen der vom ANS–Komitee (bisher) betrachteten Word–Sets. (Auch nicht zu Turbo–Forth oder ZF oder wie die Systeme alle heißen.) Andererseits kann man (auch schon als Einsteiger) zu Prüf– oder/und Emulationszwecken NAND auf dem PC sofort als Primitive (sprich CODE–Definition) z.B. wie folgt ansetzen:

```
CODE NAND ( x y -- fl )
   AX POP
   DX POP
   AX DX AND
   DX NOT
   DX PUSH
   NEXT
END-CODE
```

Ich habe es so unter Turbo–Forth und ZF (16–bittig) für die Zwecke des vorliegenden Artikels zum Ausprobieren verwendet. Die bitweise Wirkung harmoniert mit der bitweisen Wirkung der Maschinenbefehle AND und NOT im (Intel–)Prozessor.

Vektorielle Befehle beim AMD–K6

Übrigens ist die Vorstellung von *vektoriellen* Befehlen, also von Befehlen, die auf die einzelnen Komponenten von Vektoroperanden, ohne Komponenten–Übergriffe, gleichzeitig (*parallel*) wirken, auf *noch höherer Ebene* auch von den Multimedia–Befehlen beim Intel–Pentium–Prozessor her geläufig (MMX beim Pentium, XMM beim Pentium III). Beim AMD–K6 kennt man dafür die 3DNow!–Befehle: Es werden dort mit einem einzigen Additions–(oder was auch immer)–Befehl immer gleich mehrere Additionen (oder was auch immer) gleichzeitig getätigt, die je zu je innerhalb separater Kanäle wirken, ohne dass Überträge oder sonstige Verkoppelungen von einem Kanal zum anderen stattfinden.

Wahrheitstafel

Die gleich folgende Wertetabelle ist selbsterklärend und macht weitere Kommentierung überflüssig. Es ist also nicht unbedingt nötig (kann aber auch nicht direkt schaden), den eigenen Logik–Hintergrund beispielsweise über [GB] abzuchecken. In der Wertetabelle in Bild 1 auf der nächsten Seite sollen x und y eine willkürlich herausgegriffene Komponente der entsprechenden Bitvektoren (egal, welcher Vektor–Länge) und die Operationen AND OR INVERT NAND XOR die zugehörigen Bitanteile der (bitweise wirkenden) Forth–Worte gleichen Namens bedeuten. Die Werte 0 und 1 sind also Bitwerte (Wahrheitswerte auf Bitebene). Man kann diese Wahrheitstafel auch als Inbegriff der Funktionsweise meiner unten angeführten Worte BNAND BINVERT BAND BOR auffassen, die nur ein einziges Bit (das lsb, angezeigt durch das Carry–Flag CF) berücksichtigen.

NAND als Primitive genügt

Die folgenden drei Zeilen zeigen, dass man die 3 Forth–Worte INVERT AND OR aus dem einzigen Wort NAND heraus erzeugen kann (Beweis durch vollständige Aufzählung aus der Tabelle heraus). NAND kann also als Primitive für die Erzeugung sämtlicher Logik–Operationen dienen. Mit anderen Worten, die drei Primitives INVERT AND OR in Willi Strickers Minimalsystem können durch das eine Primitive NAND ersetzt werden, was zwei Primitives einspart.

x	y	x y AND	x y OR	x INVERT	x y NAND	x y XOR
0	0	0	0	1	1	0
1	0	0	1	0	1	1
0	1	0	1	1	1	1
1	1	1	1	0	0	0

Bild 1: Wahrheitswerte der Bitkomponenten von AND OR INVERT NAND XOR

```
: INVERT ( x   -- fl )
    -1 NAND ;

: AND    ( x y -- fl )
    NAND INVERT ;

: OR     ( x y -- fl )
    \ De-Morgan-Theorem
    INVERT SWAP INVERT AND INVERT ;
```

So, wie sie da stehen, arbeiten alle vier Forth–Worte, NAND INVERT AND OR, bitweise (mit 1 und 0 als Bit–Wahrheitswerte). Aber gleichzeitig, und das wird bei ihrer üblichen Interpretation als Flagwert–Verknüpfung auch meist getan, können sie auch als Byte–Operationen aufgefasst werden. Unter *Byte* verstehe ich dabei auch *16–Bit* (Wort im 16–Bit–System) oder *32–Bit* (Doppelwort im 16–Bit–System, Wort im 32–Bit–System) usw. Als Wahrheitswerte werden dann *normalerweise* die arithmetisch interpretierbaren Werte −1 (alle Bits gesetzt) und 0 (alle Bits gelöscht) genommen. Man ersetze in Bild 1 alle Einsen durch −1 und kommt schnell dahinter, dass man die zweiwertige Logik (die Aussagenlogik) auch mit den Wahrheitswerten −1 und 0 arithmetisieren kann.

NAND ist nicht ANS

Ein *Schönheitsfehler* bei NAND besteht nur darin, dass dieses Forth–Wort selbst nicht zum üblichen Forth–Wortschatz gehört. Im ANS–Standard wird es nicht erwähnt (?), nicht einmal in einem der *Extended*–Sets. Aber das gilt ja auch schon für die bei Willi Stricker als Primitives verwendeten Worte SP@ SP! RP@ RP!. Auch diese kommen, obwohl per se schon äußerst nützlich, in ANS–Forth nicht vor.

XOR sehr wohl

Andererseits ist das Logik–Wort XOR in den üblichen Forth–Systemen sehr wohl enthalten. Im ANS–Standard gehört es sogar zum Core–Word–Set. XOR wird bei Willi Stricker im Kernel (über das De–Morgan–Theorem durch Rückgriff auf INVERT und AND und OR [sic!]) wie folgt definiert:

```
: XOR    ( x y -- fl )
    OVER OVER INVERT AND >R
         SWAP INVERT AND R> OR ;
```

Bei der obigen CODE–Definition für INVERT gehe ich davon aus, dass ein Mechanismus zum Einbringen von Zahlen schon außerhalb der Basis–Primitives zur Verfügung steht (Metacompiler?). (An sich ist es ja nur die Ganzzahl −1, welche bei den Definitionen von INVERT AND OR XOR im Kernel verfügbar sein muss.) Zumindest bei dem von Willi Stricker angegebenen Forth–Kernel müsste das der Fall sein. Wenn das aber schon der Fall ist, dann kann man sich die Logik–Operationen auch auf arithmetischem Wege besorgen. (Ich habe im VD–Heft 3/2001 [FB] über Logik–Arithmetisierung geschrieben.) Überspitzt gesagt, könnte man dann auch noch auf NAND als Primitive verzichten. Hier die vier wesentlichsten Logik–Operationen in arithmetischem Gewand (man überzeuge sich von der Richtigkeit meiner Behauptung anhand einer Tabelle, die der Wertetabelle aus Bild 1 entspricht, in der man aber überall 1 durch −1 ersetzt hat, so dass man sie zur Festlegung einer Byte–Logik nehmen kann):

```
: INVERT ( x   -- fl )              -1 *        1 - ;
: AND    ( x y -- fl )            * -1 *            ;
: OR     ( x y -- fl ) OVER OVER + >R *        R> + ;
: XOR    ( x y -- fl ) OVER OVER + >R * DUP + R> + ;
```

Dieses INVERT entspricht von der Funktion her dem Ausdruck : INVERT NEGATE 1- ; und arbeitet bekanntlich bitweise. Aber Achtung: Für die so gefassten Worte AND OR XOR kann das bitweise Arbeiten nicht behauptet werden! Für sie müsste man sich auf die Byte–Wahrheitswerte −1 und 0 als Eingaben beschränken. Gegenbeispiele:

7 7	XOR	B. →	0000000001110000
7 7	AND	B. →	1111111111001111
7 7	OR	B. →	0000000000111111

Bei AND und OR ist (in dieser Fassung) die logische Idempotenz verletzt, bei einem bitweisen XOR (auf zweimal denselben Operanden angewandt) müsste man grundsätzlich 0 erwarten können (bekannter Programmiertrick zur Erzeugung einer Nullbelegung). Für eine Verwendung *nur* als Byte–Logik (Wahrheitswerte −1/0) läuft dagegen alles, wie man sich schnell überzeugt, bestens.

INVERT kommt in Willi Strickers Kernel als bitweise Operation wesentlich nur bei AND in der Definition von 0< und bei OR in der Definition von 2/ vor. Allerdings käme man dort ohne das *bitweise* nicht aus. Das sind aber nur wenige Operationen, solche, bei denen man in Willis Kernel das Vorzeichenbit herausschälen muss. Ich werde gleich zeigen, dass man das Vorzeichenbit auch über +C und U2/C bekommen kann.

Es ging mir um die Frage,

ob man die Logik–Operationen von Willi Stricker in den Colon–Definitionen des Kernels aus den Primitives heraus herleiten kann, wenn man auf INVERT AND OR in den

Primitives verzichtet. Die arithmetische Operation + ist nicht kritisch. Sie enthält nur die Primitives `+C` und `DROP`. Es reduziert sich also alles auf die Frage, ob das mit der `*`–Operation schon aus Willis Kernel heraus ohne die wie üblich bitweise wirkenden Worte `INVERT AND OR` geht oder ob man etwa `*` (oder `M*` oder `UM*`) dann vielleicht als Primitive einführen müsste.

Multiplikation als Addition

Natürlich muss das möglich sein: Multiplikation ist ja nichts weiter als verkürzte Darstellung eines bestimmten Additionsschemas. (Ob Multiplikation per schrittweiser Addition sinnvoll ist oder nicht, ist dabei nicht die Frage. Es geht hier nur um die Reduzierung der Anzahl der Primitives.) Willi Stricker macht genau das: Er addiert bei `UM*` den einen Faktor so oft zum schon erreichten (doppeltgenau angesetzten) Zwischenergebnis, wie der zweite Faktor hergibt.

Beschaffung des Vorzeichens

Man könnte `UM*` also weiter nach vorn schieben, zur Kernel–Zeile 20, und hätte dann die Multiplikation zur Definition der Logik–Operationen in Willis Kernel ohne dessen Primitives `INVERT AND OR` rechtzeitig zur Verfügung. `0<` könnte man sich auch noch schnell beschaffen:

```
: 0< ( x -- fl )
   \ fl = TRUE für x kleiner 0
   DUP +C SWAP DROP IF -1 ELSE 0 THEN ;
```

Hier kommt `+C` zum Zuge, das von Willi Stricker (zusammen mit `U2/C`) als Primitive eingeführt wird, *da Forth kein Flag–Register kennt.* Ich komme auf `+C` (und `U2/C`) weiter unten zu sprechen.

Forth kennt zwar Flags,

aber nicht auf Bit–Ebene. Auf die Bit–Flags SF, ZF, OF, CF (ich spreche vom PC), die für `IF–THEN` etc wichtig sind, könnte man zwar über `CODE`–Definitionen sofort zugreifen, aber in Minimalsystemen sollen ja `CODE`–Definitionen (zunächst einmal) nicht eingesetzt werden. Willi macht mit seinen Untersuchungen interessanterweise darauf aufmerksam, dass man mit `+C` und `U2/C` alle Bit–Flag–Fragen auch schon in High–Level–Forth erschlagen kann.

Das Schöne an der Arithmetisierung

der Logik–Operationen (auf Byte–Ebene) ist die Tatsache, dass man auch in noch so komplexen logischen Ausdrücken mit den arithmetischen Grundoperationen `+ - *` auskommt. Beweis durch (umkehrbare) Umwandlung der Disjunktiven Normalform (einer beliebigen Logik–Funktion) in eine Multilinearform. Genaueres siehe [FB],[BJ].

Erzeugung über IF ELSE THEN

`BEGIN WHILE REPEAT UNTIL AGAIN IF THEN ELSE` kommen in Willi Strickers Kernel nicht vor. Sie werden aber überall im Kernel intensiv verwendet. Ich vermute, dass sie per `BRANCH` und `?BRANCH` erzeugt werden sollen und dass für sie die Bemerkung „Er (der Kernel–Vorschlag) erhebt aber keinen Anspruch auf Vollständigkeit!“ in Willis Artikel gilt.

Mit Hilfe solcher Verzweigungs–Operatoren gelingt es leicht, die Logik–Operationen `INVERT AND OR XOR` (und damit die gesamte zweistellige (Flag–)Logik) auf wiederum andere Weise zu erzeugen — allerdings (zunächst einmal) wieder nur auf Byte–Ebene (für Flag–Verknüpfungen brauchbar, aber nicht bitweise wirkend).

Das Wort `0=` (äquivalent zum Ausdruck `IF 0 ELSE -1 THEN`) kann aus Willis Kernel–Zeile 39 herausgeholt und an den absoluten Anfang des Kernels gestellt werden. Damit stehen die folgenden Flag–Logik–Worte von Anfang an zur Verfügung — ohne dass dazu ein Primitive `NAND` nötig wäre.

```
: INVERT ( x   -- fl )
    0=                                ;

: AND    ( x y -- fl )
    0= 0= IF 0= 0= ELSE DROP  0 THEN ;

: OR     ( x y -- fl )
       0= IF 0= 0= ELSE DROP -1 THEN ;

: XOR    ( x y -- fl )
       0= IF 0= 0= ELSE      0= THEN ;
```

Aber Vorsicht wiederum vor diesen Fassungen (Beispiele alles andere als bitweise!):

7	INVERT	→	0
7 7	AND	→	-1
7 7	OR	→	-1
7 0	XOR	→	-1
0 7	XOR	→	-1

Eine weitere Möglichkeit

zur Definition von Logik–Operationen wäre über `MAX` und `MIN` gegeben, wozu dann keine Multiplikation nötig wäre. Allerdings würden dann wiederum `<` und `>` und `-` Sonderüberlegungen erfordern. Man überzeugt sich leicht von der Gültigkeit der folgenden Worte innerhalb der Byte–Logik (mit `-1` und `0` als Wahrheitswerte).

```
: INVERT ( x   -- fl )
                 >R  -1 R> - ;
: AND    ( x y -- fl )
             MAX             ;
: OR     ( x y -- fl )
             MIN             ;
: XOR    ( x y -- fl )
   OVER OVER MAX >R MIN R> - ;
```

Allerdings gilt (auch) hier weder AND noch OR noch XOR bitweise:

2 1	AND(herkömmlich)	→	0 ,
aber 2 1	MAX	→	2
2 1	OR(herkömmlich)	→	3 ,
aber 2 1	MIN	→	1
2 1	XOR(herkömmlich)	→	3 ,
aber 2 1	OVER OVER MAX >R MIN R> -	→	-1

Andererseits wirkt das eben definierte INVERT durchaus bitweise, was aus folgendem bekannten Zusammenhang folgt:

```
INVERT = NEGATE 1 - = NEGATE 1 NEGATE +
```

Noch andere Möglichkeiten

Man überzeuge sich davon, dass auch die folgenden Worte das Erwartete liefern:

```
: OR     ( x y -- fl )
   \ nicht bitweise
   0= 0= IF -1 THEN ;

: OR     ( x y -- fl )
   \ nicht bitweise und nur
   \ fuer Floored-Arithmetik [RZ]
   + 2/             ;

: OR     ( x y -- fl )
   \ nicht bitweise
   +C +             ;

: INVERT ( x   -- fl )
   \       bitweise
   -1 SWAP -        ;
```

Logik und Zuverlässigkeit

Es folgt ein sicher nicht zu schweres, aber (hoffentlich) auch nicht zu leichtes Beispiel aus der Zuverlässigkeitstheorie.

Angenommen, ein dreimotoriges Flugzeug (mit den Motoren von links nach rechts x, y, z) ist intakt (bleibt in der Luft, stürzt nicht ab), wenn der mittlere Motor (also y), oder ansonsten mindestens zwei Motore intakt sind. Anders ausgedrückt: Intakt sei das Flugzeug, wenn y oder (x und y) oder (x und z) oder (y und z) oder (x und y und z) intakt sind. Für die Abhängigkeit des Intaktseins des Flugzeugs (fl) als Gesamtsystem vom Intaktsein der Motoren gelte also:

```
fl(x,y,z)
  = y oder/und (x sowohl-als-auch z)
        (Flugzeug, Infix-Notation)
  = y x z AND OR
       (Flugzeug, übliche Forth-Logik, bitweise)
  = y x z * -1 * OVER OVER + >R * R> +
       (Flugzeug, Forth-Logik arithmetisiert),
```

wobei x, y, z und auch $fl(x, y, z)$ die beiden (Forth–Wahrheits–)Werte -1 (TRUE) oder 0 (FALSE) annehmen können.

Multilinearform und der *große Durchblick*

Zugegeben, die arithmetisierte Form (unter alleiniger Verwendung von + - *) fördert den Durchblick keinesfalls. Das ist aber auch nicht ihre Bestimmung. Schließlich würde man ein Polynom sagen wir vierten Grades in Forth–Notation auch kaum wiedererkennen. Fest steht jedenfalls, dass man in der arithmetisierten Form keine Logik–Operatoren mehr braucht. Und somit könnte man auf Logik–Operatoren als Primitives in Minimalsystemen verzichten — als Primitives! — wenn man die arithmetischen Operatoren + - * aus den restlichen Primitives herleiten kann oder sie ganz oder teilweise als Primitives einführt. Im Kernel könnte man dann ja die bitweisen Logik–Operatoren (nachträglich) als abgeleitete Colon–Definitionen bereitstellen. Den Kernel zu minimieren, war bei Willi Stricker und ist im vorliegenden Artikel nicht das Problem. Was minimiert werden sollte, war die Zahl der benötigten Primitives.

Bleibt also (für mich) die Kardinalfrage:

Kann man im Minimalsystem von Willi Stricker die arithmetischen Operationen + - * schon aus einem Minimalsystem herleiten, das keine der Logik–Operationen INVERT AND OR XOR als Primitives enthält? Konkret gesagt, kommt man bei den Logik–Operationen vielleicht sogar auch noch ohne NAND aus? Und die Krönung wäre dann die Frage, ob das eventuell gleich auch noch für eine bitweise wirkende Logik gelten kann. Beides kann mit Ja beantwortet werden (siehe gleich und weiter unten).

Willi Strickers Primitives?

Ich muss zur Überprüfung der noch folgenden Überlegungen auf Willis Primitives +C und U2/C konkret zurückgreifen können. Also richte ich mir diese beiden Primitives ganz schnell für Turbo–Forth und ZF (auf dem PC) als CODE–Definition her:

```
CODE +C ( x y -- x+y carry )
        AX POP
        DX POP
        CX CX XOR
        AX DX ADD
        CX CX ADC
        DX PUSH
        CX PUSH
        NEXT
END-CODE

CODE U2/C ( x -- carry x/2 )
        AX POP
        CX CX XOR
        AX SHR
        CX CX ADC
        CX PUSH
```

```
        AX PUSH
        NEXT
END-CODE
```

Und es geht auch ohne

ein Hilfswort `NAND` als Primitive in Willi Strickers Minimalsystem (wie ich es eingangs noch voreilig gefordert hatte) bei gleichzeitigem Überbordwerfen von INVERT und `AND` und `OR` als Primitives. Genauer gesagt, ich schaffe es, in Willis Minimalsystem drei (nicht nur zwei) Primitives einzusparen, also das System mit 23 (statt mit 26) Primitives aufzubauen. Dazu darf ich ähnlich wie Willi in seinem RSHIFT (Kernel–Zeile 76) vorgehen und von den Strukturelementen `BEGIN WHILE REPEAT` Gebrauch machen, ohne über deren Herkunft weiter nachzudenken. (`2*` wird durch `DUP +` ersetzt.) Ich gehe (für die Eingaben x und y) wie folgt vor: Bit für Bit

1. verschiebe ich (bei 16–Bit–Systemen 16–mal) x und y per `U2/C` um jeweils ein Bit nach rechts,
2. lasse ich auf die aus U2/C gewonnenen x– und y–Carry–Bits die Operation `NAND` in Bit–Logik (`BNAND` mit Wahrheitswerten 1/0) wirken,
3. addiere ich in Abhängigkeit vom Bit–Logik–Ergebnis (bei Carry = 1 ja, bei 0 nein) die nächste Zweierpotenz auf einen *Akkumulator* auf dem Stack,

und fertig ist das Ergebnis: Ein bitweise (!) wirkendes `NAND` in High–Level–Forth ohne INVERT oder `AND` oder `OR`, allein mit den dann noch übrig bleibenden 23 Primitives aus Willi Strickers Minimalsystem. Dass aus diesem `NAND` dann der ganze Rest einer bitweise wirkende Logik hergeleitet werden kann, habe ich eingangs besprochen. (Unter Bit–Logik verstehe ich, wie oben schon gesagt, eine auf eine beliebig herausgegriffene Bitstelle wirkende Logik mit den Wahrheitswerten 1/0. Ich darf zuerst die von mir jetzt benötigten Bestandteile der Bit–Logik besprechen.

Bit–Logik

Um deutlich zu machen, dass es sich hier um Bit–Logik (nicht um Byte–Logik) handelt (mit der ich auf Carry–Werte (1/0) eingehen möchte), setze ich für den Moment vor die Logik–Bezeichnungen jeweils ein `B`. Entsprechend dem De–Morgan–Theorem darf ich schreiben:

```
: BNAND ( x y -- fl )
          IF 0 ELSE 1 THEN
    SWAP  IF 0 ELSE 1 THEN  +
          IF 1 ELSE 0 THEN ;
```

Oder einfacher, wenn ich `IF–ELSE–THEN` schachtele:

```
: BNAND ( x y -- fl )
    IF
       IF 0 ELSE 1 THEN
    ELSE
       DROP 1
    THEN ;
```

Einfacher im Sinne des Aufwands (nur noch 10 statt 17 Worte), nicht in Bezug auf die Anschaulichkeit. Beide Fälle liefern:

1 1 BNAND	→	0
1 0 BNAND	→	1
0 1 BNAND	→	1
0 0 BNAND	→	1

`BNAND` ist wesentlicher Bestandteil des gleich zu besprechenden bitweisen `NAND` s. Ganz schnell noch ein paar Zusammenhänge aus der Bit–Logik, die aber für das angestrebte Byte–Logik–`NAND` nicht unbedingt benötigt werden.

```
: BINVERT 1 BNAND                          ;
: BAND    BNAND BINVERT                    ;
: BOR     BINVERT SWAP BINVERT BAND BINVERT ;
```

oder leichter zu durchschauen:

```
: BINVERT IF 0 ELSE 1 THEN ;
: BOR   + IF 1 ELSE 0 THEN ;
```

Bei Bit–Logik erübrigt sich die Frage nach *bitweise* oder *nicht bitweise*. Man mache sich aber klar, dass die (von mir absichtlich so konstruierten) `B`–Worte ihre Logik–Funktion nur dann richtig ausüben, wenn sich die Eingaben auf 1 und 0 beschränken. Das ist insbesondere dann der Fall, wenn, wie bei `U2/C`, als Resultat nur das *kleinste* Bit ins Spiel kommt. Beispiel: `(5 3 BNAND) = (101 11 BNAND) = 0`, was weit von einer bitweisen Wirkung im Sinne einer Byte–Logik entfernt ist.

`INVERT AND OR` als Primitives streichen

und ein bitweises `NAND` im High–Level–Forth des Kernels von Willi Stricker unter alleiniger Verwendung der restlichen 23 Primitives aus Willis Artikel aufbauen. Ich darf es wie folgt versuchen:

```
HEX
: NAND ( x y -- z ) \ z = (x bitweise nand y)
    -10 SWAP >R SWAP R> U2/C >R
    SWAP U2/C >R BNAND 1 SWAP  \ 1.Stufe
    IF 1 ELSE 0 THEN R> R>
    BEGIN                      \ 2.-16.Stufe
      U2/C >R >R U2/C R> SWAP >R BNAND >R >R
            DUP + R> R> IF OVER + THEN >R >R
      1 + DUP
    WHILE
      R> R> R> R>
    REPEAT
    R> R> R> R>
    DROP DROP >R DROP DROP R> ;
```

Ich habe die erste Stufe von den übrigen abgesetzt, da ich so bei den aufzuakkumulierenden Zweierpotenzen auch die nullte Potenz ($2^0 = 1$) am saubersten berücksichtigen konnte.

Zur Überprüfung überzeuge man sich, dass:

-1	-1	NAND	→	0
-1	0	NAND	→	-1
0	-1	NAND	→	-1
0	0	NAND	→	-1

Und dieses `NAND` gilt (nach Konstruktion!) tatsächlich bitweise! Als Plausibilitäts–Beispiel diene `7 2 NAND`:

```
7 NAND = 0000000000000111 NAND
2        0000000000000010
       = 1111111111111101 = -3
```

Besser machen?

Diese `NAND`–Operation hätte man sicher kompakter halten können: Man könnte es mit Willi Strickers Primitives `PICK` und `-PICK` versuchen.

Weiter oben habe ich auf die bekannte [GB] Herleitung der (zweiwertigen) Logik aus der Verknüpfung `NAND` heraus hingewiesen. Dass das (auch) bei den von Willi Stricker als Primitives verwendeten drei Worten `INVERT` `AND` `OR` bitweise wirkt (wobei man überdies auch das forth–übliche XOR bitweise einbeziehen kann), wenn das *Urwort* `NAND` bitweise wirkt, macht man sich klar, wenn man die Zusammenhänge

```
: INVERT ( x   -- fl )
    -1 NAND ;

: AND    ( x y -- fl )
    NAND INVERT ;

: OR     ( x y -- fl )  \ De-Morgan-Theorem
    INVERT SWAP INVERT AND INVERT ;

: XOR    ( x y -- fl )
    OVER OVER INVERT AND >R
         SWAP INVERT AND R> OR ;
```

mit *beliebigen Byte*–Werten für x und y (in Turbo–Forth und ZF 16–bittig) verwendet und fl auf bitweise Richtigkeit überprüft (vollständige Enumeration [Fleißarbeit] oder abkürzendes Verfahren [Aufgabe!] – vielleicht auch Programm [Entwicklungssache]). Überflüssig zu betonen, dass hier (bei bitweisem `NAND`) fl natürlich nicht auf die beiden Werte -1 ($=$ 1111111111111111) oder 0 ($=$ 0000000000000000) beschränkt bleibt und die Byte–Logik *unlogische* Ergebnisse liefern kann, wenn man *jeden* Byte–Wert ungleich `0` als `TRUE` interpretiert.

Literatur

[AF] ANS–Forth–Standard: The American National Standard for the Forth language (ANSI X3J14:1994).

[BJ] Behringer, Fred and Chris Jakeman: Arithmetized Logic in Forth — An Introduction. Forthwrite 112, Juli 2001.

[CT] c't 24/2009.

[FB] Behringer, Fred: Lego–Roboter und arithmetisierte Logik in Forth. Vierte Dimension, Heft 3/2001, S.8–12.

[GB] Böhme, Gert: Einstieg in die Mathematische Logik, Hanser–Verlag 1981.

[RZ] Zech, Ronald: Forth 83, Franzis–Verlag 1987.

Abbildung 1: Ein Blick auf den fJACK–Client (Der Artikel beginnt auf Seite 7.)

Forth im FPGA — Teil 2, b16

Ulrich Hoffmann

In diesem zweiten Artikel über Forth im FPGA widmen wir uns der Frage, wie Bernd Paysans b16-Prozessor [2] auf dem Altera–DE1-Entwicklungsboard, das wir in Teil 1 vorgestellt haben, realisiert wird.

Dann wollen wir einfache Beispiel–Programme für `b16` programmieren und den Assembler benutzen, daraus `b16`–Maschinencode zu erzeugen und diesen ablaufen zu lassen. Die auf dem DE1–Board verfügbare 7–Segment–Anzeige wollen wir dabei als Debug–Hilfe einsetzen.

b16

Der b16–Prozessor lehnt sich in seiner Architektur an den `c18`–Chip [1] von Chuck Moore an: Eine Stack–Maschine in von Neumann–Architektur (gemeinsamer Speicher für Instruktionen und Daten) mit zusätzlichem separaten Daten– und Return–Stack und einem reduzierten Instruktionssatz von nur 32 Instruktionen.

Eine Besonderheit ist die Art, wie Instruktionen kodiert sind: In einem Speicherwort werden mehrere Instruktionen in sogenannten *Slots* gepackt dargestellt. Benötigt eine Instruktion Operanden, so kann sie nur in den ersten Slots auftreten, die restlichen Bits des Speicherwortes bilden dann die Operanden. Operationen auf dem Stack — wie `and`, `+` oder `drop` — benötigen keine Operanden und können daher in jedem Slot stehen.

Bernd Paysan hat vom b16 zwei Varianten vorgestellt:

1. Der ursprüngliche b16 [2], der ein Adress–Register `A` im Stile des `c18` zur Adressierung von Speicherzellen besitzt und bei dem `NOS`, das zweite Element des Datenstacks, und `R`, das oberste Element des Return–Stacks, explizit als Register ausgeführt sind.
2. b16-small [3], eine weiter vereinfachte Version, bei der `NOS` und `R` in die jeweiligen Stacks integriert und `A` entfallen ist. Auch unterscheiden sich bedingte Sprünge von denen im b16 dadurch, dass sie wie in Forth das oberste Datenstack–Element entfernen.

Abbildung 1 zeigt den Aufbau von b16-small (Das Register P ist der Programmzähler, der auch als nulltes Element des Return-Stacks angesehen werden kann).

Weder `b16` noch `b16-small` (oder `c18`) haben die Instruktion `swap`. Statt dessen verwendet man zunächst `over` und dann später `nip`, um das überschüssige Stack–Element zu entfernen. Als Phrase ließe sich das als `over >r nip r>` formulieren.

Wir wollen uns im Folgenden auf `b16-small` konzentrieren und einige seiner Eigenheiten ansprechen. Für eine vollständige Beschreibung inklusive komplettem Verilog–Quellcode sei auf Bernd Paysans b16-small-Artikel [3] verwiesen. b16-small besitzt 32 Instruktionen, die in 5 Bits dargestellt werden. Der Instruktionssatz ist in Abbildung 2 auf der nächsten Seite zu sehen.

Ein b16-small–Speicherwort ist 16 Bit breit. Es wird in 4 Slots aufgeteilt, die 1 Bit und drei mal 5 Bits haben. Die obersten 4 Bits des ersten Slots werden implizit als 0 angenommen, womit darin also nur eine `nop` oder `call`–Instruktion dargestellt werden kann.

implizit	Slot 0	Slot 1	Slot 2	Slot 3
0000	a	bbbbb	ccccc	ddddd

b16-small–Sprungbefehle sind bedingte und unbedingte Sprünge und Unterprogrammaufrufe. Sprungbefehle, die in Slot 0, 1 oder 2 stehen, nehmen ihr Sprungziel aus den

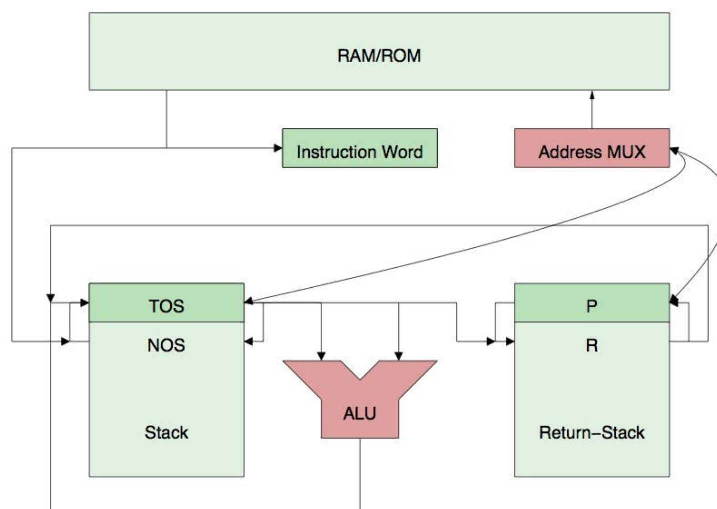


Abbildung 1: Blockschaltbild des b16-small–Prozessors [3]

	0	1	2	3	4	5	6	7	Kommentar
0	nop	call	jmp	ret	jz	jnz	jc	jnc	Sprungziel im Rest der Instruktion
	nop	exec	goto	ret	gz	gnz	gc	gnc	für Slot 3
8	xor	com	and	or	+	+c	2/	c2/	
10	!+	@+	@	lit	c!+	c@+	c@	litc	
	!.	@.	@	lit	c!.	c@.	c@	litc	für Slot 1
18	nip	drop	over	dup	>r	nop	r>	nop	

Abbildung 2: Die Instruktionen des b16-small–Prozessors [3]

restlichen Bits des Instruktionswortes: die entsprechenden Bits im Befehlszählregister werden ersetzt. Tritt ein Sprungbefehl in Slot 3 auf, so erwartet er das Sprungziel auf dem Datenstack.

Die Auto–Inkrement–Speicherbefehle `!+ ( val addr - addr' )` und `@+ ( addr - cal addr')` (sowie ihre Byte-Varianten `c!+ c@+`) verwerfen die Speicheradresse nicht, sondern erhöhen sie auf die nächste Adresse. Das Auto-Inkrement wird allerdings unterdrückt, wenn diese Befehle in Slot 1 auftreten (In Slot 0 können ja nur `nop` oder `call` stehen).

b16-small programmieren

Bernd Paysan hat basierend auf gforth für b16-small einen Assembler geschrieben. Ihn und das gesamte b16-small–Projekt stellt Bernd über das Subversion–Repository der Forth–Gesellschaft zur Verfügung [4]. Von dort lässt es sich bequem auf den eigenen Computer bringen[1]:

```
svn co http://www.forth-ev.de/repos/b16-small
```

Wir finden dann die b16-small–Verilog–Beschreibung (.v–Files), den Assembler (`b16.fs`), Dokumentation (`.lyx` und `.pdf`) sowie Beispielprogramme (`.asm` unter anderem auch die Triceps-Anwendung, die wir 2009 auf dem Linux-Tag in Berlin präsentiert haben).

Wir wollen ein ganz einfaches b16-small–Programm schreiben und uns die Eigenheiten des Assemblers ansehen, der eng an Forth angelehnt ist. Unser *Hallo b16*–Programm soll die Zahlen 3 und 4 zusammenzählen und dann in einer Endlos–Schleife enden, die das Resultat immerwährend um eins vermindert — so ein b16–Programm kann ja nicht einfach anhalten. Wir schreiben Folgendes:

```
#! /usr/bin/gforth b16-asm.fs

$2000 org

: boot
    3 #c 4 #c +

    BEGIN -1 # +
    AGAIN
;;

$3FFE org
     boot ;;

\ print verilog hex for $2000 bytes
$2000 $2000 .hex b16.hex
$2000 $2000 .hexh b16h.hex
$2000 $2000 .hexl b16l.hex

.end                        \ end of test program
```

Der `org`–Befehl legt fest, an welcher Speicherzelle mit dem Assemblieren begonnen werden soll.

Unterprogramme werden mit `:` eingeleitet. Wird der so definierte Name später verwendet, so wird ein Unterprogrammaufruf assembliert.

Anders als in Forth, aber so wie im klassichen Forth–Assembler, gibt es auch in diesem Assembler keinen Compiler/Interpreter–Zustand. Es wird immer interpretiert. Das ist vermutlich auch der Grund, warum Literale explizit mit `#` (oder wenn es Byte–Literale sind mit `c#`) assembliert werden und dies nicht implizit geschieht. Im b16-small–Maschinencode werden die Literale in nachfolgenden Programmzellen abgelegt und bei der späteren Ausführung von dort auf den Stack geladen. Der Programmzähler P wandert also weiter, wenn im aktuellen Instruktionswort `lit`–Befehle ausgeführt werden.

Der Assembler unterstützt auch Schleifen und Bedingungen in klassischer Forth–Art. Wir verwenden hier eine `BEGIN AGAIN`–Endlosschleife, um vom Wert auf dem Stack immer wieder 1 abzuziehen.

Da b16-small keine Subtraktions–Instruktion hat, wird hier -1 addiert, um den Wert auf dem Stack um 1 zu vermindern. Wir könnten statt dessen auch das Macro

```
macro: - com +c end-macro
```

definieren, das die Subtraktion durch Additon des 2er–Komplements realisiert, wobei ausgenutzt wird, dass `com` das Carry–Bit setzt. Diese und viele andere nützliche Macro–Definitionen finden sich im File `b16-prim.asm`.

Auch das Wort `;` verhält sich anders als in Forth: Es kompiliert im Prinzip nur eine `ret`–Instruktion und kann also auch mitten im Wort stehen. Zusätzlich führt `;` auch eine *Tail–Call–Optimierung* durch, die abschließende Unterprogrammaufrufe in Sprünge umwandelt, um

[1] Subversion gehört unter Mac OS X und Linux in der Regel zum Lieferumfang, unter Windows ist die Implementierung TortoiseSVN [5] (`http://tortoisesvn.tigris.org/`) weit verbreitet, bei dem dann das Importieren über den Windows-Explorer geschieht.

den Return–Stack weniger zu belasten. Das Wort `;;` beendet eine Definition, ohne ein `ret` zu assemblieren, und sorgt dafür, dass das letzte Instruktionswort ggf. mit `nop`–Instruktionen aufgefüllt wird.

Unser Programm wird ab $2000 assembliert, da der b16-small so konfiguriert ist, dass dort bis `$3FFF` Boot-RAM liegt (vgl. `b16-top.v`):

```
...
always @(r or w or sel or addr_i or SRAM_DQ)
 begin
  data <= 0;
  casez({ r, sel })
    4'b1100: data <= sfr_data;
    4'b1010: data <= { bootramh[addr_i[12:1]],
                       bootraml[addr_i[12:1]] };
    4'b1001: data <= SRAM_DQ;
  endcase // case(sel)
end

always @(addr)
 if(addr[15:8] == 8'hff)      sel <= 3'b100;
 else if(addr[15:13] == 3'h1) sel <= 3'b010;
 else                         sel <= 3'b001;
...
```

Den anderen Speicherbereichen ist Peripherie (sfr=Special Function Register, `$FF00` bis `$FFFF`) und Block–RAM (`$0000` bis `$1FFF` und `$4000` bis `$FEFF`) zugeordnet.

Bei einem Reset führt b16-small seine erste Instruktion an der Adresse `$3FFE` aus. An dieser Stelle assemblieren wir einen Unterprogrammaufruf zu unserem Programm `boot`.

Das Programm wird mit `./b16-asm hello-b16.asm` übersetzt und erzeugt unter anderem die beiden Files `b16h.hex` und `b16l.hex`, in denen die höher– und niederwertigen Bytes des b16-small–Maschinenprogramms liegen. Wir können uns aber auch das File `b16.hex` ansehen, das noch 16–Bit–Worte enthält. Vielleicht ist es ganz interessant, anhand der Befehlstabelle einmal selbst die Instruktionen zu dekodieren? (Sprünge gehen bei b16-small immer an gerade Adressen, das Sprungziel in Sprung–Instruktionen wird um ein Bit nach links geschoben.)

```
@0000 5EEC \ 0000-10111-10111-01100   nop clit clit +
@0001 0304 \ Operanden für die clits  3 4
@0002 4D80 \ 0000-10011-01100-00000   nop lit + nop
@0003 FFFF \ Operand für lit          -1
@0004 0802 \ 0000-00010-00000-00010   nop jmp 2
@0005 0000 \                          nop nop nop nop
@0006 0000 \                          nop nop nop nop
...
@0FFF 9000 \ 0001-00100-00000-00000   call 2000
```

Die b16-small-Verilog–Beschreibung (vgl. `b16top.v`) bindet die Hex-Files ein, um damit den b16-Speicher zu initialisieren.

Ein Lauf des b16-small–Programms könnte so aussehen:

```
P    Inst/Slot  op      TOS  NOS
4000 9000/0     01 call 0000 0000
2002 5EEC/0     00 nop  0000 0000
2002 5EEC/1     17 litc 0000 0000
2003 5EEC/2     17 litc 0003 0000
2004 5EEC/3     0C +    0004 0003
2006 4D80/0     00 nop  0007 0000
2006 4D80/1     13 lit  0007 0000
2008 4D80/2     0C +    FFFF 0007
2008 4D80/3     00 nop  0006 0000
200A 0802/0     00 nop  0006 0000
200A 0802/1     02 jmp  0006 0000
2006 4D80/0     00 nop  0006 0000
2006 4D80/1     13 lit  0006 0000
2008 4D80/2     0C +    FFFF 0006
2008 4D80/3     00 nop  0005 0000
200A 0802/0     00 nop  0005 0000
200A 0802/1     02 jmp  0005 0000
...
```

Der Programmzähler P enthält dabei immer die Adresse der nächsten Instruktion, da er durch das Laden des Instruktionsregisters bereits erhöht wird.

Auf der Hardware ausführen

Um dieses Programm mit dem **b16-small** auf dem FPGA–Board laufen zu lassen, öffnen wir in Quartus das b16-small–Projekt–File `b16.qpf`. Das Design kann dann mit Processing>Start Compilation übersetzt werden. Es entsteht ein b16-small aus der Verilog–Beschreibung mit zugehörigem Binärprogramm in den `.hex`–Files. Diese Übersetzung sollte ohne Fehler durchlaufen und das Ergebnis dann auf das DE1–Board geladen werden (Tools>Programmer>Start).

Aber, oh! Enttäuschung! Außer dass ein paar Leuchtdioden angehen und einige aus bleiben[2], kann man nicht beobachten, ob unser Programm läuft oder ob überhaupt irgendwas funktioniert.

Zum Glück lässt sich die auf dem DE1–Board vorhandene 7–Segment–Anzeige zur Darstellung von Werten verwenden. Sie ist im Peripherie–Bereich an der Adresse `$FF00` zu finden. Wir ergänzen unser Programm um

```
macro: ! ( a b -- )  !. drop  end-macro

: boot
    3 #c 4 #c +

    dup $FF00 # !

    BEGIN -1 # +
    AGAIN
;;
```

und geben unser Ergebnis so auf der 7–Segment–Anzeige aus.

[2] Einige Signalleitungen sind an die grünen Leuchtdioden angeschlossen:
`assign LEDG = { SRAM_WE_N, SRAM_CE_N, SRAM_OE_N, SRAM_UB_N, SRAM_LB_N, sel};`, vgl. `b16top.v`

Im File `regmap.asm` werden symbolische Namen für angeschlossene Peripherie vereinbart. Dort hat die 7–Segment–Anzeige den Namen LED7, den wir auch hätten verwenden können, wenn wir mit `include regmap.asm` die Definitionen eingebunden hätten. Der Anschluss der 7–Segment–Anzeige selbst an den Prozessor ist im File `sfr.v` beschrieben. Die Dekodierung der 7–Segment–Elemente wird in den von Altera bereitgestellten Files `SEG7_LUT.v` und `SEG7_LUT_4.v` definiert und ist in Abbildung 3 auszugsweise dargestellt.

Wir übersetzen das Programm und das Design erneut und nach dem Laden in den FPGA ertönt ein Freudenschrei: Die 7–Segment–Anzeige zeigt 0007! Unser erstes b16-small–Programm läuft!

Aber auch das Zählen würden wir ja gerne sehen. Wir ändern unser Programm abermals:

```
include regmap.asm

$2000 org

: wait ( u -- )
    BEGIN
      dup
    WHILE
      1000 # BEGIN -1 # + dup  -UNTIL drop
      -1 # +
    REPEAT drop
;
```

```
: boot
    3 #c 4 #c +

    dup LED7 # !

    BEGIN -1 # +
       dup LED7 # !
       1000 # wait
    AGAIN
;;
```

In der Endlos–Schleife geben wir den Zähler aus, warten aber eine Weile, damit unser langsames Auge auch sehen kann, was da passiert: Countdown. Das Warten wird durch das Wort `wait` erledigt, das einfach CPU–Zyklen verbrät. Da kommt natürlich sofort der Wunsch nach einem richtigen Timer auf. . . Was für die zukünftigen Überlegungen.

Ausblick

Wir haben unser erstes b16-small–Programm geschrieben und auf dem FPGA–Board laufen lassen. Die 7–Segment–Anzeige können wir von Programmen aus ansteuern. Um in die b16–Interna hineinzusehen, wäre es aber auch sinnvoll, die Anzeige vielleicht direkt an das P–Register oder an TOS zu knüpfen. Dazu müssen wir den b16–Verilog–Code anpassen. Dieser Aufgabe werden wir uns im nächsten Teil dieser Artikelserie widmen. Auch wollen wir dann einen Piezo einsetzen, um ein wenig Rabatz–IO zu machen.

Literatur

[1] http://www.colorforth.com/inst.htm Homepage von Charles Moore, Forth–Prozessor c18

[2] http://www.jwdt.com/~paysan/b16.pdf b16 — Ein Forth Prozessor im FPGA, Bernd Paysan, Forth Tagung 2003

[3] http://www.jwdt.com/~paysan/b16.pdf b16-small — Less is More, Bernd Paysan, EuroForth 2004

[4] http://www.forth-ev.de/repos/b16-small b16-small SVN–Repository

[5] http://tortoisesvn.tigris.org/ TortoiseSVN, Windows–Explorer–Integration für Subversion

SEG7_LUT.v

```
 1  module SEG7_LUT(oSEG, iDIG);
 2  input  [3:0] iDIG;
 3  output [6:0] oSEG;
 4  reg    [6:0] oSEG;
 5
 6  always @(iDIG) begin
 7      case(iDIG)
 8      4'h1: oSEG = 7'b1111001;        // ---t----
 9      4'h2: oSEG = 7'b0100100;        // |      |
10      4'h3: oSEG = 7'b0110000;        // lt    rt
11      4'h4: oSEG = 7'b0011001;        // |      |
12      4'h5: oSEG = 7'b0010010;        // ---m----
13      4'h6: oSEG = 7'b0000010;        // |      |
14      4'h7: oSEG = 7'b1111000;        // lb    rb
15      4'h8: oSEG = 7'b0000000;        // |      |
16      4'h9: oSEG = 7'b0011000;        // ---b----
17      4'ha: oSEG = 7'b0001000;
18      4'hb: oSEG = 7'b0000011;
19      4'hc: oSEG = 7'b1000110;
20      4'hd: oSEG = 7'b0100001;
21      4'he: oSEG = 7'b0000110;
22      4'hf: oSEG = 7'b0001110;
23      4'h0: oSEG = 7'b1000000;
24  endcase end endmodule
```

SEG7_LUT_4.f

```
 1  module SEG7_LUT_4 (oSEG0, oSEG1, oSEG2, oSEG3, iDIG);
 2  input  [15:0] iDIG;
 3  output [6:0]  oSEG0, oSEG1, oSEG2, oSEG3;
 4
 5  SEG7_LUT u0 (oSEG0, iDIG[3:0]);
 6  SEG7_LUT u1 (oSEG1, iDIG[7:4]);
 7  SEG7_LUT u2 (oSEG2, iDIG[11:8]);
 8  SEG7_LUT u3 (oSEG3, iDIG[15:12]);
 9
10  endmodule
```

aus b16top.v

```
222    SEG7_LUT_4 u0 (HEX0, HEX1, HEX2, HEX3, LED7);
```

Abbildung 3: Die Dekodierung für die 7–Segment–Anzeige

Lebenszeichen

Berichte aus der FIG Silicon Valley: *Henry Vinerts*

Die Teilnehmer des großen Forth–Tags 2009 in Stanford
Henry Vinerts steht in der letzten Reihe als Dritter von rechts (im SWAP-Shirt :-)

Lieber Fred und die Herren des Direktoriums:

Mit meinem Dank für das VD–Heft 3/2009 bin ich etwas spät dran. Es kam am 31. Oktober. Ich wollte aber erst unseren SVFIG–Forth–Tag abwarten, den ich am Samstag besuchte, so dass ich euch von meinen Eindrücken von dort berichten kann.

SVFIG traf sich im Maschinenbau–Bau an der Stanford University, nachdem es ja am Cogswell College ab dem Forth–Tag 2008 nicht mehr untergebracht werden konnte. Es fällt mir zunehmend schwerer, nach Stanford zu fahren, und so habe ich die SVFIG–Treffen ein ganzes Jahr lang ausgelassen. Natürlich geht mein Zögern, in der San–Francisco–Bay–Area herumzufahren, besonders bei einsetzender Dunkelheit, mit den Lebenszeit–Monaten einher, die immer schneller an mir vorbeirauschen, und dem Straßenverkehr, der nicht langsamer wird.

Das Treffen begann morgens um 8.30 Uhr und dauerte wie angekündigt bis kurz nach 18.00 Uhr, so dass eine beträchtliche Gruppe die Sitzung beim Abendessen in einem nahegelegenen Restaurant fortsetzen konnte. Am Anfang waren es etwa zwei Dutzend Teilnehmer. Um 12.30 Uhr, als Dr. Ting sein traditionelles Barbecue–Essen in einem Picknick–Raum in der Nähe veranstaltete, war die Zahl auf über 40 angewachsen. Der Tag war mit mehr als einem Dutzend Vorträgen und Vorführungen ausgefüllt und endete mit Chuck Moores üblichem *Plausch am Feuer* und einem Gruppenfoto, das bald unter `www.forth.org` zu finden sein wird. Dave Jaffe machte den ganzen Tag über Aufnahmen, die er inzwischen bereits ins Internet gestellt hat: `http://www.kodakgallery.com/gallery/creativeapps/slideShow/Main.jsp?token=177217655805%3A823737578&cm_mmc=site_email-_-new_site_share-_-core-_-View_photos_button`

Wie ich annehme, wird auch eine Zusammenfassung der Vorträge unter dem SVFIG–Link der genannten Web–Page erscheinen. Ich möchte meine Meldung mit einem bisschen Statistik von meinen eigenen Beobachtungen schließen: Von den über 40 Leuten, die über den Tag verstreut anwesend waren, waren etwa 10 Neulinge oder waren mir unbekannt. Aber mir fielen auch mehr als ein Dutzend Teilnehmer auf, die schon regelmäßig zu den Forth–Treffen gekommen waren, bevor ich 1990 als *ältester Forth–Anfänger* in Erscheinung trat. Und der Rest war mir vereinzelt aus den letzten 19 Jahren in Erinnerung. Natürlich, traurig genug, wird die Zahl derer immer größer, die nicht länger kommen werden. Und unter denen waren viele, deren Gesellschaft und deren Wissen mich in allererster Linie dazu bewog, zu den monatlichen Treffen zu kommen.

Nicht nur die Freundschaften am Ort, sondern auch die per Korrespondenz über den *Großen Teich* hinweg waren es, die mir während der ganzen Jahre meiner Beschäftigung mit Forth Freude bereiteten. Ich werde sie vermissen, aber nicht vergessen. Gentlemen, ich danke Ihnen, ich danke euch, für die Zeit, die Sie, die ihr, dem Unterzeichneten, Ihrem/eurem, *Live–Forth*–Reporter aus dem Silicon–Valley entgegengebracht habt. Dies sind sicher nicht die letzten Zeilen, die ich von mir habe hören lassen, aber ich denke, es ist an der Zeit, dass ich etwas *kürzer trete*. Ich werde mich von Zeit zu Zeit auf eurer Homepage umsehen und ich wünsche euch allen Gesundheit, Energie und Enthusiasmus bei der Fortsetzung eurer Unternehmungen mit Forth.

Wie die Holländer sagen, „met hartelijke groeten“, Your Henry Vinerts

Übersetzt von Fred Behringer

Gleitkommazahlen – Floating–Point–Numbers

Michael Kalus

Der Umgang mit Gleitkommazahlen hält eine ganze Reihe unerfreulicher Überraschungen für den Unvorsichtigen bereit (z. B. ist deren Addition nicht assoziativ) und sogar einige für den Vorsichtigen. Man sollte sie in Programmen nicht benutzen, es sei denn, man weiß, was man tut, oder es kümmert einen nicht, dass man völlig falsche Ergebnisse bekommt. Wer gründlicher lernen möchte, was für Probleme die Gleitkomma–Arithmetik in Rechnern bewirkt, könnte mit dem Papier von David Goldberg [1] anfangen.

Hier sind Übungen zusammengestellt, mit denen ihr euch in die ANS–Forth–Syntax des Floating Point hineinfinden könnt. Die Beispiele sind in gforth ausgeführt, das für die Plattformen u. a. Mac OS X, Windows und Linux frei verfügbar ist.

Der floating point stack

Gforth hat einen separaten Gleitkommastack (floating point stack, FPS). Um anzudeuten, dass die Menge der Gleitkommazahlen eine endliche Teilmenge der rationalen Zahlen ist, werden Gleitkommazahlen auf ihrem Stack gerne mit einem `r` bezeichnet (siehe Glossar). In der Dokumentation des gforth wird eine vereinheitlichte Stacknotation benutzt, aus der zunächst gar nicht ersichtlich ist, dass ein seperater FPS vorliegt. Doch es ist leicht, eine Notation für getrennte Stacks daraus zu machen. Setze dafür die Gleitkommazahlen einfach daneben. Z. B.:

```
( n r1 u r2 - r3 )
```

wird zu

```
( n u - ) ( F: r1 r2 - r3 )
```

Inspizieren lässt sich dieser Stack mit `f.s` ganz so, wie wir es schon vom Datenstack her gewohnt sind, der ja mit `.s` dargestellt werden kann.

Übungen

Eingabe einer Gleitkommazahl

Als Erstes geben wir eine Zahl mit Dezimalpunkt ein und versuchen, sie als Gleitkommazahl wieder auszugeben. (←˜ bedeutet, dass die Eingabetaste gedrückt worden ist. Dahinter oder darunter folgt die Ausgabe.)

```
1234.56789 f. ←˜
*the terminal*:22: Floating-point stack
                                  underflow
```

Hm, das gibt schon mal eine Fehlermeldung! Warum? Gforth interpretiert Zahlen mit Punkt zwar als doppelt–genaue Zahl, aber diese wird auf den Datenstack gelegt, der FPS bleibt leer, `f.` greift also ins Leere: `Stack underflow`.

Inspizieren wir die Stacks:

```
1234.56789 ←˜
.s ←˜
<2> 123456789 0 f.s ←˜
<0>
```

Um den Interpreter zu veranlassen, unsere Eingabe als Gleitkommazahl zu sehen, muss unsere Eingabe einer bestimmten Form genügen. Erst eine Zeichenkette der Form:

```
{+ -}<Dezimalziffer>{.}<Dezimalziffer>
       {e E}{+ -}<Dezimalziffer><Dezimalziffer>
```

wird als Gleitkommazahl behandelt, die dann auch auf dem FPS landet. Die folgenden Formate führen zu gleichwertigen Eingaben:

```
1e 1e0 1.e 1.e0 +1e+0 ←˜
f.s ←˜
<5> 1. 1. 1. 1. 1.
```

Auch eine Zahl mit negativem Exponenten ist so möglich:

```
+12.E-4 ←˜
f.s ←˜
<1> 0.0012
```

Wir müssen die Gleitkommazahl also mit einem Exponenten schreiben.

Neben der Zifferneingabe über die Tastatur (input stream) hält Gforth auch die Möglichkeit bereit, eine Zeichenkette (string) in eine Gleitkommazahl umzuwandeln.

```
s" 1234.56789" >float  ( -- f ) (FPS: -- r )
```

Hierbei wird dann eine Gleitkommazahl auf den FPS gelegt, wenn die Konversion der Zeichenkette erfolgreich war, und ein `true` erscheint auf dem Datenstack. `s" 1234xyz56789" >float` hingegen würde ein `false`–Flag ergeben und der FPS bliebe leer.

Ausgabe einer Gleitkommazahl

Das Wort `f.` nimmt eine Gleitkommazahl vom FPS und gibt sie aus. Das ist die einfachste Art:

```
f.s ←˜ <1> 0.0012
f. ←˜ 0.0012
```

Nun probieren wir Gleitkommazahlen auch anders formatiert auszugeben.

```
12345.6789e0 fdup cr fe. cr fs. ←˜
12.3456789000000E3
1.23456789000000E4
```

```
sp@ depth 1- cells dump ←↩

 140DFDC: 00 00 00 99  00 00 00 88 - 00 00 00 77  00 00 00 66  ...........w...f
 140DFEC: 00 00 00 55  00 00 00 44 - 00 00 00 33  00 00 00 22  ...U...D...3..."
 140DFFC: 00 00 00 11              -                            ....
 ok
```

Abbildung 1: Dump des Datenstacks `11 22 33 44 55 66 77 88 99`

Wir sehen, dass `fe.` die technische (engineering) Notation ergibt. Bei der technischen Notation wird der Exponent immer auf das größtmögliche Vielfache von drei justiert. Dei der wissenschaftlichen (scientific) Notation hingegen wird immer auf eine Stelle vor dem Komma justiert.

Doppelt–genaue Zahl in eine Gleitkommazahl umwandeln.

Doch wie können nun Zahlen vom Datenstack als Gleitkommazahlen ausgegeben werden? Nun, man schiebt sie einfach mit `d>f` rüber auf den FPS, und gibt sie von dort aus — jedenfalls im Prinzip. Nun kann `d>f` aber nicht wissen, wie diese Zahl ursprünglich mal auf den Datenstack gekommen ist und welcher Teil davon die Vorkomma– und welcher die Nachkommastellen repräsentiert. Denn Zahlen auf dem Datenstack sind formlos in Forth, sie können ja alles bedeuten, eine Adresse ebenso wie eine ganze Zahl (integer) oder irgendein Token.

Beobachten wir also einmal, was dabei abläuft:

```
: tt  ( d -- )
   cr  .s f.s   d>f   cr .s f.s   cr DPL @ . ;
```

Wir erhalten:

```
1234.56789 tt ←↩
<2> 123456789 0 <0>
<0> <1> 123456789.
5  ok
```

Nach der Eingabe von `1234.56789` lagen also **zwei** Werte auf dem Datenstack und zunächst nichts auf dem FPS. Mit `d>f` wird die doppelt–genaue Zahl genommen und auf den FPS verschoben. Anschließend ist der Datenstack leer, und der FPS enthält **einen** Wert, eine Gleitkommazahl, mit derselben Ziffernfolge wie die doppelt–genaue Zahl zuvor. Und wo ist der Dezimalpunkt geblieben? Dafür schauen wir in der Uservariablen `DPL` nach. `DPL @ .` ergibt `5`. Das bedeutet, es gab bei der letzten Zahlenkonversion 5 Nachkommastellen. Die Position des Dezimalpunktes wird also aufgehoben, aber nicht automatisch gewertet. Das obliegt dem Programmierer selbst.

So weit, so gut. Nur, wie kriegt man die Nachkommastellen wieder in die Gleitkommazahl hinein? Multiplizieren mit `10e-5` würde die richtige Gleitkommazahl `1234.56789` zurückliefern. Die `5` in `DPL` ist somit der Logarithmus zur Basis $b = 10$ unserer gesuchten Zahl a ($x = log_b a$) und man schreibt:

```
1234.56789  d>f  DPL @ s>d d>f fnegate
falog  f*  f. ←↩
1234.56789 ok
```

Interne Repräsentation der Gleitkommazahl

In Forth ist es recht einfach, sich anzusehen, wie eine Gleitkommazahl tatsächlich im RAM abgelegt wird. Man kann die Stacks mittels `DUMP` inspizieren. Da die Stacks in gforth in Richtung kleinerer Adressen wachsen, zeigen die folgenden Sequenzen deren Inhalt:

```
sp@ depth 1- cells dump
fp@ fdepth floats dump
```

(Das kann in Zukunft allerdings durch Stack–Caching unerwartete Resultate liefern.)

Eine Eingabe von:

```
hex 11 22 33 44 55 66 77 88 99 ←↩
ok
```

zeigt uns auf dem Datenstack dann

```
.s ←↩
<9> 11 22 33 44 55 66 77 88 99  ok
```

Und im `dump` des Datenstacks dann die Speicherbelegung aus Abbildung 1.

Hier wird sichtbar, dass jeweils vier Bytes zur Repräsentation der Zahl benutzt werden. Gforth stellt also einfach–genaue Zahlen in 32bit dar[1]. Und die Werte sind absteigend angelegt. Versuchen wir so etwas nun mit Gleikommazahlen.

```
decimal ←↩
ok
1.e 2.e 3.e 4.e 5.e 6.e 7.e 8.e 9.e ←↩
ok
f.s ←↩
<9> 1. 2. 3. 4. 5. 6. 7. 8. 9.  ok
```

In Abbildung 2 auf der nächsten Seite ist nun ersichtlich, dass je Gleitkommazahl immer 8 Bytes, also 64 Bit, benutzt werden.

```
float . ←↩ 8 ok
1 dfloats . ←↩ 8 ok
```

Doch dabei ist nun ganz und gar nicht mehr sichtbar, wie die Gleitkommazahl in diesen 64 Bit untergebracht

[1] Auskunft darüber erhält man so: `cell . ←↩ 4 ok`. Oder: `1 cells . ←↩ 4 ok`

```
fp@ fdepth floats dump
 1412FB8: 40 22 00 00  00 00 00 00 - 40 20 00 00  00 00 00 00  @"......@ ......
 1412FC8: 40 1C 00 00  00 00 00 00 - 40 18 00 00  00 00 00 00  @.......@.......
 1412FD8: 40 14 00 00  00 00 00 00 - 40 10 00 00  00 00 00 00  @.......@.......
 1412FE8: 40 08 00 00  00 00 00 00 - 40 00 00 00  00 00 00 00  @.......@.......
 1412FF8: 3F F0 00 00  00 00 00 00 -                            ?.......
 ok
```

Abbildung 2: Darstellung der Gleitkommazahlen im Floating–Point–Stack

worden ist. Offensichtlich wird da etwas Codiertes hinterlegt [2]. Doch auch dabei ist es mit Forth recht einfach möglich, diese IEEE–754–Codierung nachzuvollziehen. Schauen wir uns das Format einmal bitweise an.

Dazu legen wir eine Zahl auf den FPS und dumpen den Stack.

```
decimal ↩ ok
-118.625e0 ↩ ok
f.s ↩ <1> -118.625  ok
fp@ fdepth floats dump ↩
 1412FF8: C0 5D A8 00  00 00 00 00 -
```

Binär gelesen ist das:

```
2 base ! ↩
$C0 . ↩ 11000000  ok
$5C . ↩ 01011100  ok
$A8 . ↩ 10101000  ok
...
```

Dort liegt also die Bitfolge:

```
11000000 01011100 10101000 00000000
00000000 00000000 00000000 00000000
```

Zum Vergleich diene folgende Sequenz:

```
2 base ! &118.625 drop . ↩
11100111101100001  ok
```

Offensichtlich ist das **nicht** das Gleiche.

Im IEEE–754–Gleitkomma–Format gibt es drei Felder, für jede Eigenschaft der Zahl eines: Vorzeichen (sign), Exponent (exp) und dann die Ziffernfolge des binären Bruchteils hinter dem normierten binären Ausdruck (Fraction).

Im obigen Beispiel also folgende 64 Bit:

sign, 1 Bit = 1 = negative Zahl
Exponent, 11 Bit = `100 0000 0101`
Fraction, 52 bit = `1100 1010 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000`

Normalisierte Darstellung, biased

Dazu wird die Zahl binär umgewandelt, nun ohne Vorzeichen, also ohne die Zweierkomplement–Schreibweise zu nehmen. Wir erhalten:

```
&118 . ↩
1110110  ok
```

Den Wert hinter dem Komma erhalten wir so:

0.625 × 2 = 1.25 wovon die Eins hinter das Komma kommt.
0.25 × 2 = 0.50 wovon die Null hinter das Komma kommt.
0.5 × 2 = 1.00 wovon wieder die Eins hinter das Komma kommt.

Außerdem sind wir fertig, weil es nichts weiter umzuwandeln gibt.

Unsere Zahl lautet nun: `1110110.101`

Als Nächstes wird der Punkt bis auf die erste Stelle nach links verschoben: `1110110.101 = 1.110110101` × `26`. Dieses ist nun die normalisierte Form der Gleitkommazahl. Die erste binäre Eins wird einfach fallen gelassen, da sie immer gegeben ist. Die Fraction (Bruch) steht nun rechts des Punktes und wird solange mit Nullen aufgefüllt, bis alle Bits der Darstellung aufgefüllt sind.

Der Exponent ist 6, aber auch er muss noch binaär geschrieben und mit einem Bias verrechnet werden, damit der kleinstmögliche Exponent die 0 ist. Im 64–Bit IEEE–754–Format ist der Bias 1023, der Exponent wird daher als $1023 + 6 = 1029$ aufgeschrieben. Binär also: `%10000000101`.

Knobelei

- Was sind die kleinsten und größten Zahlen, die auf diese Weise dargestellt werden können? (Also welche Zahlen liegen am nächsten an der Null, welche am weitesten entfernt?)
- Funktoniert die Umwandlung auch mit anderer Basis als 10?

```
decimal 2 base !
101010.10
.s <10> 10101010 0
decimal DPL @
.s <3> 170 0 2  ok
```

`DPL` enthält auch hier die 2 Nachkommastellen. Man sieht, dass `DPL` lediglich angibt, wieviele Zeichen im Eingabestring hinter dem Punkt noch folgten. Es ist also kein *Dezimalpunkt*, sondern einfach nur ein Marker, der noch weiterer Interpretation bedarf.

Glossar

(Die Regeln, die der Textinterpreter benutzt, um Gleitkommazahlen zu erkennen, stehen im Abschnitt 5.13.2 [Number Conversion], Seite 97 des Handbuches.)

Floating–Point–Ausgabe

`f.` `r -` float-ext “f-dot”
Stelle (die Gleitkommazahl) r ohne Exponent dar, gefolgt von einem Leerzeichen.

`fe.` `r -` float-ext “f-e-dot”
Stelle r in der technischen Notation dar (mit dem Exponenten als ein Vielfaches von drei), gefolgt von einem Leerzeichen.

`fs.` `r -` float-ext “f-s-dot”
Stelle r in der wissenschaftlichen Notation dar (mit dem Exponenten), gefolgt von einem Leerzeichen.

`f.rdp` `rf +nr +nd +np -` gforth “f.rdp”
Die Ausgabe der Gleitkommazahl `rf` wird formatiert. Die ganze Breite der Ausgabe beträgt `+nr` Zeichen. Das Komma ist fixiert, die Anzahl der Stellen hinter dem Komma beträgt `+nd` und das Minimum an signifikanten Stellen ist `+np`. `Set-precision` hat keinen Effekt auf `f.rdp`.

Solch eine Notation mit fester Kommastelle wird benutzt, wenn die Anzahl der signifikanten Stellen vor dem Komma mindestens np beträgt und wenn die Stellen vor dem Komma hineinpassen.

Dabei wird auf die exponentielle Ausgabe umgeschaltet, wenn die Stellen nicht hineinpassen, oder es werden Sterne ausgegeben, wenn auch das nicht passt. Wir empfehlen ein $nr \geq +5$, um zu vermeiden, dass eine Zahl überhaupt nicht passt. Und wir empfehlen ein $nr \geq pn + 5$, um zu vermeiden, dass auf die exponentielle Schreibweise umgeschaltet wird, weil der Gleitkommazahl zu wenig signifikante Stellen bereitgestellt wurden, bietet doch die exponentielle Schreibweise weniger signifikante Stellen. Und wir empfehlen $nr \geq nd + 2$, falls Sie Gleitkommazahlen für mehrere Zahlen brauchen. Schließlich sollte $np > nr$, falls die Ausgabe nur in exponentieller Schreibweise erfolgen soll.

Hier ist zu sehen, wie die Zahl $1234.5678e23$ von den verschiedenen Formaten ausgegeben wird:

```
f. 123456779999999000000000000.
fe. 123.456779999999E24
fs. 1.23456779999999E26
```

Austausch zwischen Datenstack und Floating–Point–Stack

`d>f`	d – r	float “d-to-f”
`f>d`	r – d	float “f-to-d”

Data und Code ansehen

Die folgenden Worte inspizieren den Stack non-destruktiv:

`.s`	–	tools “dot-s”	Datenstack ansehen.
`f.s`	–	gforth “f-dot-s”	Gleitkommastack ansehen.
`depth`	– +n	core “depth”	Tiefe des Datenstacks.
`fdepth`	– +n	float “f-depth”	Tiefe des FPS.
`clearstack`	... –	gforth “clear-stack”	Löscht den Datenstack.

Inspizieren von Speicherbereichen

`?`	a-addr –	tools “question”
`dump`	addr u –	tools “dump”
`see`	«spaces>name"–	tools “see”
`xt-see xt`	–	gforth “xt-see”
`simple-see "name"`	–	gforth “simple-see”
`simple-see-range`	addr1 addr2 –	gforth “simple-see-range”

Floating–Point–Stack–Operatoren

`floating-stack`	– n	environment "floating-stack"
	n ist nicht-null, was bedeutet, dass Gforth einen separaten Floating–Point–Stack der Tiefe n hat.	
`fdrop`	r –	float "f-drop"
`fnip`	r1 r2 – r2	gforth "f-nip"
`fdup`	r – r r	float "f-dupe"
`fover`	r1 r2 – r1 r2 r1	float "f-over"
`ftuck`	r1 r2 – r2 r1 r2	gforth "f-tuck"
`fswap`	r1 r2 – r2 r1	float "f-swap"
`fpick`	u – r	gforth "fpick"
	Eigentlich ist der Stackeffekt so: r0 . . . ru u – r0 ... ru r0	
`frot`	r1 r2 r3 – r2 r3 r1	float "f-rote"

Floating–Point–Operatoren

`f+`	r1 r2 – r3	float "f-plus"
`f-`	r1 r2 – r3	float "f-minus"
`f*`	r1 r2 – r3	float "f-star"
`f/`	r1 r2 – r3	float "f-slash"
`fnegate`	r1 – r2	float "f-negate"
`fabs`	r1 – r2	float-ext "f-abs"
`fmax`	r1 r2 – r3	float "f-max"
`fmin`	r1 r2 – r3	float "f-min"
`floor`	r1 – r2	float "floor"
	Runde auf den nächsten kleineren Integer-Wert ab, d.h. runde gegen negativ Unendlich.	
`fround`	r1 – r2	gforth "f-round"
	Runde auf den nächsten Integer-Wert.	
`f**`	r1 r2 – r3	float-ext "f-star-star"
	r3 ist r1 hoch r2	
`fsqrt`	r1 – r2	float-ext "f-square-root"
`fexp`	r1 – r2	float-ext "f-e-x-p"
`fexpm1`	r1 – r2	float-ext "f-e-x-p-m-one"
	r2 = e**r1 -1	
`fln`	r1 – r2	float-ext "f-l-n"
`flnp1`	r1 – r2	float-ext "f-l-n-p-one"
	r2 =ln(r1 +1)	
`flog`	r1 – r2	float-ext "f-log"
	Der dezimale Logarithmus.	
`falog`	r1 – r2	float-ext "f-a-log"
	r2 = 10**r1	
`f2*`	r1 – r2	gforth "f2*"
	Multipliziere r1 mit 2.0e0	
`f2/`	r1 – r2	gforth "f2/"
	Multipliziere r1 mit 0.5e0	
`1/f`	r1 – r2	gforth "1/f "
	Teile 1.0e0 durch r1.	
`precision`	– u	float-ext "precision"
	u ist die Anzahl der signifikanten Ziffern, die nun von `F.` `FE.` und `FS.` benutzt werden.	
`set-precision`	u –	float-ext "set-precision"
	Setzt die Anzahl der signifikanten Ziffern, die nun von `F.` `FE.` und `FS.` benutzt werden, zu `u`.	

Winkelfunktionen als Floating–Point–Operationen

Die Winkel werden immer in Radianten angegeben. Ein voller Kreis hat 2π Radianten.

`fsin`	r1 – r2	float-ext “f-sine”
`fcos`	r1 – r2	float-ext “f-cos”
`fsincos`	r1 – r2 r3	float-ext “f-sine-cos”
	r2 =sin(r1), r3 =cos(r1)	
`ftan`	r1 – r2	float-ext “f-tan”
`fasin`	r1 – r2	float-ext “f-a-sine”
`facos`	r1 – r2	float-ext “f-a-cos”
`fatan`	r1 – r2	float-ext “f-a-tan”
`fatan2`	r1 r2 – r3	float-ext “f-a-tan-two”
	r1/r2 =tan(r3). ANS–Forth fordert es zwar nicht, wollte es aber vermutlich, so dass dies die Umkehrung des fsincos ist. In gforth ist es so.	
`fsinh`	r1 – r2	float-ext “f-cinch”
`fcosh`	r1 – r2	float-ext “f-cosh”
`ftanh`	r1 – r2	float-ext “f-tan-h”
`fasinh`	r1 – r2	float-ext “f-a-cinch”
`facosh`	r1 – r2	float-ext “f-a-cosh”
`fatanh`	r1 – r2	float-ext “f-a-tan-h”
`pi`	– r	gforth “pi”
	Die Konstante r ist der Wert Pi; das Verhältnis des Kreisumfangs zum Durchmesser.	

Ein spezielles Problem der Floating–Point–Arithmetik ist, dass ein Test auf Gleichheit oft gerade dann fehlschlägt, wenn man eigentlich erwartet, dass die Werte gleich sein sollten. Daher versucht man, sich mit einer ungefähren Gleichheit zu behelfen (aber man sollte dabei wissen, was man tut). Beachte auch, dass die IEEE NaNs vielleicht anders vergleicht, als du erwartet hattest.

Folgende vergleichenden Worte gibt es:

`f~rel`	r1 r2 r3 – flag	gforth “f rel”
	Ungefähre Gleichheit mit relativem Fehler: $\|r1 - r2\| < r3 \cdot \|r1 + r2\|$.	
`f~abs`	r1 r2 r3 – flag	gforth “f abs”
	Ungefähre Gleichheit mit absolutem Fehler: $\|r1 - r2\| < r3$.	
`f~`	r1 r2 r3 – flag	float-ext “f-proximate”
	ANS–Forth–Sammlung, um $r1$ und $r2$ auf Gleichheit zu testen: Für $r3 > 0$ nimm `f abs` ; für $r3 = 0$ nimm bitweisen Vergleich; für $r3 < 0$ nimm `fnegate f rel` .	
`f=`	r1 r2 – f	gforth “f-equals”
`f<>`	r1 r2 – f	gforth “f-not-equals”
`f<`	r1 r2 – f	float “f-less-than”
`f<=`	r1 r2 – f	gforth “f-less-or-equal”
`f>`	r1 r2 – f	gforth “f-greater-than”
`f>=`	r1 r2 – f	gforth “f-greater-or-equal”
`f0<`	r – f	float “f-zero-less-than”
`f0<=`	r – f	gforth “f-zero-less-or-equal”
`f0<>`	r – f	gforth “f-zero-not-equals”
`f0=`	r – f	float “f-zero-equals”
`f0>`	r – f	gforth “f-zero-greater-than”
`f0>=`	r – f	gforth “f-zero-greater-or-equal”

Literatur

[1] David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic" ACM Computing Surveys 23(1):5–48, March 1991. http://www.validgh.com/goldberg/paper.ps

[2] http://en.wikipedia.org/wiki/IEEE-754

32-Bit-Umordnung für FFT bei nur 16-Bit-Breite des Forth-Assemblers

Fred Behringer

Der vorliegende Artikel ist eine überarbeitete und leicht erweiterte deutsche Fassung meines Leserbriefes aus Heft 114 (2001) der Zeitschrift Forthwrite unserer Forth-Freunde aus Großbritannien (FIG-UK).

In der Maschinensprache des INMOS-Transputers T800 [IT] gibt es (neben weiteren drei Bitmanipulationsbefehlen) den Befehl bitrevnbits (reverse bottom n bits). Bei den INTEL-Prozessoren sucht man einen solchen Befehl vergeblich. Es gibt Aufgaben, bei denen man ihn sehr gut gebrauchen könnte. Die Schnelle Fourier-Transformation (FFT - fast fourier transformation) gehört dazu [FT].

Selbstverständlich kann man die Bitumordnung auch in High-Level-Forth erledigen (Schleife und Schiebungen nach links und rechts) [siehe Forthwrite 113]. Aber FFT verlangt nicht ohne Grund nach Maschinen-Code.

Bitbearbeitungsbefehle werden bei INTEL recht stiefmütterlich behandelt [TP]. Das bemängelt auch Julian V. Noble in seinem 2001 in der Forthwrite veröffentlichten dreiteiligen Artikel *A Call to Assembly 1/3* [JN], in welchem er für die Verwendung von mehr Assembler in Forth-Programmen plädiert. (Eine von mir angefertigte deutsche Übersetzung erschien in der Vierten Dimension im Jahre 2002 unter dem Titel *Ein Assembl(i)eraufruf* [DN].)

Ich selbst habe immer wieder betont, dass es mir große Freude bereitet, Forth zum Assemblieren *missbrauchen* zu können, und ich stimme mit Julian Noble (der ja leider vor nicht allzu langer Zeit von einer höheren Instanz abberufen wurde) darüber überein, dass es nicht ungeschickt wäre, Forth den Ein- und Umsteigern über den ungemein eleganten und leicht den eigenen Wünschen anzupassenden Assembler näherzubringen.

Eine Bemerkung in dieser Richtung beläuft sich auf Folgendes: ZF und Turbo-Forth, meine Lieblingsprogramme, sind 16-Bit-Systeme. Mit so gut wie gar keinem Zusatzaufwand und in fast derselben Computerzeit (ich spreche vom Pentium - eigentlich vom AMD-K6/2) kann die Bitumordnung (des 16-Bit-Programms von J.V. Noble) sofort auf 32 Bits ausgedehnt werden. Ohne 32-Bit-Breite im Assembler! (Bei Noble heißt das Umordnungswort stib, also bits rückwärts. Ich werde (siehe Listing) DSTIB (stib für Double-Werte) sagen. Die CPU-Takte, die ich angeben werde, sind dem Buch von T.E. Podschun entnommen [TP].

Zum Ausprobieren ist Folgendes zu bemerken: Seit einer kaum noch rekonstruierbaren Anzahl von Jahren arbeite ich mit einem Programm, mit dessen Hilfe ich Zeichen vom Bildschirm einsammeln und an die momentane DOS-Eingabezeile legen kann - auch aus Turbo-Forth oder ZF heraus. Das spart sehr viel Tipparbeit und hat auch hier das halbautomatische Ausprobieren sehr erleichtert. (Wenn ich mal zwischendurch nicht mehr weiß, mit was ich mich beschäftigen soll, werde ich versuchen, ein solches Programm in Forth zu schreiben, damit ich es den jeweiligen Gegebenheiten besser anpassen kann.) In Turbo-Forth gibt es das Wort B. zur binären Darstellung eines einfachgenauen Stackwertes. Man braucht sich nicht lange mit Forth beschäftigt zu haben, um herauszubekommen, wie man sich eine Entsprechung DB. für doppeltgenaue Werte selbst definiert. Ich mache im Listing (siehe unten) einen Vorschlag. Wenn man aber beim Ausprobieren des Wortes DSTIB oder DSTIB-2 (siehe Listing), vielleicht aus Bequemlichkeitsgründen, nur mit B. arbeiten möchte, beachte man, dass bei B. führende Nullen unterdrückt werden. Man glaubt nicht, wie oft das bei Ansicht zweier Binärgruppen zu Fehlinterpretationen führt! Außerdem besteht ja auch auf dem Bildschirm die unselige Umordnung high/low: Würde man (in einem 16-Bit-Forth-System) einen 32-Bit-Wert so auf den Stack legen, wie er bei Intel in einem 32-Bit-Register steht und in den Arbeitsspeicher gelegt wird (Little-Endian), dann kämen die beiden 16-Bit-Hälften verkehrt herum an.

Listing

```
 1  HEX
 2
 3  : OP: 66 C, ;           \ Praefix zum Umschalten auf 32-Bit-Register
 4
 5  \ Lege die unteren n Bits des doppeltgenauen Wertes d in umgekehrter
 6  \ Reihenfolge auf den Datenstack und setze die restlichen Bits auf 0
 7
 8  CODE DSTIB ( n d -- d' )
 9    OP: BX POP            \   1 CPU-Takt
10    OP: C1 C, CB C, 10 C, \   1 CPU-Takt  10 # EBX ROR ; hi/lo > lo/hi
11        CX POP            \   1 CPU-Takt
12    OP: DX DX XOR         \   1 CPU-Takt
```

```
13    HERE
14      OP: BX SHR          \   1 CPU-Takt
15      OP: DX RCL          \   1 CPU-Takt
16          CX DEC          \   1 CPU-Takt
17    JNE                   \   1 CPU-Takt
18    OP: C1 C, CA C, 10 C, \   1 CPU-Takt  10 # EDX ROR ; lo/hi > hi/lo
19    OP: DX PUSH           \   1 CPU-Takt
20    NEXT END-CODE         \  70 CPU-Takte bei Umordnung von 16 Bits
21                          \ 134 CPU-Takte bei Umordnung von 32 Bits
22                          \ (4*n)+6 Takte bei Umordnung von  n Bits
23                          \ Code-Laenge = 40 Bytes
24
25  \ Was folgt, ist die von Julian Noble stammende
26  \ 16 Bit-Version (Forthwrite # 113)
27
28  \ Lege die unteren n Bits des einfachgenauen Wertes n2 in umgekehrter
29  \ Reihenfolge auf den Datenstack und setze die restlichen Bits auf 0
30
31  CODE STIB ( n1 n2 -- n2' )
32    BX POP                \   1 CPU-Takt
33    CX POP                \   1 CPU-Takt
34    DX DX XOR             \   1 CPU-Takt
35    HERE
36      BX SHR              \   1 CPU-Takt
37      DX RCL              \   1 CPU-Takt
38    LOOP                  \   6 CPU-Takte
39    DX PUSH               \   1 CPU-Takt
40    NEXT END-CODE         \ 132 CPU-Takte bei Umordnung von 16 Bits
41                          \ (8*n)+4 Takte bei Umordnung von  n Bits
42                          \ Umordnung von hoechstens 16 Bits
43                          \ Code-Laenge = 25 Bytes
44
45  \ Das STIB bei Julian Noble verbraucht fast zweimal so viel CPU-Zeit
46  \ wie DSTIB und deckt nur Umordnungen von bis zu 16 Bit Laenge ab.
47
48  \ Wenn es auf Speicher nicht ankommt, kann die benoetigte Zeit fuer DSTIB
49  \ sogar noch weiter reduziert werden (vergleiche das folgende DSTIB-2).
50
51  \ Lege die unteren n Bits des doppeltgenauen Wertes d in umgekehrter
52  \ Reihenfolge auf den Datenstack und setze die restlichen Bits auf 0
53
54  CODE DSTIB-2 ( n d -- d' )
55    OP: BX POP            \   1 CPU-Takt   Hole d
56    OP: C1 C, CB C, 10 C, \   1 CPU-Takt   10 # EBX ROR ; lo/hi > hi/lo
57        CX POP            \   1 CPU-Takt   Hole n
58        20 # AX MOV       \   1 CPU-Takt   AX = nmax
59        CX AX SUB         \   1 CPU-Takt   AX = nmax - n
60        AX DI MOV         \   1 CPU-Takt   Aequivalent
61        DI SHL            \   1 CPU-Takt   zu AX*6,
62        DI AX ADD         \   1 CPU-Takt   aber mit
63        AX SHL            \   1 CPU-Takt   weniger Takten.
64    OP: DX DX XOR         \   1 CPU-Takt   DX = 0
65    HERE 7 + # DI MOV     \   1 CPU-Takt   Plus 7 Bytes fuer MOV, ADD und JMP.
66        AX DI ADD         \   1 CPU-Takt   n Kopien von EBX SHR EDX RCL
67        DI JMP            \   2 CPU-Takte  nach dem Sprung noch auszufuehren.
68    HERE                  \                Zur Bearbeitung von XXX
69    0C0 ALLOT             \ 2*n CPU-Takte  32-mal EBX SHR EDX RCL
70    OP: C1 C, CA C, 10 C, \   1 CPU-Takt   10 # EDX ROR ; lo/hi > hi/lo
71    OP: DX PUSH           \   1 CPU-Takt
72    NEXT END-CODE         \  48 CPU-Takte bei Umordnung von 16 Bits
```

```
73                            \  80 CPU-Takte bei Umordnung von 32 Bits
74                            \ (2*n)+16 Takte bei Umordnung von n Bits
75                            \ Code-Laenge = 245 Bytes
76
77   : XXX ( -- )
78     DUP 0C0 + SWAP
79     DO  66 I     C! 0D1 I 1 + C! 0EB I 2 + C!  \ EBX SHR
80         66 I 3 + C! 0D1 I 4 + C! 0D2 I 5 + C!  \ EDX RCL
81     6 +LOOP ;
82
83   XXX                      \ Fuelle DSTIB-2 mit 32 Kopien von EBX SHR EDX RCL
84   FORGET XXX               \ Beseitige Hilfswort XXX, das nun ueberfluessig ist.
85
86   \ Man beachte, dass es verheerende Wirkung hat, wenn man n uebermaessig
87   \ gross waehlt! Man versuche es spasseshalber mal mit DECIMAL 33 4 DSTIB-2.
88   \ (Der Aufhaengeschleife entkommt man mit CTRL-ALT-PAUSE!) Andererseits
89   \ wuerden ein paar weitere CPU-Takte genuegen, das Problem gegen derartige
90   \ nicht sinnvolle Eingaben abzusichern. Bei DSTIB oder STIB gibt es das
91   \ Problem nicht. Man müsste dann aber bei uebermaessig grossem n einen Moment
92   \ ueberlegen, um das Ergebnis richtig zu interpretieren.
93
94   \ Es ist wirklich erstaunlich, wie weit man zur Erreichung des gesteckten
95   \ Zieles gehen kann, obwohl weder ZF noch Turbo-Forth einen 32-Bit-Assembler
96   \ haben. Man stelle sich nur einen einzigen der hier verwendeten
97   \ "Quick-and-dirty-Tricks" in irgendeiner der zahlreichen
98   \ "Mainstream-Sprachen" ausserhalb von Forth vor!
99
100  \ Julian Nobles STIB benoetigt 25 Bytes. DSTIB (von mir) benoetigt 40 Bytes.
101  \ DSTIB-2 (ebenfalls von mir) benoetigt 245 Bytes. Das ist ein interessantes
102  \ Trade-off-Beispiel von Zeit gegen Speicher. Aber was hat man heutzutage
103  \ nicht alles an Arbeitsspeicher in der Maschine? Einige Hundert Megabyte bis
104  \ zu 4 Gigabyte! Was kuemmert einen da schon der benoetigte Speicher -
105  \ solange das (16-Bit-)Forth-System ueberhaupt dermassen viel Arbeitsspeicher
106  \ verkraften kann. Kann es das? Jawoll, Turbo-Forth, ZF und andere
107  \ 16-Bit-Systeme koennen das sehr wohl! Bereits im Real-Modus unter DOS.
108  \ Siehe [VB]. (Ganz stimmt das nicht. Wo liegt der Haken? Aber das ist eine
109  \ andere Geschichte.;-)
110
111  \ Zur Vereinfachung des Ausprobierens:
112  \ Binaere Anzeige eines 32-Bit-Wertes in einem 16-Bit-System
113  \ Fuehrende Nullen werden nicht angezeigt!
114
115  : DB. ( d -- )
116       BASE @        \ Zahlenbasis aufbewahren
117       2 BASE !      \ Auf binaer schalten
118       ROT ROT       \ Bisherige Basis nach vorn
119       UD.           \ 32-Bit-Absolutzahl binaer
120       BASE ! ;      \ Basis wiederherstellen
121
```

Literatur

[DN] Noble, Julian V.: Ein Assembl(i)eraufruf. Vierte Dimension, Heft 2/2002 (Übersetzung Fred Behringer).
[FB] Behringer, Fred: Leserbrief in Forthwrite 114. FIG-UK, November 2001.
[FT] Bronstein et al.: Taschenbuch der Mathematik. Verlag Harry Deutsch 1993, S.786.
[IT] INMOS Limited: Transputer Instruction Set, a Compiler Writer's Guide. Prentice Hall, 1988.
[JN] Noble, Julian V.: Forthwrite 113-115. FIG-UK, 2001-2002.
[TP] Podschun, Trutz Eyke: Die Assembler-Referenz. Addison-Wesley, 2002.
[VB] Behringer, Fred: Real-Mode-32-Bit-Erweiterung fuer Turbo-Forth. Vierte Dimension, Heft 2/1998.

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) – 79 85 57
`bernd.paysan@gmx.de`
Treffen: Jeden 4. Mittwoch im Monat um 19:00, im Chilli Asia Dachauer Str. 151, 80335 München.

Hamburg Küstenforth
Klaus Schleisiek
Tel.: (0 40) – 37 50 08 03 (g)
`kschleisiek@send.de`
Treffen 1 Mal im Quartal
Ort und Zeit nach Vereinbarung
(bitte erfragen)

Mainz Rolf Lauer möchte im Raum Frankfurt, Mainz, Bad Kreuznach eine lokale Gruppe einrichten.
Mail an rowila@t-online.de

Gruppengründungen, Kontakte

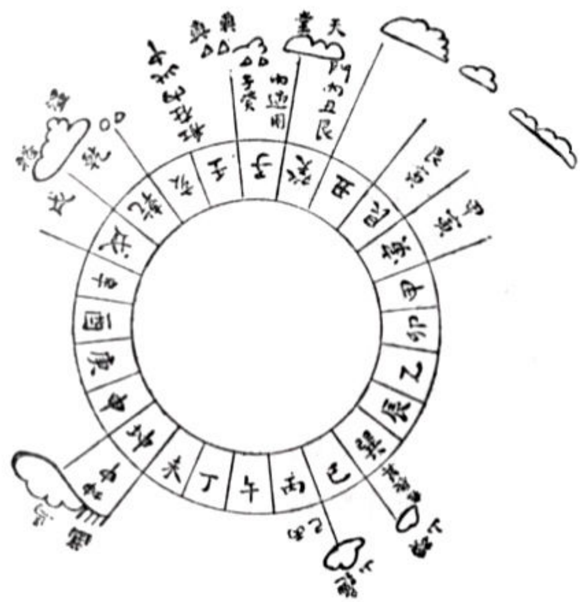
Hier könnte Ihre Adresse oder Ihre Rufnummer stehen — wenn Sie eine Forthgruppe gründen wollen.

µP–Controller Verleih

Carsten Strotmann
`microcontrollerverleih@forth-ev.de`
`mcv@forth-ev.de`

Spezielle Fachgebiete

FORTHchips (FRP 1600, RTX, Novix)	**Klaus Schleisiek–Kern** Tel.: **(0 40) – 37 50 08 03** (g)
KI, Object Oriented Forth, Sicherheitskritische Systeme	**Ulrich Hoffmann** Tel.: **(0 43 51) – 71 22 17** (p) Fax: **– 71 22 16**
Forth–Vertrieb volksFORTH ultraFORTH RTX / FG / Super8 KK–FORTH	Ingenieurbüro **Klaus Kohl–Schöpe** Tel.: **(0 70 44)–90 87 89** (p)



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.

Einladung zur

Forth–Tagung 2010

vom 26. bis 28. März 2010

im Strand–Hotel Hübner

Seestraße 12, 18119 Rostock-Warnemünde

http://www.hotel-huebner.de

Geplantes Programm

Donnerstag, 25.03.2010

nachmittags	Treffen der Frühankommer
	Forth–200x–Standard–Treffen

Freitag, 26.03.2010

nachmittags	Beginn der Forth–Tagung
	Vorträge und Workshops

Samstag, 27.03.2010

vormittags	Vorträge und Workshops
nachmittags	Exkursion

Sonntag, 28.03.2010

09:00 Uhr	Mitgliederversammlung
nachmittags	Ende der Tagung

Anreise siehe: http://www.hotel-huebner.de/kontakt/anfahrt.html

Quellen: www.hotel-huebner.de