Vierte Dimension

3+4/2015

Das Forth–Magazin

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

In dieser Ausgabe:

Dienstleistungen und Produkte fördernder Mitglieder des Vereins

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`

Impressum

Name der Zeitschrift

Vierte Dimension

Herausgeberin

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

Redaktion & Layout

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

Anzeigenverwaltung

Büro der Herausgeberin

Redaktionsschluss

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

Erscheinungsweise

1 Ausgabe / Quartal

Einzelpreis

4,00€ + Porto u. Verpackung

Liebe Leser,

Wer suchet, der findet. Auf einmal kommen von allen Seiten Beiträge für unser Magazin. Wunderbar. Ganz herzlichen Dank an alle Autoren und die Übersetzer, schön, dass ihr so schnell so Tolles beigetragen habt. Und auch den fleißigen Helfern im Hintergrund möchte ich danken, Fred, Bernd, Ulli und Ewald - super!

Und nun, nachdem dieses Doppelheft erstaunlich flott fertig geworden ist, kann ich mich erst mal entspannt zurücklehnen, und gönne mir einen Moore-Abend - es läuft eine alte Platte von Gary Moore, “Still got the blues“, das ist Romantic Rock, und dazu ein Glas Moore's Diamond, das ist ein Wein aus der Gegend von Cape Cod, Massachusetts, in den Staaten von Amerika, der mit der foxiness im Geschmack. Und werkel etwas mit Forth.

Forth von Charles H. Moore ist das Motiv für dieses Blatt. Auch aus den USA. Stimmt, es ist eine kleine Gemeinde, der unseren hier ähnlich. Beim Forth Day 2015 Roll Call auf You Tube könnt ihr sie sehen. Um so erstaunlicher, dass nun in China AmForth benutzt wurde, sogar preisgekrönt. Man muss dazu sagen, dass die Jungs dort es geschickt angestellt haben, da drum herum eine gut funktionierende Programmierumgebung zu erstellen. Worüber ich ja gerne noch mehr erfahren hätte - aber lest selbst. Und von einem Englishman in, nein nicht New York, sondern Honkong, kommt nun der erste stand-alone Microcomputer, welcher Forth als seine native Programmiersprache verwendet. Ich bin begeistert. Und in Bath, England, auf der EuroForth, zeigte sich, dass viele engagierte Menschen Forth weiter entwickeln, hin zu einem Werkzeug, das mehr als nur eine Person bedienen kann. Vor allem gilt es, sich auf den modernen MCUs zu behaupten. Da ist schon sehr deutlich geworden, dass über einen minimalen Forthkern hinaus, der definiert sein will, die reichhaltige Peripherie, die diese Maschinen auf dem Chip haben, bedient werden muss. Da sind gute Treiber und Module wichtiger als ein üppiges Forthsystem selbst. Bisher ist noch offen, in welcher Notation diese Module bedient werden sollen, noch wird sehr maschinennah und individuell damit umgegangen - was wiederum die Puristen freut. Dass solche Peripherie nicht nur Ports sind, sondern auch so etwas wie Flash, Timer, FRAM, PWM, SPI, analoge Eingänge mit allerlei Sensoren, Bluetooth, Displays und Gleitkoma-Einheiten, und dass der Kontrollfluss dort besonders wichtig ist, leuchtet ein. Die Liste solcher Themen ist lang - da wird uns der Stoff nicht ausgehen, da bin ich mir sicher. Amforth und seine Verbreitung hat dazu schon viel beigetragen. Und mecrisp holt gerade tüchtig auf, auf das kommende Sonderheft dazu bin ich schon sehr gespannt.

Und wo wird die Forthtagung 2016 sein? Angedacht ist es ja, beim Linuxtag in Augsburg dabei zu sein. Aber wie das so ist, je größer der Kreis der Beteiligten, um so schwieriger wird es, einen Termin und eine passende Unterkunft zu finden. Derzeit verhandelt das Organisationsteam noch. Also harren wir der Dinge, die da kommen. Und hoffen, dass das Heft 1/2016 sich rasch füllt, dann auch früh im kommenden Jahr erscheinen kann und die Einladung zur Forthtagung 2016 enthält.

Fröhliche Weihnachten, Euer Michael

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
`http://fossil.forth-ev.de/vd-2015-0304`

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

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

Was ist denn ein Swap–Frosch?

Yu-Gi-Oh! (jap. „König der Spiele“) ist eine erstmals 1996 veröffentlichte Manga-Serie des japanischen Zeichners Kazuki Takahashi, die auch als Anime umgesetzt wurde und zu der etliche Merchandising-Produkte, darunter ein sehr erfolgreiches Sammelkartenspiel vertrieben werden.

Die Geschichte handelt von dem sechzehnjährigen Schüler Yugi, der mit dem Zusammenbau eines ägyptischen Puzzles den Geist eines Pharaos freisetzt. Gemeinsam mit diesem muss er in verschiedenen Kämpfen die Welt vor dem Untergang bewahren. Viele Freunde stehen ihm dabei unterstützend zur Seite.

Die Geschichte wurde auch in ein Sammelkartenspiel umgesetzt. Heute erfreut es sich weltweit großer Beliebtheit, so dass in vielen Ländern, darunter Japan, die USA und Deutschland nationale Turniere ausgetragen werden, aber auch internationale Wettkämpfe.

Die Spieler, Duellanten genannt, besitzen jeweils ein Deck von Karten und wechseln sich während eines Duells mit ihren Zügen ab, die ihrerseits in mehrere Phasen unterteilt sind und für die unterschiedliche Regeln gelten - alles sehr kompliziert. Es wird auf Spielfeldern gespielt, die jeder Spieler vor sich liegen hat. Da gibt es eine Extradeckzone, einen Friedhof, eine Deckzone, fünf Monsterkartenzonen, fünf Zauber- und Fallenkartenzonen, eine Spielfeldzone für Spielfeldzauberkarten und zwei Pendelzonen für die Pendelmonster. Tja, und der Swap-Frog ist so eine Monsterkarte.

Auf der Suche nach einem Titelbild für unser vorheriges Forth–Magazin, die Vierte Dimension, Heft 2/2015, hüpfte er mir über den Weg. Hoppla, dachte ich, noch ein Swap, wie unser Swap–Drachen. Aber als magisches Monster habe ich unseren Swap bisher gar nicht gesehen. Denn Forth war ja traditionell verständlich. Sogar der Compiler. Eigentlich. Im Gegensatz zu so manch anderer Computersprache, bei der man nicht so recht weiß, wie da der ausführbare Code zustande kommt. Lange Rede, kurzer Sinn, so kam es, dass der Swap–Frosch zum Symbol der Forthkritik erkoren wurde. Mal sehen, ob er noch weitere Karriere macht.

`https://de.wikipedia.org/wiki/Yu-Gi-Oh!`

`http://de.yugioh.wikia.com/wiki/Austauschfrosch`

Wie geht das eigentlich, chinesische Zeichen über eine Tastatur einzutippen?

Du brauchst einen Input–Method–Editor (IME), in Linux ist das z.B. *Scim*, und eine Input–Methode. Da gibst du dann die romanisierte Umschrift chinesischer Worte ein (pinyin), und wählst unter den gleich ausgesprochenen Zeichen das passende aus. Wenn man mehrere Zeichen direkt eingibt, ist die Auswahl Nummer 1 meistens schon richtig. Der IME lernt auch, was du häufig verwendest, landet dann vorn.

Romanisierte Umschrift bedeutet, dass die Aussprache eines chinesischen Wortes so gut es geht in lateinischen Buchstaben nachempfunden wird. Damit wird, je nach Wahl des Editors, das traditionelle oder vereinfachte chinesische Zeichen gewählt, dem wiederum ein Unicode zugeordnet ist. So können mittels einer gewöhnlichen QWERTY-Tastatur chnesische Zeichen generiert werden.

Das Unicode-Zeichen wird dann durch zwei oder drei Bytes UTF-8 repräsentiert, vom Terminal an das amforth im Target gesendet, und dort ganz wie gewohnt als Name eines Wortes compiliert, also wie jedes gewöhnliche ASCII–Byte im Input–Stream sonst auch. Das funktioniert, solange das Terminal die 8–Bit–Codes unverändert an das amforth überträgt – oder an jedes andere Forth, das volle 8–Bit–Zeichen akzeptiert. Eine Datei mit chinesisch geschriebenem Forth–Quellcode enthält also lediglich diese UTF–8–Codes für die Namen, kann also von jedem Terminal an das Target geschickt werden.

Und ein chinesisch eingestelltes Terminal, das solche 8–Bit–Ausgaben vom Forth ermpfängt, stellt diese dann als chinesische Zeichen dar. Es sind dafür also gar keine Änderungen am Forth erforderlich.

Li Long und sein Team benutzten beim 1. Hackaton an der Shenzhen Graduate School of HIT das *fcitx* und *rime* als IME and Input Method für das amforth. bp/mk

`http://www.chinalink.de/sprache/umschrift.html`

`https://github.com/fcitx/fcitx`

`https://github.com/fcitx/fcitx-rime`

Meetup?

Forth Day! – Meet Chuck Moore – the Inventor of Forth.

Dazu eingeladen hatte Kevin Appert im November des Jahres 2015 per *meetup*. Man erfuhr den Termin, Ort und Link zur Agenda. Meetup behauptet von sich: „...bringt Menschen zusammen, die gemeinsame Interessen teilen um sich persönlich zu treffen. Erwecke Deine Community zum Leben...“ Ob das auch was für uns ist in der Forth Gesellschaft?

`http://www.meetup.com/de/`

You are writing an CPU emulator in TeX, the TYPESETTING system?

Yep.

`https://gitlab.brokenpipe.de/stettberger/avremu`

Da hat doch tatsächlich jemand einen AVR-Simulator in TeX gemacht, gezielt eigentlich für den ATmega8. Christian Dietrich heißt der Mann und *avremu* das Projekt. Auf seinem Laptop erreicht er damit eine Taktrate von 2,5kHz. Bestimmt ein cooles Ding – und herrlich sinnlos. :-) erw

amforth erobert auch den MSP430

Matthias hat angefangen, weitere MSP430–Typen mit amForth zu unterstützen, und zwar den F5529 (Flash) und den FR5969 (FRAM). Die MCU mit dem FRAM ist sehr interessant, weil sie ihren Speicherzustand ohne Strom beibehält. FRAM funktioniert so schnell wie normales RAM und kann wortweise addressiert und geschrieben werden — im Gegensatz zum Flash, welches blockweise gelöscht werden muss. Und es kann beliebig oft geschrieben werden, auch das im Gegensatz zu Flash und EEPROM. Das ermöglicht vermutlich ganz neue Techniken und Stromspardinge.

`http://amforth.sourceforge.net/` erw

amforth cookbook

The Cookbook is a collection of small and not so small recipes. Every recipe is intended to deal with exactly one task. It is a living document, so expect changes at any time. Same link as above. Contents fo far:

Popular Boards: Arduino Hello World, Arduino Analog, AVR Butterfly, Texas Instruments LaunchPad 430, Amforth with Raspberry PI.

Hardware Modules: Dallas 1-Wire Devices, Digital Ports, EEPROM, Efficient Bit Manipulation, I2C EEPROM Blocks, I2C Bus Scanner, I2C EEPROM, I2C Generic, I2C EEPROM VALUE, Interrupt Service Routines, Interrupt Critical Section, NRWW Flash, Serial Peripheral Interface SPI, Telnet Timer, Two Wire Interface TWI/I2C, Usart Settings, Watchdog.

General Code Examples: Defining and using Arrays, Blocks, <BUILDS / DOES>, Using create/does>, Deferred Words, Disabling the terminal command echo, Defining and using Macros, Multitasking, Pitfalls, Saving Power, Redirect IO, Reason For Reset, Simple Strings, Structures, Loop With Timeout, Trouble Shooting, Turnkey applications, Use of the amforth-shell.py utility

Programming and Debugging: Forth Assembler, Build Timestamp, Conditional Interpret, Coroutines, Ctrl-C, Customize AmForth, Debug Shell, Dump Utilities, Exceptions, Extended VM, Un-Doing Definitions, Forward Declarations, Port Code From C, Profiler, Quotations, Efficient RAM Usage, Recognizer, Configuration Stacks, Testing, Tracer, Upgrade AmForth, Unbreakable AmForth, Values, Walking Wordlists, Watcher. mat

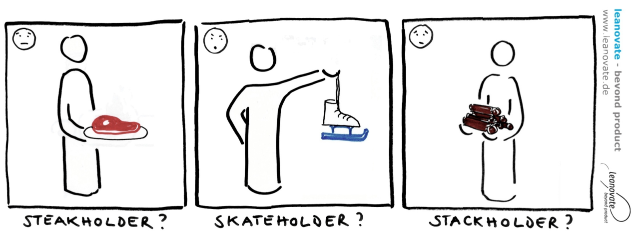
amforth Commented Projects

Diese Kategorie ist ziemlich neu dort. Die Cookbook Einträge sind alle sehr knapp, zwar technisch korrekt, aber für was man das jetzt brauchen kann, ist oft nicht so klar. Deswegen nun die „Commented Projects“. Erich Wälde zeigt wie es gehen sollte am Beispiel seines *Collector–Projects* in dem MCU–Knoten an einem RS485–Bus betrieben werden. Das Netzwerk arbeite sei rund fünf Jahren stabil und zuverlässig in Forth.

Und es sind Links eingestellt worden zu anderen Projektseiten: Dennis Ruffers *Coinforth* samt *ARD101 Tutorial* und das *Cranberry-net* von Andreas Wagner, in dem er ein Remote–Sensing–Network, das in der sumpfigen Umgebung von Cranberry–Feldern und anderen Feuchtgebieten arbeitet, vorstellt. mk

Wohin Tippfehlen einen so führen können...

Strahd von Foguvich meinte ...



If youre creating a software system using Forth, are your clients the Stackholders?

`https://twitter.com/fogus/status/653589535687839745`

cas

Software - Vergangenheit und Zukunft

Charles H. Moore

Charles Moore war einer der Pioniere von Computer-Software-Programmiertechniken. In den späten 1960igern entwickelte er Forth, eine der innovativen Computersprachen jener Zeit. Dies ist das ins Deutsche übersetzte Transkript[1] *seiner in Englisch gehaltenen TED^{X}-Rede in der Davidson Academy, Nevada, USA, gefilmt und hochgeladen ins Netz am 18. Dezember 2014.*

Abbildung 1: C. Moore bei Tedx

Hallo Freunde. Dank fürs Hiersein. In diese Gegend hier zog ich vor ungefähr 15 Jahren, von Silicon Valley her. Und ich bin gern hier. Ich zog erst nach Syracuse City in Californien, das auf dem Pacificrest Trail (oder Long Trail) liegt. Dort bin ich viel gewandert. Dann zog ich weiter, Richtung Ridge of Sierra, dann nach Incline Village, dann Carson City, und jetzt lebe ich in Wellington. Bei all dieser Herumzieherei ist die Stadt Reno mein Ausgangspunkt geblieben. Die Bibliotheken sind hier ausgezeichnet. Ich bin ein großer Bibliotheks-Fan. War das eigentlich schon immer, seit ich mit dieser Art von Software-Entwicklung begann.

Ich erfand die Sprache Forth im Jahre 1968. Das war dieselbe Zeit, da auch C erfunden wurde. Es war eine überaus kreative Zeit für die Erfindung von Sprachen - Computer-Sprachen. Fortran, COBOL, Algol, Lisp, C.

Prinzipiell gibt es drei Arten von Computer-Sprachen - wenn man die Notation betrachtet. In C z.B. gibt es Ausdrücke wie "A + B gleich irgendwas". Man nennt das Infix-Notation. Der Operator befindet sich zwischen den beiden Argumenten. Lisp verwendet die Präfix-Notation, + A B , bei der der Operator vornweg steht. Forth verwendet Postfix-Notation. Der Operator steht hinten. Meiner Meinung nach ist das die weitaus beste Notation [die Zuhörern lachen]. Sie führt zu wesentlich einfacheren Programmen. Nehmen wir beispielsweise C. Da gibt es eine Menge Klammern. In Lisp stehen noch viel mehr Klammern. Und in Forth? Da gibt es überhaupt keine Klammern. Das ist also eine viel natürlichere Sprache für ein menschliches Wesen, das nicht mit Algebra auf irgendeiner höheren Schule gequält wurde. Algebra zwingt einen dazu, die Operatoren zwischen die Argumente zu setzen. Das ist höchst unnatürlich. Der Operator sollte wirklich zuletzt kommen. Man hat erst eine Reihe von Zahlen. Die Summe steht unten. Die Summe ist das Letzte, was man zu bilden hat.

Ich entwickelte also Forth und ich versuchte, hm, ich konnte es sehr gut bei meinen Programmierarbeiten gebrauchen. Ich überzeugte ein paar andere Leute, es zu verwenden. In den 70iger-Jahren existierte eine ansehnliche Gemeinschaft von tausenden von Forth-Programmierern überall auf der Welt. Viele von ihnen

[1] *Übersetzt von Fred Behringer*

in Deutschland, in Russland und in China. Aber das hörte dann auf. Es gibt immer noch tausende von Forth-Programmierern, aber man sieht sie nicht. Der Grund liegt zum Teil darin, dass man Forth als Betriebsgeheimnis betrachten kann. Wenn man Forth-Programme schreibt, werden diese kleiner, schneller und fehlerfrei sein. Und das sind Eigenschaften, die die Leute nicht gern mit anderen teilen. Die behalten sie gern für sich.

Ich war sehr stolz darauf, dass ich Programme schreiben konnte, die mich in die Lage versetzten, Dinge zu tun, die ich sowieso tun wollte. Ich musste Programme für andere Leute schreiben, um Geld zu verdienen. Aber das waren nicht die schlechtesten Erfahrungen. Es ist immer besser, Aufgaben, die mir gestellt werden, auf meine Weise zu erledigen. So ziemlich das Erste, womit ich mich beschäftigte, war der Entwurf von Leiter-Platinen. Dieses Bild da kommt mir gerade recht [Lachen bei den Zuhörern].

Ich würde vorschlagen, dass sich einige von euch, ein paar Jüngere unter euch mit dem Schreiben eigener Software beschäftigen. In den 70gern hat man das getan. Da konnte man keine Software kaufen. Da gab es keine Software, die man kaufen konnte. Und ich dachte, in Zukunft würde jeder seine eigene Software schreiben, in der wirkungsvollsten und einfachsten Sprache, die man sichvorstellen konnte. Dann aber kam Bill Gates und überredete alle Welt zum Kauf von Software. Zum Kauf, nicht zum Selbermachen. Das eigentlich Geniale an ihm war, dass er die Leute dazu überredete, immer wieder zu kaufen. Immer wieder und immer wieder [Lachen bei den Zuhörern]. Und genau so hat er es geschafft, die Fehler in den ursprünglichen Versionen und den Mangel an brauchbaren Eigenschaften auszubügeln.

Forth-Software ist einfacher als C-Software. Windows ist in C programmiert, Linux ist in C programmiert, OS ist, glaube ich, in C programmiert. Es gibt in der heutigen Welt nur zwei Sprachen. Drei: C, Java und Forth. Und von Forth hört man nichts - obwohl es den Saturn umkreist.

Das erste Programm, das ich für den eigenen Gebrauch schrieb, war ein Programm zum Entwerfen von Leiter-Platinen. Und alles, was ich dazu brauchte, war ein kleines Forth-Programm und die Möglichkeit, Rechtecke zu zeichnen. Das ist wirklich eine gute Übung in der Kunst des Programmierens, wenn man eine Sprache findet, irgendeine Sprache für einen Computer, den man hat, und Rechtecke zeichnet, Rechtecke, die man herumschiebt, und Leiter-Platinen entwickelt. Die Rechtecke kann man sehen. Die meisten Leitungen sind Bestandteile von Rechtecken, diese hier . . . hm, diese Kontakte hier sind Hexagone, aber sie könnten Rechtecke sein. Und mit alldem könnte man etwas Nichttriviales tun - und dabei eine Menge Spaß haben. Man kann verschiedene Farben verwenden. Bunte Bilder verschieben. Wirklich viel Spaß!

Warum tut man das? Ganz einfach: Die bestehenden Sprachen können alles. C wurde dazu entworfen, alles damit zu tun, was man tun möchte. Mit Windows kann man alles Mögliche machen. Mit Forth ist das anders. Mit Forth kann man auch alles machen, was einem einfällt, aber nicht von vornherein. Forth ist das, was ich erweiterbar nennen möchte. Wenn man in Forth einen neuen Befehl einbauen möchte, einen, den man vorher noch nicht ins Auge gefasst hatte, so kann man das ganz leicht tun. Man kann sich eine Sprache bauen, die in jede nur denkbare Richtung geht. Am Anfang ist das große Nichts. Man braucht sich also nicht mit irgendeinem Overhead herumzuschlagen. Und der Overhead ist wichtig.

Abbildung 2: Das wäre ein Papierstapel von dieser Höhe...

Um eine Leiter-Platine zu entwerfen, braucht man etwa ein Megabyte an Code. In Forth würde das nur ein Kilobyte ausmachen. Drei Größenordnungen weniger Code. Drei Größenordnungen, nicht 1% , sondern ein Zehntel von einem Prozent weniger Code. Ich habe das x-mal gesagt, aber keiner interessiert sich dafür. Eine ganze Industrie wurde um große ineffiziente und fehlerhafte Software herum aufgebaut. Angefangen bei Windows [Lachen bei den Zuhörern]. Das gelangt sogar ins Standard-Komitee. Man denke nur an diesen Fehler, den man kürzlich im Secure-Socket-Layer fand. Man kann das als schockierend betrachten, man kann es als Betrug bezeichnen, man kann sagen, es sei eine Verschwörung. Aber das ist großes Geschäft. Die Leute sind bereit, Geld auszugeben. Es schafft Arbeitsplätze. Wer hat denn schon Interesse an einer destruktiven Technologie, die 9 von 10 Millionen Programmierern auf die Straße setzt? Aber man kann ja auch seine eigene Software schreiben. Man bleibt dabei sein eigener Herr und es mangelt nicht an Zufriedenheit. Es macht viel mehr Spaß als das Raten von Kreuzworträtseln.

Nachdem ich Leiter-Platinen entworfen hatte, dachte ich, es würde Spaß machen, Computer-Chips zu entwerfen. Ich blieb also bei derselben Vorgehensweise wie bei den Leiter-Platinen und entwarf einen Computer-Chip, drei Größenodnungen weniger Aufwand als bei anderen Programmierern. Das ist gar nicht so schwierig, wie man vielleicht meinen könnte. Nein, es ist wirklich ganz leicht. Alles, was man dazu braucht, sind Rechtecke, die man hin und her schiebt. Dass diese Rechtecke ganz klein sind, statt makroskopisch groß, ist nicht so wichtig. Und man legt diese Rechtecke schön übereinander, drei oder vier Schichten, und fertig ist der Computer-Chip.

Abbildung 3: Vielleicht werde ich das in Zukunft tun!

Einen solchen Chip tatsächlich herzustellen, kostet von 5000 Dollar bis etwa 25 Millionen, je nachdem, was man eigentlich genau machen möchte. Das kann man sich wirklich nicht leisten. Aber den Chip tatsächlich herzustellen, ist nicht so wichtig. Was man eigentlich möchte, ist, ihn zu entwerfen, ihn zu simulieren, und ihn dann im virtuellen statt im reellen Raum auszutesten. Wenn man einen wirklich gelungenen Entwurf hat, zu dem man steht, und wenn man jemanden davonüberzeugen kann, ihn zu finanzieren, dann hat man ein Produkt. Nun, wir haben ein Produkt: Wir haben 144 Computer auf einem 1-cm-Chip in einer recht konservativen Technologie. Die Kosten belaufen sich auf 5000 Dollar, wenn man von kleinen Stückzahlen ausgeht. Es kann also gemacht werden.

Warum tut man sowas? Eigentlich tue ich das, um zu beweisen, wie schlecht andere Leute arbeiten [Lachen bei den Zuhörern]. Meine Software zum Entwerfen solch eines Chips benötigt etwa 100 KB. Cadence und Retro Graphics, die Standard-Software-Werkzeuge, die die Industrie zur Herstellung von Chips verwendet, benötigen 100 MB. Man hat schon von Programmen gehört, die eine Million Code-Zeilen enthalten.

Ich weiß, dass die Software, die in den F-22-Kampfflugzeugen verwendet wird, mehr als eine Million Code-Zeilen enthält - mehrere Millionen. Eine Million Codezeilen ist Wahnsinn. Das wäre ein Papierstapel von dieser Höhe. Eine Million Codezeilen kann man nicht lesen, geschweige denn schreiben. Man benötigt tausend Leute, um eine Million Codezeilen zu schreiben. Keiner weiß, wozu die gut sind. Keine einzelne Person weiß das. Man braucht die gesamte Firma, um hinter das Geheimnis des Programms zu kommen. Und kein Einziger in der ganzen Firma gibt einem eine Antwort auf die Frage, wozu das Programm gut ist, sollte sie jemals gestellt werden. Man muss sich darauf verlassen, dass alles irgendwie gut zusammenarbeitet. Und wir alle wissen, wie schwer das ist.

Windows kommt jedes Jahr mit einer weiteren Version heraus und es hat, nun, ich weiß nicht genau wieviel Codezeilen, aber es werden sicher mehr als eine Million sein. Große Programme. Große Programme sind gut für die Wettervorhersage, für die Simulation von Atomtests, für die Koordination der Schiffbewegungen in der Welt. Es gibt große Probleme und es gibt große Datenmengen. Aber dazu braucht man keine großen Programme. Man kann das deutlich sehen, wenn man versucht, eine App auf das Smartphone zu laden. Die Apps enthalten 10 MB an Code. 10 MB. Würde man das in Forth machen, dann käme man mit 10 kB aus. Andererseits würde ich sie das ganze Phone programmmieren lassen, um überhaupt einen Nutzen aus der ganzen Sache zu ziehen. Bis jetzt zögere ich noch, auf dem Smartphone zu programmieren. Vielleicht werde ich das in Zukunft tun! [Die Zuhörer lachen] Ich habe einhundertvierundvierzig kleine Computer. Die können das tun.

Zum Abspeichern von Bildern braucht man Speicherplatz, für große Datenbanken braucht man Speicherplatz. Speicherplatz ist wirklich gut, aber Speicherplatz ist nicht gut zum Speichern von Programmen. 10 MB an

Programm-Code enthält Fehler. 10 KB nicht. Wer Effizienz will, wer Kompaktheit will, wer Zuverlässigkeit will, braucht kleine Programme. Das heißt, das Programm kann nicht alles machen, aber man kann alles mit den Mitteln des Programms machen.

Wir haben einen Browser in Forth geschrieben. Er war klein. Er war nicht ganz so klein, wie ich es haben wollte. Mir gefiel die Art nicht, wie das angepackt wurde. Aber es ist machbar: Man kann eine E-Mail-Anwendung schreiben. Wie arbeitet das Telefon, wie arbeitet überhaupt alles auf dieser Welt? Die Leute schreiben kaum noch Software. Sie nehmen Software-Pakete aus den verschiedensten Quellen und kleistern sie zusammen. Es ist dasselbe wie mit der Hardware. Die Handys enthalten einen Chip, der erstaunlich viele Sachen macht: Eine Komponente von hier, eine Komponente von dort, und alles zusammengelötet. Das verbraucht zehnmal so viel Energie, zehnmal so viel Platz, zehnmal so viel Entwiclungsarbeit, zehnmal mehr als nötig. Aber so machen die Leute das heutzutage. Keine Verschwörung. Das scheint einfach der einfachste Weg zu sein. Das ist die Art und Weise, in der es die Firmenchefs von euch gern gesehen hätten. Es braucht aber nicht so zu sein. Ihr könnt es besser machen, kleiner, schneller, billiger - und es bleibt der jüngeren Gereration vorbehalten, dafür zu sorgen, dass sich an den überkommenen Zugeständnissen einiges ändert. Erreichen könnt ihr das dadurch, dass ihr ein kleines Programm schreibt, das genau das tut, was ihr haben wollt. Hebt das entsprechend hervor und beweist es. Schreibt eure eigene Software.

Eines der Gründe, weshalb Forth nie zum Zug kam: In Forth ist es überaus leicht, ein Betriebssystem zu bauen. Alles, was man dazu braucht, sind 10K an Code. Jedermann, der sich für Forth interessiert, hat sein eigenes System geschrieben. Es gibt also eine Menge Systeme in der Welt. Wenn man mehr darüber wissen will, braucht man nur herumzugooglen. Man findet jede Art von Hinweisen. Ich habe da eine Webseite unter dem Namen colorforth.com. Das ist COLORFORTH. DieFirma, mit der ich jetzt zusammenarbeite, heißt Green Arrays. Die Firma hat eine Webseite. Es existiert eine Webseite unter dem Namen forth.com. Forth.com gibt es schon immer, solange es das Internet gibt. Informationen über Informationen. Es macht Spaß, herumzustöbern. Und mir macht es Spaß, mit Forth in Verbindung gebracht zu werden.

Die herkömmliche Art, die Dinge zu betreiben, ist nicht immer die richtige. Vielleicht sollte alles nur ein historischer Zufall gewesen sein. Und das gilt in gewisser Weise auch für uns. Forth kam mehr oder weniger zufällig in die Welt und wurde nur zufällig zu dem, was es ist. Wenn ich nicht Forth erfunden hätte, wäre es wahrscheinlich überhaupt nicht erfunden worden. Es hat keine große Zustimmung gefunden und es bringt einen nicht viel weiter, wenn man sich damit beschäftigt. Aber es hat sich als eine gute Sache erwiesen, auf die ich sehr stolz bin. Ich danke euch fürs Zuhören. [Applaus]

(Quelle: http://tedxtalks.ted.com/video/Software-Past-and-Future-|-Char;search%3Amoore)

Why should we care about the geometric signs?

: , . + @ # ' ! ~ ; ?] (

„... I believe that the abstract nature of these signs is some of the best proof we currently have that these images were not being made purely for their aesthetic quality. It suggests a more symbolic role for these markings, and a desire to communicate ideas ...“ Aus: Initial findings from the study of 146 French rock art sites, by Genevieve von Petzinger. (Quelle: http://bradshawfoundation.com/geometric_signs/geometric_signs_france.php)

A Hackathon in Shenzhen Graduate School and the "Lambda–Tortoise"

Li Long "Atommann"

About two weeks ago[1] I met a pretty lady in our office, she told me there is a Hackathon hold by their company and Shenzhen Graduate School of HIT [2]. At the first moment I had no idea what to build.

Me: "What's the topic?" The pretty lady: "Robots" Me: "Good topic! I'll send an email to our mailing list to ask if someone else want to join."

When I wrote that email I suddenly realized that I can build a robot using amforth with a bluetooth module, it can accept simple commands from computer side and draw graphics on a white board on the floor.

Teamwork

So the idea was clear. I decided to go. But I just started to learn forth and I could just write some simple code in *amforth*. Fortunately we just have a coder in our hackspace and he just graduated from applied math department and implemented a Forth dialect with x86 assembler. But he had no MCU knowledge.

But it's OK. I could finish the low level code (such as the code for servo motor which used to implement words of pen–up and pen–down) and he could finish the high level functions. Meanwhile another guy, "bnw", finished the mechanical drawing with inkscape, assembled the robot and helped to configure the Bluetooth module in GNU/Linux.

A funny story during the hackathon: In a deep night we sat on a terrace and were drinking coffee and discussing which kind of graphics the robot should draw for the final demonstration.

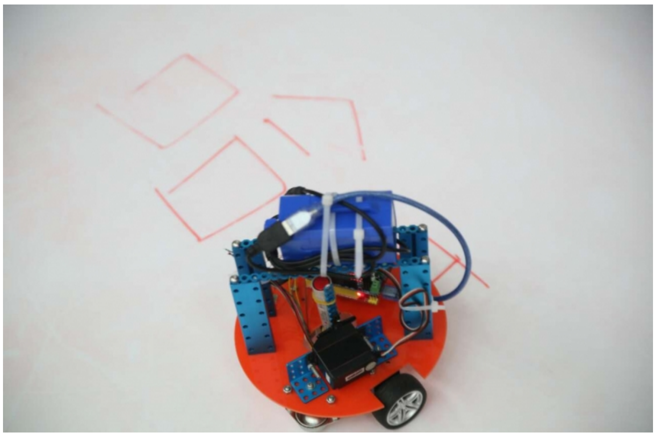


Abbildung 1: The „lambda-tortoises" robot

xyh (the forth coder): "I want to draw three Chinese characters" (The Chinese name of the university) Me: "You are crazy! Why not just draw HIT, all straight lines, easier." I was tired and went to sleep and didn't put xyh's idea of drawing Chinese in mind.

The next morning when I woke up I went to see xyh's job. I saw there is a square on the white board paper on the floor. "Wow, surprise!" I said. Suddenly I found that he was coding in Chinese in Forth. All the words are defined in Chinese. OMG. And I never expected this before. You find the code which draws Chinese for the final demonstration at [6].

Final demonstration

Just before the final demonstration, we found that the robot can not draw the first Chinese character nicely enough. xyh change the code time by time.

Me: "xyh, I think it's good enough now" xyh: "No, I can make it better! This character is worth 10k RMB[2]!"

And the good thing of *amforth* is that we can upload the new words to the robot by Bluetooth! We don't need to plug cable, unplug cable and no need of edit, compile, download cycle which wastes time which is valuable for a hackathon. We finished the robot and got the No. 1 prize! You will find the scheme code for our "lambda-tortoises" robot at [4], the motor driver we used at [1], very handy! And more photos at [3].

Abbildung 2: The winning team receives the prize from Haier (white shirt). Next to him, left to right, the team: "Atommann" Li Long (orange), "xyh" Yuheng Xie (black), "bnw" Fonzie Huang (red).

[1] *Got his email on amforth developer list 2015-10-20 05:15 CET*

[2] The renminbi (RMB) is the official currency of the People's Republic of China. The name literally means "people's currency."

Perspectives

The original idea of this project was to teach kids programming, English, and how to control robots. At our computer site, next thing I want to build is a mini computer based on Raspberry Pi, or the 9 dollar C.H.I.P. computer [5], battery powered. So kids can have their own computer and a robot they can play with. In our computer site, kids can use Lisp language as well.

Amforth is awesome! Thanks. We'll continue.

Annotation

“Atommann” is the nickname Li Long used for years. Because he likes physics, and the physics concept of “everything is made of atoms” is so important and a *Mann* is also made of atoms. Yuheng Xie calls himself “xyh” and Fonzie Huang is called “bnw”. So look for these Aliases while searching the web.

There will be 14 hackathons from 2015 to 2016 in different cities in China, and this was the first one. The 15th will be in Bay Area USA.

Abbildung 3: Participants of the 1. hackathon in 2015

Referenzen

[1] https://github.com/wmercer/amforth/blob/609c6594076422e6592f50811ef73f054b02a256/community/Arduino-Shields/ada-shield.frt

[2] https://en.wikipedia.org/wiki/Shenzhen_Graduate_School_of_Harbin_Institute_of_Technology

[3] http://www.leiphone.com/news/201510/Hk7mizNbynrIInam.html

[4] https://github.com/szdiy/lambda-tortoise

[5] https://www.kickstarter.com/projects/1598272670/chip-the-worlds-first-9-computer/description

[6] https://github.com/xieyuheng/ada-shield/blob/master/show.org

[7] https://de.wikipedia.org/wiki/Haier

Listing

Primitive Functions

```
variable time-pwm:help-var

: time-pwm:turn-left ( time pwm -> [move] )
    time-pwm:help-var !
    forward time-pwm:help-var @ 15 * 14 / 2 m-run
    backward time-pwm:help-var @ 3 m-run
    ms
    release 2 m-run
    release 3 m-run
;
: time-pwm:turn-right ( time pwm -> [move] )
    time-pwm:help-var !
    backward time-pwm:help-var @ 15 * 14 / 2 m-run
    forward time-pwm:help-var @ 3 m-run
    ms
    release 2 m-run
    release 3 m-run
;
: time-pwm:forward ( time pwm -> [move] )
    time-pwm:help-var !
    forward time-pwm:help-var @ 15 * 14 / 2 m-run
    forward time-pwm:help-var @ 3 m-run
    ms
    release 2 m-run
    release 3 m-run
;
: time-pwm:backward ( time pwm -> [move] )
    time-pwm:help-var !
    backward time-pwm:help-var @ 15 * 14 / 2 m-run
    backward time-pwm:help-var @ 3 m-run
    ms
    release 2 m-run
    release 3 m-run
;

variable pwm-for-360
\ 1530 pwm-for-360 ! \ 80
\ 1430 pwm-for-360 ! \ 8.37
\ 1390 pwm-for-360 ! \ 8.37
\ 1380 pwm-for-360 ! \ 8.37
1385 pwm-for-360 ! \ 8.37

\ 360 左轉角

: 左轉角 ( angle -> [move] )
    5 *
    pwm-for-360 @
    time-pwm:turn-left
;
: 右轉角 ( angle -> [move] )
```

```
      5 *
      pwm-for-360 @
      time-pwm:turn-right
;
: 步前進 ( step -> [move] )
      200 *
      pwm-for-360 @ time-pwm:forward
;
: 步後退 ( step -> [move] )
      200 *
      pwm-for-360 @ time-pwm:backward
;
: 提筆 ( -> )
    300 ms
    1350 OCR1A high!
    400 ms
;
: 落筆 ( -> )
    300 ms
    1105 OCR1A high!
    400 ms
;
: servo-init
      1 1 lshift DDRB or!
      20000 ICR1 high!

      1 7 lshift
      1 1 lshift or
      TCCR1A c!

      1 4 lshift
      1 1 lshift or
      TCCR1B c!

      提筆
;
: 微秒稍等 ms ;
```

HIT (哈工大)

```
: 口字旁
    2 步前進
    落筆      2 步前進
    90 右轉角
              2 步前進
    90 右轉角
              2 步前進
    90 右轉角
              2 步前進
    提筆
              2 步後退
    90 右轉角
;

: 人字頭
    15 右轉角
              3 步前進
    40 右轉角
    落筆      2 步前進
    300 微秒稍等
    75 右轉角
              2 步前進
    提筆
              2 步後退
;

: 寫個一
    90 右轉角
              2 步前進
    落筆
    140 左轉角
              2 步前進
    提筆
              2 步後退
;

: 口字底
    90 右轉角
              1 步前進
    落筆
              2 步前進
    提筆
              2 步後退
    90 左轉角
    落筆
              2 步前進
    90 右轉角
              2 步前進
    300 微秒稍等
    90 右轉角
    落筆
              2 步前進
    提筆
              2 步後退
              1 步後退
    90 右轉角
;

: 哈
    口字旁
    人字頭
    寫個一
    口字底
;


: 短横
    20 右轉角
              4 步前進
    70 右轉角
    落筆
              2 步前進
    提筆
;

: 豎
              1 步後退
    落筆
    90 右轉角
              3 步前進
    提筆
;

: 底部長横
    90 右轉角
              2 步前進
    落筆
              4 步後退
    提筆
    90 右轉角
    300 微秒稍等
    25 右轉角
;

: 工
    短横
    豎
    底部長横
;

: 中部長横
          3 步前進
    落筆
    90 右轉角
          4 步前進
    提筆
          2 步後退
    90 左轉角
;

:
          1 步前進
    落筆
          3 步後退
    45 右轉角
          3 步後退
    提筆
          3 步前進
;

:
    89 左轉角
    落筆
          3 步後退
    提筆
;

: 小車閃開
    5 步後退
;

: 大
    中部長横


    小車閃開
;

哈 工 大
```

小魚

```
: 小魚的脊背
    30 右轉角
    落筆
        4 步前進
    30 右轉角
        2 步前進
    30 右轉角
        3 步前進
    45 右轉角
        2 步前進
    提筆
;

: 小魚的肚子
    85 右轉角
    落筆
        2 步前進
    45 右轉角
        3 步前進
    30 右轉角
        4 步前進
    提筆
;

: 小魚的眼睛
    155 右轉角
        6 步前進
    落筆
        360 右轉角
    提筆
;

: 小車走開
    4 步前進
;

: 小魚
    小魚的脊背
    小魚的肚子
    小魚的眼睛
    小車走開
;
```

The N.I.G.E. Machine

Andrew Read

Versetz dich zurück in die 1980er (Gratulation all jenen, welche diese Vorstellungskraft besitzen). Du programmierst deinen Heimcomputer. Vielleicht ist es ein Commodore PET (wie in meinem Fall), ein Commodore 64, oder sogar ein Electronika BK? Es gäbe so viele Programme, die geschrieben werden wollen, doch die Maschine zwängt dir ihre Grenzen auf! War es nicht auch dein Traum, die Maschine möge mit 100MHz laufen, statt mit 1MHz zu zuckeln? Wie wäre es mit 16MiB RAM statt 16 KiB? Was wenn wir 240 Zeichen pro Zeile hätten, statt nur 40? Oh und dann sollte natürlich noch Forth nativ laufen statt BASIC? Und Multitasking? Tja, ich habe geträumt! Allerdings benötigte es 30 Jahre, um diesen Traum Realität werden zu lassen. In diesem Artikel möchte ich ein Projekt von mir vorstellen, genannt "N.I.G.E. Machine". Die N.I.G.E. Machine ist Open Source, und ich möchte nicht nur die Idee, sondern die Maschine selbst mit jedem Interessierten teilen.

Was genau ist diese N.I.G.E. Machine? Ich kann mit einer Gegenüberstellung anfangen. Du kennst vielleicht einige der exzellenten Forth-Kerne hier aus Deutschland oder aus Übersee. Zum Beispiel Bernd Paysans b16, Klaus Schleisieks MicroCore oder der J1 von James Bowman. Die N.I.G.E. Machine ist kein Forth-Kern (allerdings enthält sie einen)! Sie ist ein stand-alone Microcomputer, welcher *Forth* als seine native Programmiersprache verwendet, und vollständigen Peripheriezugriff bietet. Eine Zusammenfassung der Spezifikationen der aktuellen Version findet sich in Tabelle 1

CPU-Typ	32-Bit-Stackmaschine
Taktfrequenz	100 MHz
Systemspeicher	128 KiB
Externer Speicher	16 MiB
Programmierumgebung	Forth mit hardware-unterstütztem Multitasking
Grafikauflösung	Full HD 1920 x 1080
Grafiktyp	Zeichenbasiert
Farben	256 am Bildschirm von insgesamt 4096
Tastatur	USB
Serieller Anschluss	RS232
Massenspeicher	FAT32 auf einer microSD-Karte
Erweiterungen	5 x 10 Kanel I/O Ports

Tabelle 1: Spezifikationen der N.I.G.E. Machine

Bevor ich weitermache, würde ich mich gerne vorstellen. Mein Name ist Andrew Read. Ich bin Engländer, lebe aber schon viele Jahre in Übersee. Zur Zeit bin ich Geschäftsmann in Hong Kong. Ich habe einen Abschluss in Naturwissenschaften von der Cambridge University. (Ich hoffe, vielleicht einmal die Zeit zu finden, mich weiter mit dem Studium der Experimentalwissenschaften zu beschäftigen). Wenn ich mal gerade nicht mit meinem Digitaltechnikhobby beschäftigt bin, gehe ich gerne mit meiner jungen Familie wandern, erforsche historische Sehenswürdigkeiten, oder beschäftige mich mit klassischem Latein.

Abbildung 1: Andrew Read mit seiner jungen Assistentin

Wo hatte die N.I.G.E. Machine ihre Anfänge? Ich habe als Assistent im Team des NATAMI-Projekts etwas VHDL gelernt. Das war ein Unterfangen zur Neuimplementierung des Amigas auf aktueller Hardware. Leider hat sich das Team nach einiger Zeit aufgelöst, aber ich wollte weiterarbeiten. Anstatt den Amiga weiterzuverwenden, habe ich mich entschlossen, einige meiner eigenen Ideen zu verwirklichen.

Systemübersicht

Solch ein Projekt benötigt eine gute Hardwareplattform. Der Einfachheit und Flexibilität halber verwende ich das Digilent Nexys 4 educational circuit board - Abbildung 2. Die Schaltzentrale dieser Platine ist ein wiederprogrammierbarer Siliziumchip (ein FPGA, in diesem Fall ein Xilinx ARTIX-7), welcher das digitale Design beheimatet. Die Platine beinhaltet noch viel mehr Komponenten. Für die Anwendung als Mikrocomputer sind dies die unverzichtbaren: Der USB Tastatur- und VGA-Anschluss, das externe RAM (16 MiB), einige Expansionsanschlüsse sowie ein paar Schalter und LEDs.

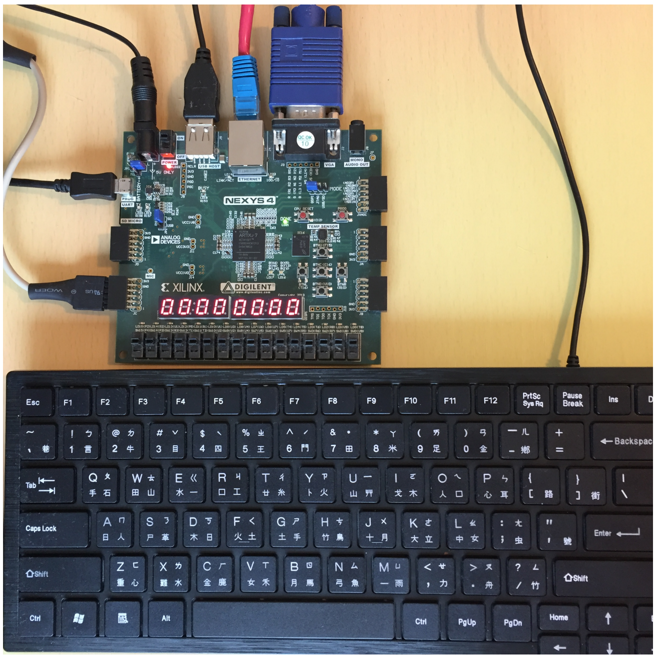


Abbildung 2: Die N.I.G.E. Machine mit Tastatur

Wenn du die N.I.G.E. Machine einschaltest, wirst du von einer Power-On-Message begrüßt (Beeinflussung vom Commodore PET!) und du bist sofort in der Forth-Umgebung - Abbildung 3.

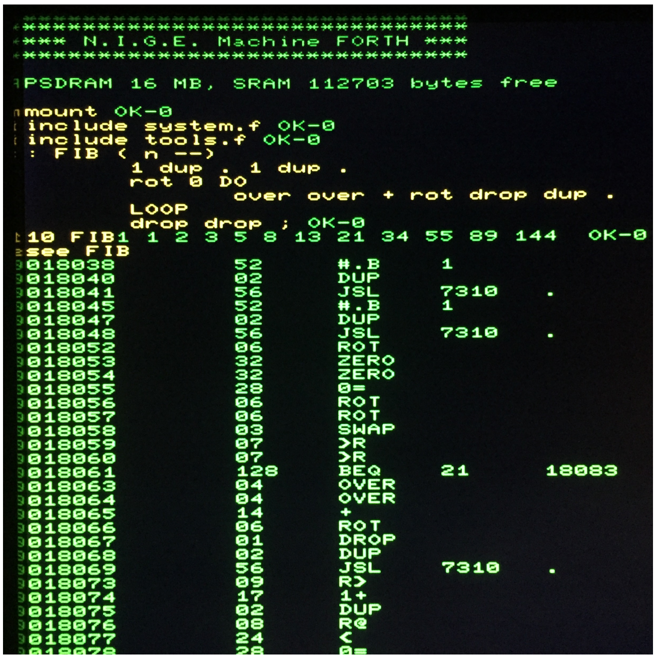


Abbildung 3: Screenshot der N.I.G.E. Machine

Die Hardware der N.I.G.E. Machine ist modular um eine maßgefertigte 32-Bit-Softcore-CPU organisiert - Abbildung 4. Die CPU hat zwei getrennte Speicherschnittstellen (SRAM und PSDRAM, siehe unten) mit separaten Speichercontrollern. Zusätzlich gibt es vier Speicherblöcke, welche als Stacks organisiert sind (Parameterstack, Returnstack, Unterprogrammstack und Ausnahmestack). Die CPU hat auch Zugriff auf Hardwareregister, welche aus FPGA-Logikbausteinen zusammengesetzt sind. Dieses Modul stellt I/O und Konfigurationsregister im Adressraum der CPU zur Verfügung. Ein Interruptcontroller übernimmt die Annahme und Priorisierung von externen Interruptanforderungen und leitet diese an die CPU weiter. Das Systemtaktmodul stellt einige Taktsignale zur Verfügung, unter anderem den 100-MHz-Systemtakt. Weitere Module sind ein RS232-, ein Tastatur-, ein SPI- und ein VGA-Controller.

Systemkomponenten

Gehen wir nun Schritt für Schritt durch die Komponenten.

Softcore-CPU

Die Softcore-CPU ist eine 32-Bit-Stackmaschine, optimiert für die Programmiersprache Forth. Beim Entwickeln der CPU habe ich die Herausforderung gesucht, ein Gerät von professioneller Qualität zu entwickeln und dabei neuartige Funktionen einzuführen. Das ging nicht von heute auf morgen, ich habe 2010 begonnen und die neueste Version stellte ich auf der EuroForth 2015 vor. Hier sind die Kernfunktionen:

- Deterministische Ausführung. Alle Instruktionen, auch bedingte Sprünge und lokaler Speicherzugriff, benötigen eine fixe Anzahl von Zyklen. Zusätzlich ist die Ausführungspipeline so gestaltet, dass keine Konfliktstati auftreten können, welche fehlende Zyklen verursachen. Das ermöglicht eine deterministische Ausführung und erlaubt die Generierung periodischer Signale ohne Fluktuationen (Jitter).
- Schnelle Interruptantwortzeit. Da die CPU stackbasiert ist, besteht nicht die Notwendigkeit, Register beim Aufrufen von Serviceroutinen zu sichern respektive beim Verlassen wiederherzustellen. Die typische Interruptantwortzeit der N.I.G.E. Machine beträgt 4 Zyklen: 2 Zyklen, um in die Interruptvektortabelle zu springen, plus 2 Zyklen, um zur Serviceroutine selbst zu springen.
- Hoher Instruktionsdurchsatz. Die dreistufige Pipeline der CPU erreicht für die meisten Instruktionen einen Durchsatz von einer Instruktion pro Taktzyklus.
- Hohe Codedichte. Die CPU nutzt Mikrocode anstelle eines fest verdrahteten Dekodierers. Dadurch ist es möglich, mit relativ kleinem Opcode viele Kontrollleitungen anzusteuern. Alle Instruktionen sind als einzelne Bytes codiert, optional von anwendbaren Literalen gefolgt. (Anmerkung: Klaus Schleisiek schlägt

in seinen Werken allgemein vor, dass längere Opcodes von z.B. 19 Bit möglicherweise optimal für die Codedichte wären, sofern die CPU dadurch Parallelisierung im Datenpfad und den Registern ausnutzen kann).

- Schnelle Sprungleistung. Auf der N.I.G.E. Machine benötigen bedingte und unbedingte Sprünge (BEQ bzw. BRA) zwei Taktzyklen.
- Schnelle Ausnahmebehandlung. Die N.I.G.E. Machine verfügt über dedizierte Hardwarestacks für Unterprogramme sowie Ausnahmeframes und implementiert `CATCH` und `THROW` als atomare Instruktionen, welche in 2 bzw. 3 Taktzyklen ausgeführt werden.
- Schnelles Multitasking. Die N.I.G.E. Machine unterstützt Hardwaremultitasking: 32 Tasks sind verfügbar. Eine komplette Taskumschaltung benötigt 2 Taktzyklen. Sowohl kooperatives als auch preemptives Multitasking werden unterstützt.

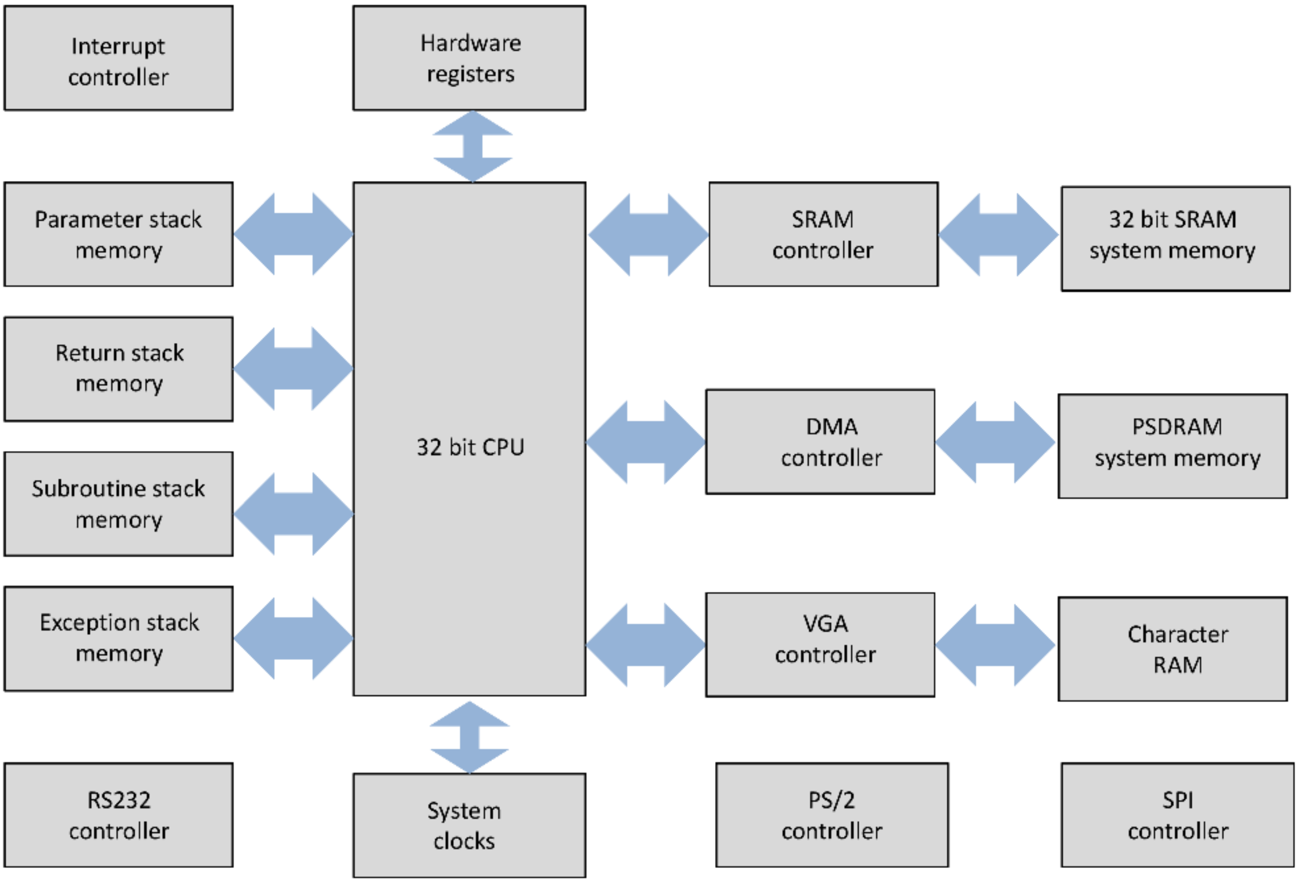


Abbildung 4: Die Systemtopologie der N.I.G.E. Machine

Anmerkung: Das gesamte CPU-Design (und der Rest der N.I.G.E. Machine) ist meine eigenhändige Arbeit. Die große Ausnahme ist die Hardwaremultitaskingfunktionalität, welche das Ergebnis der einjährigen, fruchtbaren Zusammenarbeit mit Ulrich Hoffmann ist (und weitergeht). Nach 4 Jahren alleiniger Arbeit am Projekt ist es ein großartiges Gefühl, die Kräfte mit einem Enthusiasten wie Ulli zu vereinen.

Systemspeicher ("SRAM")

Der CPU stehen 128KiB Speicher zur Verfügung, welcher byteadressierbar ist. Daten und Code können ohne Einschränkung abgelegt werden. Die CPU speichert Wörter und Langwörter im Big-Endian-Format. Der SRAM-Speicher ist wie folgt für Forth und andere Anforderungen eingebetteter Systeme optimiert:

- Schneller Speicherzugriff. SRAM ist als FPGA BLOCK RAM implementiert. Alle Lade- und Speicherbefehle an das SRAM benötigen deterministisch 2 Taktzyklen.
- Flexibler Speicherzugriff. Der Datenbus zwischen CPU und SRAM ist 32 Bit breit. Die CPU stellt eigene Befehle zum Laden und Speichern von Bytes, Wörtern und Langwörtern zur Verfügung. Unausgerichteter Zugriff wird vom SRAM-Controller unterstützt: Langwörter und Wörter können ohne zusätzliche Taktzyklen gelesen und geschrieben werden.

Externer Speicher ("PSDRAM")

Der externe Speicher besteht aus einem 16MiB großen pseudo-statischen dynamischen RAM (PSDRAM) - Modul auf dem Nexys 4. Der Speicher ist für Applikationsdaten gedacht, der CPU ist es nicht möglich, Code direkt vom PSDRAM auszuführen. CPU-Zugriff auf das PSDRAM wird von einem DMA-Controller gesteuert, welcher auch den Zugriff des VGA-Controllers auf den Bildschirmpuffer verwaltet.

Hardwareregister

Das Hardwareregistermodul stellt Zugriff auf die speicherabgebildeten Ein-Ausgaberegister sowie Systemkonfiguration zur Verfügung. Register können 1 bis 32 Bit breit sein, müssen jedoch alle langwortausgerichtet sein. Register können les-, schreibbar oder beides sein. Die Ausführung eines Schreibbefehls auf ein bestimmtes Register (z.B. RS232DOUT) löst die relevante Funktion im Hardwaremodul aus. Zusätzlich können benutzerdefinierte Register erstellt werden, um eigene VHDL-Komponenten an die N.I.G.E. Machine anzubinden.

Interrupts

Das Interruptmodul verwaltet Systeminterrupts. 15 Interrupt-Anfrageleitungen sind verfügbar, wobei 4 davon für interne Funktionen verwendet werden (RS232 RDA, RS232 TBE, PS2, Millisekundentick). Die verbleibenden 11 sind benutzerdefinierbar. Interrupts auf der N.I.G.E. Machine sind priorisiert und blockend. Blockierende Interrupts werden vom Interruptmodul gespeichert und prioritätsbasierend abgearbeitet, sobald die aktuelle Interruptroutine abgeschlossen wurde. Alle Interrupts sind softwaremaskierbar über ein speicherabgebildetes 16Bit-Interrupt-Mask-Register.

Grafik

Die N.I.G.E. Machine hat einen eingebauten Videocontroller für die direkte Ausgabe eines VGA-Signals. Das VGA-Signal kann softwaremäßig zwischen den folgenden Pixelauflösungen umgeschaltet werden: 640x480, 800x600, 1024x768, 1920x1080. Die Ansteuerung erfolgt über ein speicherabgebildetes Zeichensystem. Die Zeichengröße ist benutzerdefinierbar zwischen 8 und 16 Pixel in Breite und Höhe. Schriftarten (Glyphenbitmaps) sind ebenfalls benutzerdefinierbar. Die Standardschriftart ist ein Klon vom Commodore PET. Die N.I.G.E. Machine kann auch Amiga Schriftart-Dateien in deren nativem Format laden!

Die Farbauflösung der Nexys-4-VGA-Ausgabe ist 12 Bit. Das ermöglicht 4096 verschiedene Farben, von welchen 256 gleichzeitig auf dem Bildschirm verwendet werden können. Der Grafikcontroller bietet die Möglichkeit, zwischen 0 und 15 leere Scanlines pro Textzeile einzufügen, um die Lesbarkeit zu verbessern.

SD-Karten-Schnittstelle

Die N.I.G.E. Machine beinhaltet eine microSD-Karten-Schnittstelle mit einem vereinfachten FAT-Dateisystem, welches 8+3-Dateinamen verwendet. Verzeichnisse werden ebenfalls unterstützt. Dateien auf der SD-Karte sind kompatibel zu einem PC.

Erweiterungsports

Das Nexys-4-Board enthält 5 Erweiterungsports, die *PMMODS*, welche den Anschluss fertiger oder selbstgemachter Zubehörteile ermöglichen.

Die Forth-Implementierung

Die Forthumgebung der N.I.G.E. Machine ist mit wenigen Einschränkungen ANSI-Forth-kompatibel. Viele CPU-Instruktionen sind Forth-Primitive:

```
NOP DROP DUP SWAP OVER NIP ROT >R R@ R> PSP@
CATCH RESET-SP THROW + -NEGATE 1+ 1- 2/ SUBX
ADDX = <> < > U< U> 0= 0<> 0< 0> ZERO AND OR
INVERT XOR 2* LSR XBYTE XWORD MULTS MULTU DIVS
DIVU FETCH.L STORE.L FETCH.W STORE.W FETCH.B
STORE.B ?DUP LOAD.B LOAD.W LOAD.L JMP JSL JSR
TRAP RETRAP RTI PAUSE RTS ,RTS BEQ BRA
```

Die Forth-Systemsoftware ist in Assembler geschrieben - `COMP` zum Beispiel. Ein Crossassembler[1] läuft auf der PC-Version von *VFX Forth* und generiert N.I.G.E. Maschinencode.

```
; COMP ( n1 n2 -- n )
\ n= -1 if n1<n2, +1 if n1>2, 0 if n1=n2
COMP.LF dc.l WITHIN.NF   ; link field
COMP.NF dc.b 4 128 +     ; name field
        dc.s COMP
COMP.SF dc.w COMP.Z COMP.CF del ; size field
COMP.CF over     ( n1 n2 n1)
        over     ( n1 n2 n1 n2)
        <        ( n1 n2 <flag)
        rot      ( n2 <flag n1)
        rot      ( <flag n1 n2)
        >        ( <flag flag)
        negate   ( <flag >flag)  +1 if n1>n2
COMP.Z  +,rts    ( n)            combine flags
```

Um die Forth-Systemsoftware klein zu halten, können zusätzliche Wörterbücher über Dateien auf der SD-Karte inkludiert werden.

Spezialfunktionen

Einige Funktionen der CPU der N.I.G.E. Machine, wie die Ausführungspipeline, die deterministische Ausführung und die schnelle Interruptantwort sind Funktionen, welche wohl von jedem Prozessor erwartet werden. Mit der Freiheit, welche ich mangels Pflichtenheft und Fristen genossen habe, war es mir möglich, die eine oder andere Spezialfunktion zu implementieren.

[1] Im Crossassembler sind (und ; gleichbedeutend. Beide kommentieren den Rest der Zeile aus.) hat keine spezielle Bedeutung. \ hingegen ist eine Anweisung und kein Kommentarzeichen.

Unausgerichteter Speicherzugriff auf 32-Bit-Datenpfad

Die erste Version der N.I.G.E. Machine, welche auf der EuroForth 2013 präsentiert wurde, hatte einen 8 Bit breiten Datenpfad, welcher die CPU mit dem SRAM-Systemspeicher verbunden hat. Lese- und Schreibzugriffe auf Langwörter und Wörter wurden als Multizyklus-Operationen implementiert, die in jedem Zyklus ein Byte adressiert haben. Zusammen mit dem Einzelbyteinstruktionsformat war dies ein relativ simples System.

Allerdings war dieses System nicht leistungsoptimiert. Beim Zählen der Assemblerinstruktionen in der Forth-Systemsoftware habe ich herausgefunden, dass `LOAD.W` (16-Bit-Wort laden, 18% aller Instruktionen) die häufigste Instruktion war. Aber genau diese Instruktion litt unter dem schmalen Datenpfad. In einem Gespräch mit Gerald Wodni und Bernd Paysan bei einem Abendessen dieser *EuroForth* stellten wir fest, dass die Verbreiterung des Datenpfades vermutlich die größte Auswirkung auf die Leistung der CPU haben würde, und daher die höchste Priorität haben sollte.

Es dauerte ein Weilchen herauszufinden, wie man das am besten angeht. Ich wollte keinen typischen Datenpfad verwenden, der nur ausgerichtete Adressierung für Wörter und Langwörter erlaubt. Dadurch ginge Flexibilität für den Programmierer verloren, und es würde außerdem Forth-Systemsoftware verkomplizieren. Ich kam zu dem Schluss, die Natur des Dual-Port-RAM-Blocks im Xilinx-FPGA einzusetzen, um einen Speichercontroller zu entwickeln, welcher einen 32Bit breiten, unausgerichteten Zugriff erlaubt - Abbildung 5.

Durch Lesen zweier benachbarter Langwörter kann der Speichercontroller diese vereinen und das unausgerichtete Langwort synthetisieren. Die komplizierten Details gibt es in meinem *EuroForth-2014-Paper*.

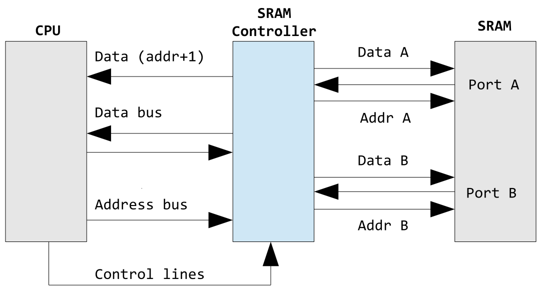


Abbildung 5: Der SRAM Speichercontroller

CATCH und THROW Maschineninstruktionen

Auch wenn lokale Variablen Forthprogrammierer polarisieren, finde ich sie hilfreich und unterstütze sie auf der N.I.G.E. Machine. Ich habe mir Gedanken darüber gemacht, wie man lokale Variablen in Hardware unterstützen könnte. Dabei bin ich auf eine Lösung gestoßen, welche es möglich macht, `CATCH` und `THROW` als eigene Maschineninstruktionen zu implementieren, etwas das ich mir am Anfang nicht erwartet hatte.

Der erste Schritt in diese Richtung war festzustellen, dass das Konzept des Returnstacks meiner Meinung nach logischer in zwei separater, verknüpfte Stacks aufgeteilt werden kann. Einen Unterprogrammstack und den originalen Returnstack. In diesem Modell wird bei jedem Aufruf eines Unterprogramms die Rücksprungadresse auf den Unterprogrammstack geschoben bzw. beim Verlassen des Unterprogramms wieder geholt. Währenddessen dient der Returnstack der Ablage von Objekten mittels `R>` und `>R` sowie `DO` und `LOOP` innerhalb eines individuellen Unterprogramms. Um dieses Modell konsistent mit der Verwendungsweise in Forth zu machen, bedarf es einiger Regeln, die Stacks zu verknüpfen. Wenn zum Beispiel der Unterprogrammstack ausgekellert[2] wird, muss der Returnstackpointer auf die Position zurückgesetzt werden, die er beim Eintreten in das Unterprogramm hatte. Zusätzlich muss der Returnstack eine Kopie der Rücksprungadresse enthalten und wenn diese ausgekellert wird, so muss auch das oberste Element auf dem Unterprogrammstack ausgekellert werden.

Außerdem bin ich zu dem Schluss gekommen, dass der Subroutinenstack viel breiter als die einzelne Zelle sein kann welche für die Rücksprungadresse benötigt wird. Eine weitere Zelle könnte einen Zeiger auf die Position des Returnstacks vor dem Aufruf des Unterprogramms haben, um die oben genannte Regel zu erlauben. Weitere 8 oder 16 Zellen könnten für lokale Variablen dieses Unterprogramms verwendet werden, und wenn diese auf fixe Adressen im Systemspeicher gelegt werden, ist es ein Einfaches, lokale Variablen in der Forth-Systemsoftware zu implementieren.

Ich habe festgestellt, dass das soeben beschriebene System ohne allzu viel Aufwand in Hardware auf der N.I.G.E. Machine implementiert werden kann. Allerdings war ich noch immer über die Ausnahmebehandlung besorgt. Von der Theorie über das Konzept geordneter Stacks geleitet, die durch spezielle Regeln miteinander verbunden sind, dachte ich darüber nach, ob es nicht möglich sei, die Ausnahmebehandlung über einen dritten Stack zu implementieren, einen Ausnahmestack, der an den Unterprogrammstack angebunden ist. Zurück am Reißbrett, habe ich mich nach einigem Getüftle selbst davon überzeugt, dass es keine Sonderfälle mehr gibt, welche die Logik zerstören könnten. Jetzt habe ich den Schatz gehoben: Die komplette ANSI-Forth-Funktionalität von `CATCH` und `THROW` könnte als einzelne Maschineninstruktion implementiert werden, welche einfach den Ausnahmestack in 2 oder 3 Zyklen ein- oder auskellert!

[2] Die Stack-Operationen push und pull werden im Deutschen schonmal als ein- und auskellern übersetzt. Quelle: Wikipedia - Stapelspeicher.

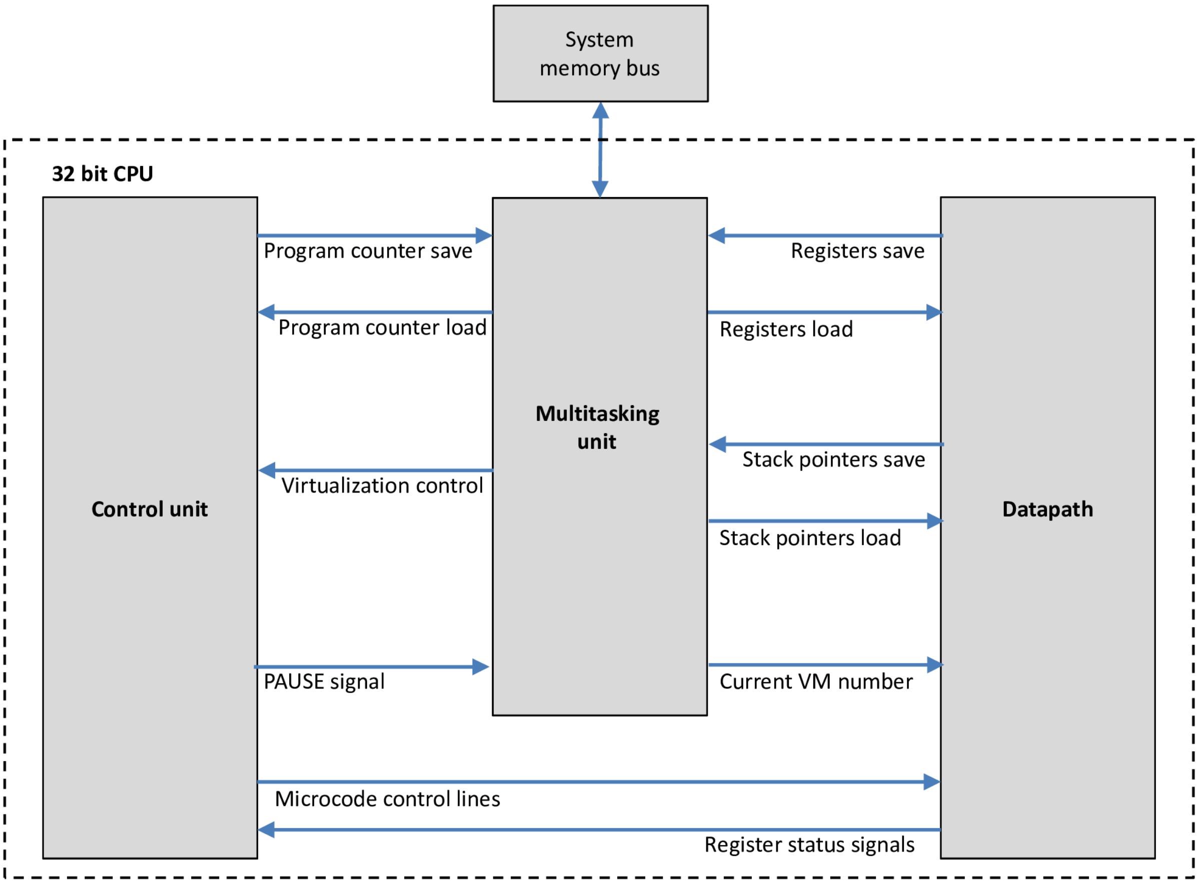


Abbildung 6: CPU Multitaskingmodul

Es gab einen weiteren Vorteil. Ich konnte den Ausnahmestack breiter gestalten und ihn nutzen, um die lokalen Variablen im Geltungsbereich einer Ausnahme abzulegen. Bei jedem Aufruf von `CATCH` wird dem neuen Set an Ausnahmevariablen die gleichen Werte wie in dem Niveau davor zugewiesen. Mit einem `THROW` werden die Variablen auf die Werte des vorhergegangenen Geltungsbereichs zurückgesetzt. Dies erlaubt die Definition von Variablen, welche im Falle einer Ausnahme garantiert einen bekannten Wert haben. In der Systemsoftware der N.I.G.E. Machine habe ich die Variable `BASE` und andere interne Variablen - wie `STIN` und `STOUT` - in dem Ausnahmeeintrag abgebildet. Anton Ertls HEX.-Problem illustriert, wie nützlich dies sein kann.

```
: hex.-helper
  hex \ die Variable BASE
      \ befindet sich auf dem Ausnahmestack
  u.
;

: hex.
  ['] hex.-helper catch throw
  \ BASE hat hier *garantiert*
  \ den originalen Wert
;
```

Dies ist eine kurze Zusammenfassung. Bei Interesse finden sich alle Details in meinem *EuroForth-2014-Paper*.

Hardware-unterstütztes Multitasking

Nach der EuroForth habe ich glücklicherweise angefangen, mit Ulrich Hoffmann zusammenzuarbeiten. Ulli und ich haben regelmäßig Ideen ausgetauscht und eine davon war das Thema Multitasking. Ulli hat mich schnell überzeugt, dass die N.I.G.E. Machine Multitasking unterstützen sollte. Meine erste Reaktion war: Tolle Idee, aber wie machen wir das in Hardware?! Nach Durchsicht einiger Literatur war mir klar, dass viele brillante Köpfe dieses

Thema untersucht haben, von Virtualisierung bis zum Barrelprozessor und vieles mehr. Es gab also viele Ideen, an die man anknüpfen konnte.

Wie gesagt, Vorgaben ändern sich mit der Zeit. Das meiste originale Forschungsmaterial kam aus den 1960ern und 1970ern, als Logikgatter in Hardware eine knappe und teure Ressource waren. Es schien mir auch, der Forschungsmainstream fokussiert sich auf die aktuellen Marktgiganten wie INTEL und ARM und weniger auf ausgefallene Alternativen. Könnte hier eine Lücke für innovatives Denken liegen?

Das von Ulli und mir entwickelte System fügt der CPU neben der Kontrolleinheit und dem Datenpfad eine neue Komponente hinzu, die *Multitaskingeinheit*. Die `PAUSE`-Maschineninstruktion friert den Status des aktuellen Tasks ein, und lädt den Status des nächsten Tasks in den Prozessor. Ein kompletter Taskwechsel benötigt 2 Taktzyklen, die gleiche Zeit wie ein bedingter oder unbedingter Sprung. In der Voreinstellung sind 32 Tasks verfügbar. Die Forth-Systemsoftware stellt *PolyFORTH*-ähnliche Wörter bereit, um Tasks zu starten, zu stoppen oder anderwärtig zu kontrollieren. Die Taskreihenfolge ist benutzerdefinierbar (Voreinstellung ist Rundlauf). Jeder Task hat einen privaten, 2KiB großen Speicherbereich, der parallel zum aktuellen Task gewechselt wird. Preemptives wird ebenso wie kooperatives Multitasking mit `PAUSE` unterstützt.

Der Hauptvorteil, Multitasking in Hardware zu implementieren (zumindest in der Theorie), ist die schnelle Ausführung, aber wir haben auch zusätzliche Funktionen wie "Virtuelle Interrupts" implementiert. Dabei kann ein Task einem anderen Task einen Unterprogrammnaufruf in den Exekutionspfad legen, welcher sich auf den Zieltask wie ein Interrupt auswirkt.

Die feinen Details, um das ganze System zu verstehen, finden sich in dem *EuroForth-2015-Paper* von Ulli und mir.

Wozu ist das nun gut?

Die Entwicklung der N.I.G.E. Machine hat in den letzten 5 Jahren viele Wochenenden beansprucht, und geht noch immer weiter. Was ist nun ihr Zweck? Zum einen natürlich der Spaß und die persönliche Befriedigung, welche so ein Projekt einher bringt. Aber ich habe auch ein klares Ziel im Auge.

Die N.I.G.E. Machine führt unterschiedliche Dinge in ein Paket zusammen. Zum Ersten ist sie ein System, welches komplett verstanden werden kann. Alle Designdateien sind Open Source und wurden hauptsächlich von einer Einzelperson entwickelt, also kann das gesamte System auch von einer Einzelperson verstanden werden. Das steht im krassen Kontrast zu System-on-a-Chip-Geräten und Prozessoren, wo das Datenblatt tausende Seiten hat und trotzdem Fragen unbeantwortet lässt. Gleichermaßen kann auch, im Gegensatz zu einer Linux-Distribution, der gesamte Quelltext der Systemsoftware einer N.I.G.E. Machine in ein paar Stunden durchgelesen werden.

Zweitens sind alle notwendigen Komponenten zum schnellen Applikationsentwickeln auf einer einzelnen Plattform vorhanden. Die N.I.G.E. Machine wird direkt mit einem Monitor und einem Keyboard verbunden, die Systemsoftware enthält schon den `EDITOR.F` als einfachen Texteditor, es gibt ein SD-Karten-Lesegerät, welches Standard-FAT32-Dateien lesen und schreiben kann, und wie die Leser dieses Magazins bereits wissen, Forth selbst ist beides: Eine interpretierte und eine kompilierte Sprache. Das Entwickeln von Software für die N.I.G.E. Machine benötigt also nicht die sonst gängigen mehrstufigen Toolchains.

Zuletzt ermöglicht das die Entwicklung von Applikationen nicht nur in Software, sondern auch in Hardware. Die N.I.G.E. Machine verwendet weniger als 10% der Registerresourcen auf dem Nexys-4-Artix-7-FPGA, wenn auch einen höheren Anteil von BLOCK-RAM, abhängig von der Konfiguration beim Synthetisieren. Es ist einfach, selbstgebaute Hardwaremodule in die N.I.G.E. Machine über Speicherabbildung und Interruptsignale einzuklinken. Das Nexys Board selbst hat mehrere Expansionsanschlüsse auf gepufferten FPGA Pins.

Unter Berücksichtigung dieser Faktoren denke ich, die N.I.G.E. Machine kann eine interessante Plattform für die Entwicklung verschiedenster Prototypen sein. Mein eigenes Ziel, sehr allgemein gesprochen, ist die Prototypenentwicklung von wissenschaftlicher Hardware.

Ich hoffe, ihr findet den Artikel interessant, und dass es mir vielleicht möglich ist, weitere Artikel mit euch zu teilen. Bis dahin findet sich für Interessierte das komplette Design der N.I.G.E. Machine auf Github. Ich plaudere gerne mit Enthusiasten per email - bitte in Englisch schreiben.

Mein Dank gilt Ulrich Hoffmann für seine Ideen und die Unterstützung beim Entwickeln der N.I.G.E. Machine, Michael Kalus für den Vorschlag zu diesem Artikel und Gerald Wodni für die Übersetzung ins Deutsche.

Ich wünsche euch Frohe Weihnachten und ein Frohes Neues Jahr!

Links

https://github.com/Anding/N.I.G.E.-Machine

andrew81244@outlook.com

Forth 200x–Treffen auf der EuroForth 2015

Anton Ertl

An den beiden Tagen vor der EuroForth 2015 in Bath fand wieder einmal das Treffen des Forth–200x–Komitees statt, das am nächsten Forth–Standard arbeitet. Diesmal standen zwei Erweiterungsvorschläge zur Abstimmung.

2s-Complement Wrap-Around Integers[1]

Dies legt die Zweierkomplementdarstellung als einzige Darstellung für negative ganze Zahlen fest (bisher waren theoretisch auch Einerkomplement und Sign–Magnitude Standard, wurden aber nicht genutzt). Außerdem soll bei einem Überlauf vieler ganzzahliger Operationen das Ergebnis mittels Modulo–Arithmetik berechnet werden, wie das die Forth–Systeme auch jetzt schon machen (bisher war Überlauf eine *ambiguous condition*, ein Standard–Programm durfte also keinen Überlauf haben). Die vorgeschlagene Änderung ermöglicht es, Hash–Funktionen und Kryptographie in Standard–Forth ohne Verrenkungen zu programmieren. Dieser Vorschlag wurde einstimmig angenommen.

1 chars = 1

Dies legt fest, dass ein Character[2] genau eine *address unit* (typischerweise ein Byte, auf wort–adressierten Maschinen eine Cell) groß ist. Dadurch wären Forth-Systeme mit *address units*< 8 Bits nicht mehr Standard (solche Systeme gab es aber ohnehin nicht), und auch nicht Forth–Systeme mit Bytes als *address units* und 16-Bit-Characters (da gab es nur experimentelle Forth-Systeme); dafür würden viele Programme Standard, in denen CHARS vergessen oder absichtlich weggelassen wurde. Die ursprüngliche Idee von CHARS war wohl, dass die Welt in Richtung 16–Bit–Zeichen gehen würde, was aber durch Unicode 2.0 (für das 16 Bits nicht mehr ausreichen) und die Erfindung von UTF–8 (sodass man auch mit 8–Bit–Einheiten Unicode darstellen kann) überholt war; in Forth–2012 spiegelt sich diese Entwicklung in der Einführung des xchar–Wordsets wider, die Möglichkeit zu 16–bit Characters auf byte–adressierten Maschinen würde mit dem vorliegenden Vorschlag entfernt.

Bei der Abstimmung zum Vorschlag *1 chars = 1* gab es drei Enthaltungen, weil diese Mitglieder sich noch nicht ausreichend klar über das Thema waren; da das nicht für einen Konsens spricht, haben wir den Vorschlag als vorerst nicht akzeptiert eingestuft und werden ihn nächstes Jahr erneut aufgreifen.

Sonstiges

Weiters wurden noch unfertige Vorschläge diskutiert, allen voran **Quotations** und **Recognizers**. Bei beiden wollte das Komitee auf mehr Erfahrungen warten. Bei den meisten älteren Vorschlägen gab es keine Fortschritte.

Ausblick

Neben diesen konkreten Vorschlägen drehte sich in diesem Treffen viel darum, wie wir den Standard künftig entwickeln wollen.

Gerald Wodni stellte die Web-Version [siehe unter Links] des Standards vor, die in Zukunft zu einer Web–2.0–Version werden soll, wo Benutzer Fragen zu einem Wort stellen und Beispiele dafür oder Änderungsvorschläge posten können.

Er stellte auch die überarbeitete Form des Paket–Repositories [siehe unter Links] vor, und wir gaben da noch einige Anregungen zum Format der Paket-Metadaten.

Wir wollen auch ausprobieren, an künftigen elektronischen Standardisierungstreffen Beobachter (oder genauer Zuhörer) teilnehmen zu lassen, die nicht Mitglieder des Komitees sind. Solche Treffen finden gelegentlich statt, um Dinge abzuschließen, die beim physischen Treffen angefangen wurden, aber wo zum Beispiel die genaue Formulierung nicht fertig war.

Das nächste Treffen findet wieder in den zwei Tagen vor der **EuroForth 2016 in Konstanz** statt. Ihr seid herzlich eingeladen, diesem Treffen beizuwohnen.

Links

`http://forth-standard.org`

`http://theforth.net`

[1] Das Zweierkomplement (auch 2-Komplement, Zweikomplement, B(inär)–Komplement, Basiskomplement, two's complement) ist eine Möglichkeit, negative Integer–Zahlen im Dualsystem darzustellen, ohne zusätzliche Zeichen wie + und − zu benötigen [wikipedia].

[2] Hierbei sind *Characters* im Sinne einer kleinen Speichereinheit gemeint, darstellbare Zeichen können seit dem xchar-Wordset von Forth-2012 mit mehreren Characters repräsentiert werden.

Minimal Forth Workbench

Ulrich Hoffmann

This package (Version 1.1.1 - 2015-10-31) provides a workbench implementation of ***Minimal Forth*** *as proposed by Paul Bennett and Peter Knaggs at EuroForth 2015 in Bath .*

Their aim is to define a Standard Forth subset suitable for educational purposes. For this they propose to cut down the number of Forth words initially explained to Forth newcomers and only stepwise introduce new concepts such as number output, compiling words, strings, file access, exceptions, ...

This package - the ***Minimal Forth Workbench*** *- allows to experiment with different sets of primitive (i.e. predefined) definitions in order to further elaborate on Paul's and Peter's ideas.*

Implementation

This implementation picks the appropriate words from a Standard (Forth 94 oder Forth 2012) system (the so called host system) and puts them in a wordlist of their own named minimal. In the end, this wordlist is defined to become the only wordlist in the search order and also the current wordlist. In essence this makes all other word definitions from the host system unavailable to the Minimal Forth Workbench.

The source code is seperated into several files:

1. The primitive words of Minimal Forth, to take from the host system. They are defined in the file *primitives.fs*

2. Words of Minimal Forth that can be defined in terms of primitives and other Minimal Forth words, i.e. "All that can be built from the primitives". They are defined in the file *secondaries.fs*

3. Several word set add-ons in the categories arithmetic, compiling words, control structures, defining words, doubles, error handling, file access, memory words, numeric output, stack operators, string words in files named after the category. Each of these will load independently of the others on top of Minimal Forth. They implement common words in that category by possibly requesting additional primitives from the host system and then building new words on top of them.

4. The Minimal Forth Workbench configuration in the file *prelude.fs*. Currently none of the add-ons is included. You can select any of them to get more functionality.

5. The implementation of the word *PRIMITIVE* and the overall setup: file *minimal.fs*.

6. A testbench in the file *testbench.fs* which compiles each of the add-ons on its own to check, they compile independently.

As currently setup, the words defined in *primitives* and *secondaries* are exactly the words proposed by Paul and Peter with 69 Minimal Forth words.

In addition the words *INCLUDE, WORDS, PRIMITIVE,* and *BYE* that we consider more development environment than language are available. They are not included in the primitives and word counts.

Your are invited to experiment: remove or add primitives or modífy, add, remove definitions in *secondaries*. Extend and modify add-ons. Define new add-ons. Please, share your findings.

Installation

To use the Minimal Forth workbench just *INCLUDE minimal.fs* on any standard system. This makes the Minimal Forth definitions and only them available. After loading, all defined words still have their standard meaning - however Minimal Forth is no standard system itself as it does not provide all definitions from the *CORE word set*.

Example usage

```
$ sf minimal.fs

Minimal Forth Workbench: 48 primitives, 69
words

ok

words ALIGNED CELL+ CHAR+ ROT 2/ LSHIFT XOR
OR > = 0= TRUE FALSE MOD 2* / * + VARIABLE
CONSTANT DUP primitive WORDS INCLUDE bye \ .S
( CR KEY? EMIT KEY DOES> ; CREATE : EXECUTE
J LOOP UNTIL AGAIN BEGIN ELSE ' I DO REPEAT
WHILE THEN IF R> OVER DROP R@ >R SWAP RSHIFT
INVERT AND < - */MOD CHARS CALIGNED CALIGN C@
C, C! CELLS ALIGN @ , !

48 primitives, 69 words ok
```

Bug Reports

Please send bug reports, improvements and suggestions to Ulrich Hoffmann <uho@xlerb.de>

Conformance

This program conforms to Forth-94 and Forth-2012. It has been verified to load succesfully on SwiftForth, iForth and gForth.

May the Forth be with you!

Links

Download current version from:

TheForth.Net (http://theforth.net)

Listings

Minimal

```
\ Minimal Forth Workbench: main file   uh 2015-10-05

: tick
  ( <spaces>name<spaces> -- comp-xt exec-xt flag )
    STATE @ >R
    ] >IN @ >R  BL WORD FIND
    IF R> >IN !
       POSTPONE [  BL WORD FIND
    ELSE R> DROP
       DROP 0 0 false
    THEN
    R> IF ] ELSE POSTPONE [ THEN ;

: immediate-alias
  ( comp-xt exec-xt <spaces>name<spaces> -- )
    CREATE , , IMMEDIATE
    DOES> STATE @ IF CELL+ THEN  @ EXECUTE ;

: non-immediate-alias
  ( comp-xt exec-xt <spaces>name<spaces> -- )
    CREATE , , IMMEDIATE
    DOES> STATE @ IF CELL+ @ COMPILE,
                  ELSE  @ EXECUTE THEN ;

VARIABLE #primitives  0 #primitives !
VARIABLE #words 0 #words !

: another-primitive ( -- )
  1 #primitives +!  1 #words +! ;

wordlist Constant minimal

: primitive
  ( <space>ccc<space> -- )
    get-order
      minimal 1 set-order
      >IN @ >R  tick R> >IN ! NIP NIP
      0= IF
        forth-wordlist 1 set-order
        another-primitive
        >IN @ >R tick  R> >IN !  DUP 0= Abort" ?"
        0< IF non-immediate-alias
           ELSE immediate-alias THEN
      ELSE
        CR BL WORD COUNT TYPE ."  is already defined."
      THEN
    set-order ;

minimal set-current

include primitives.fs    \ Minimal Forth primitives

get-order ' set-order
minimal 1 set-order

include prelude.fs  \ System configuration

execute
cr .( Minimal Forth Workbench: )
#primitives @ . .( primitives, )
#words @ . .( words) cr
minimal 1 set-order
```

Prelude

```
\ Minimal Forth Workbench - system configuration

\ primitives

\ Minimal Forth secondaries, defined in terms of others
include secondaries.fs
```

Primitives

```
\ Minimal Forth Workbench: definitions of primitives
\ uh 2015-10-10

primitive !
primitive ,
primitive @
primitive ALIGN
\ primitive ALIGNED
\ primitive CELL+
primitive CELLS
primitive C!
primitive C,
primitive C@
: CALIGN ;   another-primitive
: CALIGNED ; another-primitive
\ primitive CHAR+
primitive CHARS

\ primitive +
\ primitive *
\ primitive 2*
primitive */MOD
primitive -
\ primitive /
\ primitive 2/
\ primitive MOD

\ primitive 0=
primitive <
primitive AND
primitive INVERT
\ primitive TRUE
\ primitive LSHIFT
\ primitive =
\ primitive >
\ primitive OR
\ primitive XOR
\ primitive FALSE
primitive RSHIFT

\ primitive DUP
primitive SWAP
primitive >R
primitive R@
primitive DROP
primitive OVER
primitive R>

primitive IF
primitive THEN
primitive WHILE
primitive REPEAT
primitive DO
primitive I
primitive '
primitive ELSE
primitive BEGIN
primitive AGAIN
primitive UNTIL
primitive LOOP
primitive J
primitive EXECUTE
```

```
\ primitive :
: :   :    1 #words +! ; another-primitive
\ primitive CONSTANT
\ primitive CREATE
: CREATE   CREATE    1 #words +! ; another-primitive
primitive ;
\ primitive VARIABLE
primitive DOES>
primitive KEY
primitive EMIT
primitive KEY?
primitive CR
primitive (
primitive .S
primitive \
: bye bye ;
: INCLUDE include ;
\ primitive WORDS
: WORDS
   WORDS
   CR #primitives @ . ." primitives, "
   #words @ . ." words" ;
\ : order  order ;
\ primitive EXIT
: primitive    primitive ;
```

Secondaries

```
\ Minimal Forth Workbench: Secondary words
\  - Minimimal Forth words defined in terms of others
\ uh 2015-10-15
: DUP ( X1 -- X1 X1 )  >R R@ R> ;
: CONSTANT ( X "<Spaces>Name" -- )  CREATE , DOES> @ ;
: VARIABLE ( "<Spaces>Name" -- )  CREATE 0 , ;
: + ( N N -- N )  0 SWAP - - ;
: * ( N N -- N )  1 */MOD SWAP DROP ;
: / ( N N -- N )  1 SWAP */MOD SWAP DROP ;
: 2* ( N N -- N )  DUP + ;
: MOD ( N N -- N )  1 SWAP */MOD DROP ;
0 CONSTANT FALSE
FALSE INVERT CONSTANT TRUE
: 0= ( x -- f )  IF FALSE ELSE TRUE THEN ;
: = ( N N -- F )  - 0= ;
: > ( N N -- F )  SWAP < ;
: OR ( X X -- X )
  INVERT SWAP INVERT AND INVERT ; ( do Morgan )
: XOR ( X X -- X )
  OVER OVER INVERT AND >R SWAP INVERT AND R> OR ;
: LSHIFT ( X1 U -- X2 )
  BEGIN DUP WHILE >R  2*  R> 1 - REPEAT DROP ;
: 2/ ( X1 -- X2 )  1 RSHIFT ;
: ROT ( X1 X2 X3 -- X2 X3 X1 )  >R  SWAP R> SWAP ;
: CHAR+ ( c-addr1 -- c-addr2 )  1 CHARS + ;
: CELL+ ( addr1 -- addr2 )  1 CELLS + ;
: ALIGNED ( addr -- a-addr )
  CELL+ 1 -   1 CELLS 1 - INVERT  AND ;
```

Permanente Tabellen im AmForth-Flash ablegen

Erich Wälde

Wie legt man vorberechnete Werte in einer Tabelle im FLASH-Speicher ab, wenn man ein AmForth in der MCU hat? Vor dieser Aufgabe steht man des Öfteren. In einem Fall waren das gepackte Werte, die aus einer speziellen, deklarativen Syntax generiert wurden [1]. In einem aktuellen Fall sind das Einsprungadressen (XTs) von kleinen Programmschnipseln, welche durch ihre Lage in der Tabelle ausgewählt und aufgerufen werden, also die altbekannte Sprungtabelle [2]. Das folgende Rezept *wurde urspünglich auf der Webseite von AmForth veröffentlicht.*

So nicht

Nachdem ich gerade in einem anderen Zusammenhang auf *Quotations* aufmerksam geworden war, schrieb ich unbekümmert

```
create Table \ WRONG!
:noname 1 VarA +! ; ,
:noname 2 VarA +! ; ,
...
```

Das ist nicht nur naiv, sondern schlicht falsch: Damit vermischte ich die kompilierten Programmschnipsel mit den Einsprungadressen, die ich aufheben wollte. Die Tabelle enthält zwar die Einsprungadressen, **aber nicht an den vorgesehenen Stellen**. Nachdem ich das verstanden hatte, entschied ich, die XTs zunächst in einer Tabelle im RAM abzulegen und am Ende die fertige Tabelle ins FLASH zu kopieren. Zusätzlich konnte ich so bequem die Tabelle mit `' :noname` initialisieren, denn einige Einträge der Tabelle würden *leer* bleiben. Alternativ hätte ich die XTs auch auf den Stapel legen können. Aber bei mehr als 50 Einträgen erschien mir das falsch, von der umgekehrten Reihenfolge im Quelltext mal ganz abgesehen.

Wie dann?

Zunächst definiere ich die Variablen, die in den Programmschnipseln benötigt werden, dann die Anzahl der Einträge, die in der Tabelle Platz finden sollen, und schließlich die Tabelle im RAM. `T.init` füllt, wenn aufgerufen, die Tabelle im RAM mit der Einsprungadresse von `noop`. Ein Aufruf von `fill` hilft hier nicht, weil `fill` nur mit Bytes operiert und nicht mit Zellen.

```
\ variables needed by the code snippets
\ should go BEFORE variable T.ram
variable SomeVar
variable OtherVar

#10 constant T.len
variable T.ram T.len cells allot
```

```
: T.init
  ['] noop T.len 0
  do dup T.ram i cells + ! loop
  drop ;
```

Der von `T.ram` belegte Platz kann nach dem Umkopieren wieder freigegeben werden, auch wenn das Wort `T.ram` weiterhin existiert.

Während die Programmschnipsel[1] mit `:noname ... ;` kompiliert werden, sollen die generierten XTs an eine bestimmte Stelle in `T.ram` eingefügt werden. `>T.ram` dient dazu, macht den Quelltext lesbarer und überprüft auch, dass die gewünschte Stelle innerhalb der Tabelle liegt.

```
: >T.ram ( xt idx -- )
  dup 0 T.len within if
    cells T.ram + !
  else drop \ or throw
  then ;
```

Die Funktion `>ftable` kopiert die Werte dann vom RAM in eine neue Tabelle im FLASH. `>ftable` erwartet die Anzahl der Tabelleneinträge und die Adresse der Tabelle im RAM auf dem Stack und es verwendet den im Quelltext nachfolgenden Namen als Namen für die neu definierte Tabelle im FLASH.

```
: >ftable ( srcaddr len -- ) ( ccc.name )
  create ( consumes ccc.name )
  ( len ) 0 do
     ( srcaddr ) dup i cells + @ ,
  loop
  ( srcaddr ) drop does> \ fixme: needed??? ;
```

Solchermaßen ausgerüstet können wir nun die namenlosen Programmschnipsel generieren, die zugehörigen Einsprungadressen in die RAM-Tabelle schreiben und dann ins Flash kopieren.

```
T.init

\ create an anonymous function...
:noname #1 SomeVar +! ;

\ ... and store it into field #3
#3 >T.ram

\ another one stored to field #4
:noname #8 SomeVar +! #1 OtherVar ! ;
#4 >T.ram

\  ...
```

Die namenlosen Programmschnipsel dürfen verschieden lang sein. Die Reihenfolge, in der die XTs nach `T.ram` geschrieben werden, ist unerheblich. Es müssen auch nicht alle Felder von `T.ram` beschrieben werden, weil sie schon mit `noop` initialisiert wurden. Wenn die RAM-Tabelle `T.ram` auf diese Weise vorbereitet worden ist, kann sie umkopiert werden.

```
T.ram T.len >ftable T.flash
```

Die neue, permanent abgelegte Tabelle heißt in diesem Beispiel ab jetzt `T.flash` . Den jetzt nicht mehr benötigten Platz im RAM geben wir frei.

```
T.ram to here
```

Vorausgesetzt, wir haben beim Erstellen der Programmschnipsel keine neuen Variablen angelegt. Die würden nach der RAM-Tabelle abgelegt und möglicherweise später *überschrieben* werden. Das kommt nur dann vor, wenn nach dem Zurücksetzen von `here` so viele neue Variablen angelegt werden, dass der Platz, den die RAM Tabelle belegt hatte, vollständig neu vergeben wird. Dann kann es dazu kommen, dass Speicherplatz im RAM doppelt vergeben wurde, was zu schwer nachvollziehbaren Fehlern führen kann[2].

Und so läuft es dann...

Die Einträge in `T.flash` können wir beispielsweise so nutzen:

```
: T.run ( index -- )
  dup 0 T.len within if
    ( index ) T.flash + @i execute
  else drop \ or throw then ;
```

Viel Spaß!

Referenzen

[1] VD 2012-03, S. 25ff: Morsen 5, eine deklarative Version.

[2] http://amforth.sourceforge.net/TG/recipes/Jump-Tables.html

[1] Das sind die *anonymous functions.*

[2] Mit anderen Worten: das Anlegen von Variablen *nach* dem Erstellen und *vor* dem Freigeben der RAM-Tabelle ist *nicht* sofort ein Problem, sondern vielleicht später, wenn man die Akrobatik längst vergessen hat. (ew)

Ein- & Ausgabe für IEEE-32-Bit-Float

Rafael Deliano

Immer mehr größere Controller haben FPUs, so dass einem dieses Zahlenformat auch bei embedded Anwendungen nun häufiger begegnet.

Das Datenwort besteht aus drei Komponenten (Abb. 1), von denen das Vorzeichen als reines Flagbit die wenigsten Probleme macht.

Der 8-Bit-Exponent ist ursprünglich vorzeichenlos, weil er einen Offset von $7F hat, den man entfernen muss. Damit ergibt sich ein Wert -127 ... 0 ... +128, um den man den gedachten Dezimalpunkt der Mantisse skaliert. Ein positiver Wert bedeutet, nach links, ein negativer, nach rechts schieben.

Er befindet sich ursprünglich links neben der Mantisse. Die allerdings dann noch um ein Bit mit festem Wert 1 erweitert werden muss, das im Datenwort nicht explizit dargestellt ist. Gegenüber einem Integer hat der Dezimalpunkt dieser Binärzahl bereits einen Offset von 23 Bit, ein Wert, der zusätzlich in den Exponent eingerechnet werden muss. Den man deshalb besser als 16-Bit-Zahl verarbeitet. Die extrahierte Mantisse belegt ein 32-Bit-Datenwort.

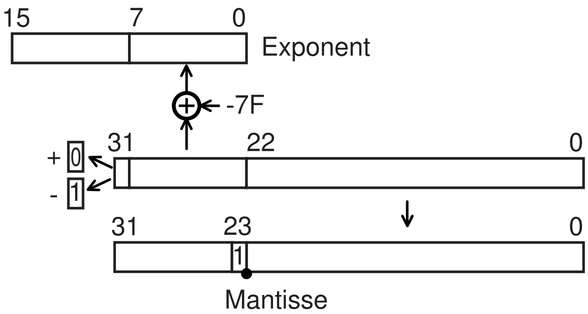


Abbildung 1: IEEE-32-Bit-Format

Float hat im Vergleich zu 32-Bit-Festkommazahlen zwar weitere Dynamik, aber geringere Genauigkeit. In Controllern werden in Float oft nur Zahlen mit recht eingeschränkter Dyamik dargestellt. Da ja Sensoren meist kaum mehr als 16-Bit-Auflösung haben. Dann sind vereinfachte Routinen angemessen.

Umwandlung-32-Bit

Wenn man in Float umskalieren kann, so dass der Wert für einen 32-Bit-Integer mit Vorzeichen passt, ist die Ein/Ausgabe besonders simpel. Allerdings ergibt sich beim Befehl

```
F->SD \ ( FD1 --- D1 )
```

bei Zahlen größer als 00FFFFFF spürbar reduzierte Genauigkeit, weil die 24-Bit-Mantisse dann ja nicht mehr ausreicht:

```
---| FFFF 7FFF
--- FFFF 7FFF | 2DUP SD.
+2147483647
--- FFFF 7FFF | SD->F F.
+2139095168.000000000
---|
```

Beim entsprechenden Befehl

```
SD->F \ ( D1 --- FD1 )
```

wurde für die „too big“-Zahlen Sättigungsverhalten auf 7FFFFFFF implementiert.

Drucken

Für die simple Variante des Befehls

```
F. \ ( FD1 --- )
```

wird der 32-Bit-Druckbefehl `D.` von nanoFORTH zweimal verwendet. Erst um eine Zahl von bis zu 10 Stellen vor und dann eine nach dem Dezimalpunkt auszugeben:

```
---|
B% 0011001100110011
B% 0100000110010011
--- 3333 4193 | F.
+0000000018.399999619
---|
```

Für die Zahl vor dem Dezimalpunkt ist die Mantisse rechtsbündig zu skalieren (Abb. 2). Ob man links oder rechts schiebt, hängt vom Exponenten ab.

Abbildung 2: Integer zur Ausgabe der Vorkommaziffern

Verschiebt man nach links, muss man überwachen, ob Überlauf eintritt. Hier möglich, da diese Festkomma-Druckroutine nicht den gesamten möglichen Dynamikbereich der Float darstellen kann. Sie druckt gegebenenfalls too big. Das erschlägt auch die Sonderfälle Infinity und NaN[1].

Für Ziffern nach dem Komma wird linksbündig skaliert (Abb. 3). Die Umformung auf Integer kann dann durch den 64/32-Bit-Befehl `UD*/` vorgenommen werden. Der Skalierungsfaktor von ca. 0,233... ist nicht exakt, die letzten Nachkommastellen passen nur ungefähr. Es werden auch nur 9 Stellen nach dem Komma ausgegeben.

[1] Not a Number

Die Darstellungsweise ist besonders anschaulich, wenn man Zahlenkolonnen ausdruckt. Aber sie deckt den enormen Dynamikbereich von Float nicht ab.

Abbildung 3: Integer zur Ausgabe der Nachkommaziffern

scientific notation

Das tut nur die vom Taschenrechner gewohnte Exponentialschreibweise:

`+1,234E-5`

Sie entspricht Float in seiner Aufteilung in Mantisse und Exponent. Allerdings basiert die Darstellung auf 10^x, während die IEEE-Float 2^x verwendet. Simple Umrechnung zwischen beiden Formaten ist nicht möglich.

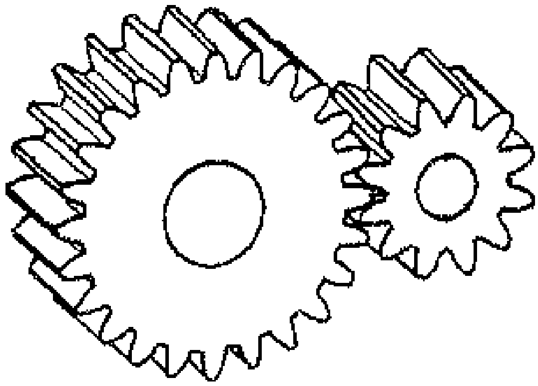


Abbildung 4: Zahnräder

Man kann sich den nötigen Prozess aber als Drehung von zwei Zahnrädern, die unterschiedlichen Durchmesser haben, veranschaulichen (Abb. 4). Dabei kann sich das „binäre“ Zahnrad nur um Schritte `*2` und `/2` bewegen, dafür können vorzeichenlose 32-Bit-Shiftbefehle `1DSHIFT>` und `1<DSHIFT` verwendet werden. Für die Schritte `*10` und `/10` des dezimalen Zahnrads könnte man `*10` auch durch Shifts darstellen (Abb. 5). Aber wegen der Division `/10` wird hier der 32/64-Bit-Befehl verwendet (Abb. 6).

```
: *10 \ ( UD1 --- UD2 )
  2DUP
  1<DSHIFT 1<DSHIFT 1<DSHIFT
  2OVER D+ D+ ;
```

Abbildung 5: *10 durch Shifts implementiert

```
: UD*/ \ ( UD1 UD2 UD3 \ --- UD4 ) ... ;
\ UD4=(UD1*UD2)/UD3
HEX
: /10 1 0 A 0 UD*/ ;
: *10 A 0 1 0 UD*/ ;
```

Abbildung 6: *10 und /10 basierend auf UD*/

Die Variablen `2EXP` und `10EXP` enthalten die beiden Exponenten, also ca. den Drehwinkel des jeweiligen Zahnrads. Die 24-Bit-Mantisse kann in einem 32-Bit-Register so angeordnet werden, dass links und rechts 4 Bit frei sind (Abb. 7). Durch Bewegung der Daten in der Mantisse kann man einen der Exponenten auf null bringen und damit seinen Inhalt in den anderen Exponenten überführen.

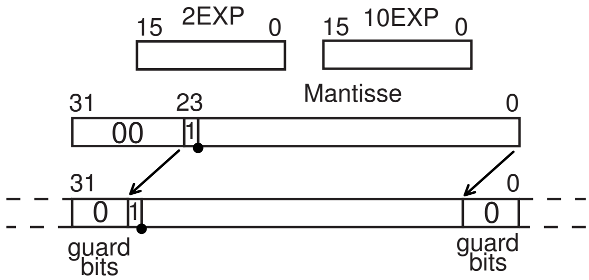


Abbildung 7: guard bits

Eingabe

Normalerweise muss man drei Festkommazahlen eingeben, um die Zahl darzustellen: `X,YeZ`. Wobei typisch so normiert wird, dass `X` eine einzelne Dezimalziffer wird. Man kann sich den Aufwand reduzieren, indem man festlegt, dass `X` null sein muss, also: `0,YeZ`. Sowie der Anwender die Mantisse mit fester Länge als 9 Ziffern eingibt, und als Trennung `SPACE` einfügt:

```
+0, 123456789 e- 02 :F
```

Das Format ist sicher gewöhnungsbedürftig, weil es nicht genau dem Taschenrechner entspricht. Vorteil ist aber, dass die Eingabe leicht auf den Eingabebefehl `D:` für 32-Bit-Festkomma aufsetzen kann (Abb. 8).

Bei der Verarbeitung wird nun `10EXP` auf null reduziert und dann die Mantisse auf das IEEE32-Format skaliert (Abb. 9). Die Konstante 9 ist auf die Eingabe in 9 Digits Nachkomma abgestimmt.

```
: +0,  0 D: ; \ ( 0000 UD1 )
: -0, -1 D: ; \ ( FFFF UD1 )
: e-  -1 D: DROP ; \ ( FFFF UN1 )
: e+   0 D: DROP ; \ ( 0000 UN1 )
: :F   ... ; \ ( flag UD1 flag UN1 UDfloat )
```

Abbildung 8: Eingabe

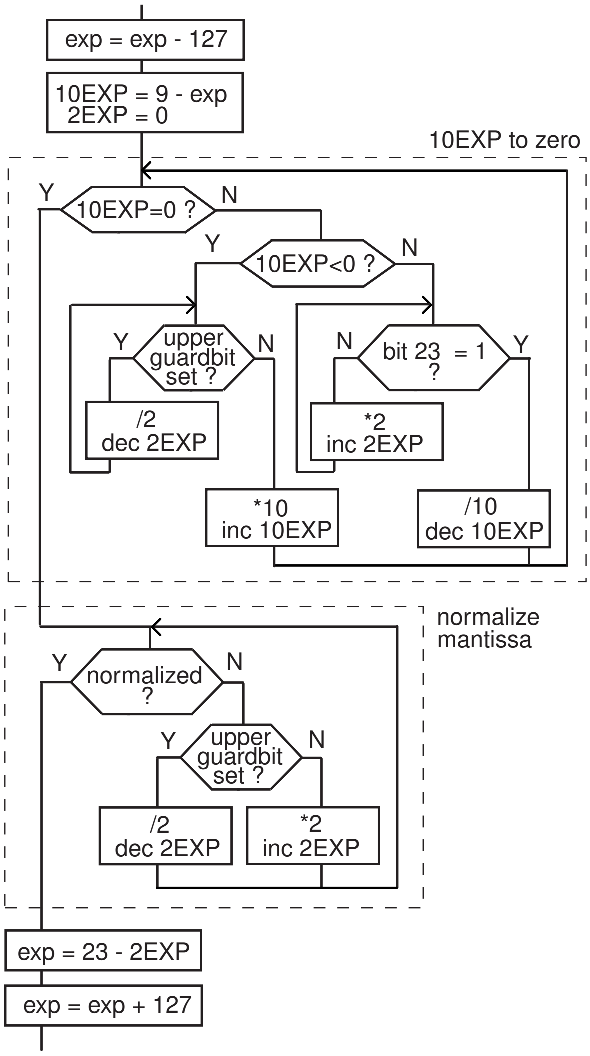


Abbildung 9: Flussdiagramm Eingabe

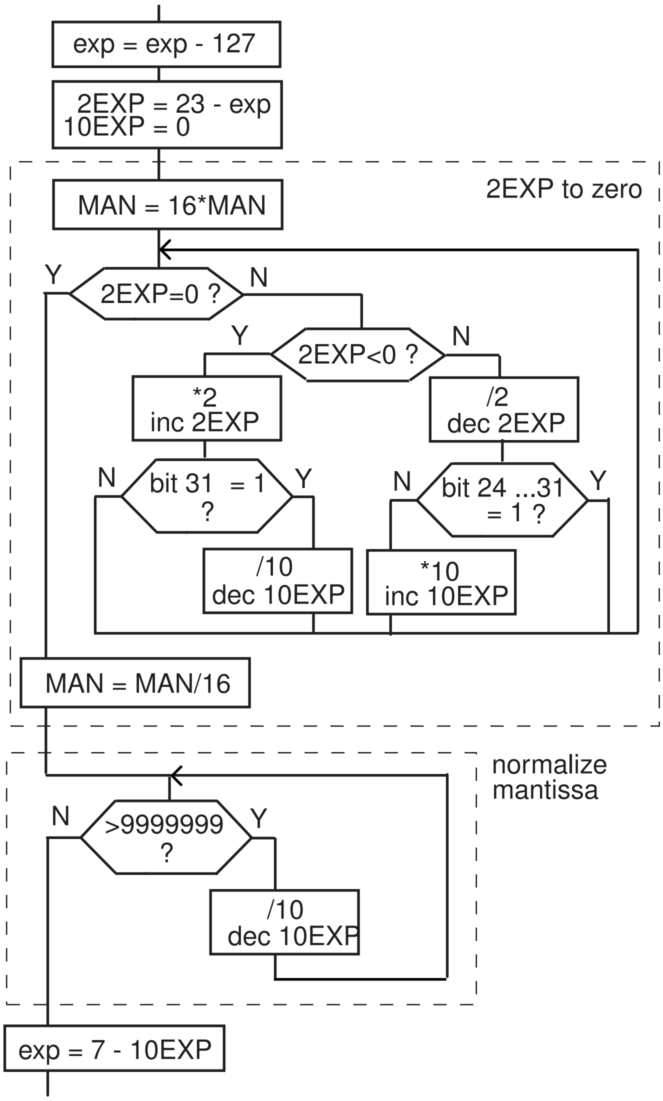


Abbildung 10: Flussdiagramm Ausgabe

Ausgabe

Hier wird erst `2EXP` auf null reduziert und dann die Mantisse auf das 7-Digits-Nachkomma skaliert (Abb. 10). Dadurch ergibt sich hier die Konstante „7“. Beide Routinen erzielen keine perfekte Genauigkeit. Aber für Controller typisch ausreichend.

Die Darstellung bezieht sich hier auf ein 16-Bit-Forth. Controller mit FPU sind 32-Bit-CPUs. Entsprechendes Forth erreicht durch die verdoppelte Wortlänge dann auch höhere Genauigkeit.

Bei 8-Bit-Controllern ist es manchmal zweckmäßig, eine Motorola MC68882 FPU nachzurüsten. Die enthält auch genaue Funktionen zur Umwandlung in ASCII.

Referenzen

[1] Jorke, Lampe, Wengel; Arithmetische Algorithmen der Mikrorechentechnik, VEB Technik 1989

Links

https://de.wikipedia.org/wiki/IEEE_754

Listing

```
\ Test data

\ 18.39999961853
  B% 0011001100110011
  B% 0100000110010011 DCONSTANT 18.39
\ +0
  B% 0000000000000000
  B% 0000000000000000 DCONSTANT +0.0
\ 2^-149  smallest positive value
  B% 0000000000000001
  B% 0000000000000000 DCONSTANT +2^-149
\ +2^-127
  B% 0000000000000000
  B% 0000000001000000 DCONSTANT +2^-127
\ +2^-126
```

```
  B% 0000000000000000
  B% 0000000010000000 DCONSTANT +2^-126
\ +2
  B% 0000000000000000
  B% 1000000001000000 DCONSTANT +2.0
\ +6.5
  B% 0000000000000000
  B% 0100000011010000 DCONSTANT +6.5
\ most positive finite value
  B% 1111111111111111
  B% 0111111101111111 DCONSTANT +MAX
\ + infinity
  B% 0000000000000000
  B% 0111111110000000 DCONSTANT +INF
  B% 0000000000000000
  B% 0111111110000010 DCONSTANT +NaN
\ -0
  B% 0000000000000000
  B% 1000000000000000 DCONSTANT -0.0
\ -2
  B% 0000000000000000
  B% 1100000000000000 DCONSTANT -2.0
\ -6.5
  B% 0000000000000000
  B% 1100000011010000 DCONSTANT -6.5
\ -MAX
  B% 1111111111111111
  B% 1111111101111111 DCONSTANT -MAX
\ - infinity
  B% 0000000000000000
  B% 1111111110000000 DCONSTANT -INF
  B% 0001001010101010
  B% 1111111110010001 DCONSTANT -NaN
  B% 0000000000000000
  B% 0100000110011100 DCONSTANT 19.5
  B% 0000000000000000
  B% 1100000010100000 DCONSTANT -5.0
  B% 0000000000000000
  B% 1100000011000000 DCONSTANT -6.0
  B% 0000000000000000
  B% 0011111111000000 DCONSTANT 1.5
  B% 0000000000000000
  B% 0100001111000000 DCONSTANT 384.0
\ 1/16
  B% 0000000000000000
  B% 0011110110000000 DCONSTANT 1/16

1DSHIFT> \ ( UD1 --- UD2 ) LSR
1<DSHIFT \ ( UD1 --- UD2 ) LSL
UD*/     \ ( UD1 UD2 UD3  --- UD4 )
         \ UD4 = ( UD1 * UD2 ) / UD3
         \ 64 bit intermediate result
(D.)     \ ( D1 #  ---  ) D1 = 0 ... 7FFFFFFF
         \ # = 2 ... A : number of digits printed

\ SD->F  F->SD

\ ------------------------------------------
\ include 1DSHIFT> 1<DSHIFT

: SD->F           \ ( D1 --- FD1 )
    \ 32 bit signed integer to IEEE32 float
0 ROT ROT         \   --- sign D1 )
DUP 8000 AND
IF DNEGATE ROT 8000 OR ROT ROT
THEN
2DUP OR
IF.               \   --- sign UD1 )
  DUP FF00 AND
  IF              \ shift right
    B 1 DO
    1DSHIFT>
    DUP FF00 AND LNOT IF 007F AND I LEAVE THEN
    LOOP
  96 +  \   --- sign UD1 eee )
  ELSE.
  DUP 0080 AND
   IF               \   --- sign UD1 )
     007F AND 96
   ELSE
     D% 24 1 DO
     1<DSHIFT
     DUP 0080 AND IF 007F AND I LEAVE THEN
     LOOP
     96 SWAP -    \ 96 = D% 23 + H% 7F
   THEN
  THEN              \   --- sign UD1 eee )
  8<SHIFT 1SHIFT> OR  \   --- sign UD2 )
  ROT OR            \   --- FD1 )
ELSE
ROT DROP   \ Zero
THEN
;

: F->SD                    \ ( FD1 --- D1 )
    \ IEEE32 float to 32 bit signed integer
DUP 8000 AND ROT ROT       \ --- sign FD1 )
DUP 1<SHIFT 8SHIFT>        \ --- sign FD1  eee )
SWAP 007F AND 0080 OR SWAP \ --- sign UD1' eee  )
96 -                       \ --- sign UD1' eee' )
7F D% 23 -
DUP
IF DUP 8000 AND            \ --- sign eee' UD1' )
  IF NEGATE
     1 DO 1DSHIFT> LOOP
  ELSE
     1 DO 1<DSHIFT
     DUP 8000 AND IF 2DROP FFFF 7FFF LEAVE THEN
     LOOP
  THEN
ELSE
DROP     \ zero
THEN
ROT IF DNEGATE THEN
;


\ F. limited dynamic version

\ ------------------------------------------
\ include (D.) 1DSHIFT> 1<DSHIFT  UD*/

: F.      \ ( FD1 ---  ) print IEEE 32 bit float
DUP 8000 AND IF 2D ELSE 2B THEN EMIT
DUP 1<SHIFT 8SHIFT> \ --- FD1 eee )
SWAP 007F AND 0080 OR SWAP  \ --- UD1' eee  )
7F - HOPP HOPP             \ --- UD1' eee' UD1' )
HOPP FFE9 +    \ D% -23
DUP
IF DUP 8000 AND            \ --- UD1' eee' UD1' )
  IF NEGATE
     1 DO 1DSHIFT> LOOP 1
  ELSE 1 ROT ROT
     1 DO
        1<DSHIFT DUP 8000 AND IF
           2DROP DROP 0 0 0 LEAVE
        THEN
     LOOP ROT
  THEN
ELSE
DROP 1    \ = 0
THEN      \ --- UD1' eee' UD1' flag )

IF.
    A (D.)                      \ D. without SPACE
    2E EMIT                     \ "."
                                \ ( UD1' eee' --- )
    9 +
    DUP
    IF DUP 8000 AND
      IF NEGATE
        1 DO 1DSHIFT> LOOP
```

```
      ELSE
        1 DO 1<DSHIFT LOOP
      THEN
    ELSE
      DROP     \ = 0
    THEN
                                    \ ( UD1 ---  )
    B6B4 0DE0   CA00 3B9A   UD*/
    9 (D.)

ELSE 2DROP DROP 2DROP ." too big"
THEN SPACE
;


\ :F  input scien. notation

\ -----------------------------------------
\ include 1DSHIFT> 1<DSHIFT UD*/


2 ZVARIABLE  2EXP
2 ZVARIABLE 10EXP

: /10  1 0 A 0 UD*/ ;
: *10  A 0 1 0 UD*/ ;

\ ------------------------------

\     1...9 digits 1...2 digits
\ +0, 123456789 e- 02
\ -0, 123456789 e+ 03

: +0,  0 D: ;       \ (  --- 0000 UD1 )
: +0. +0,   ;
: -0, -1 D: ;       \ (  --- FFFF UD1 )
: -0. -0,   ;
: e-  -1 D: DROP ;  \ (  --- FFFF UN1 )
: e+   0 D: DROP ;  \ (  --- 0000 UN1 )

: B-STEP>             \ ( UD1 --- UD2 )
BEGIN
  DUP F000 AND  \ test if any upper guardbit set
  IF   1DSHIFT> -1 2EXP +! 0
  ELSE 1
  THEN
UNTIL ;

: B-<STEP             \ ( UD1 --- UD2 )
BEGIN
  DUP 0800 AND        \ test if normalized
  IF   1
  ELSE 1<DSHIFT  1 2EXP +! 0
  THEN
UNTIL ;

: :F     \ ( flag UD1 flag UN1 --- UDfloat )
SWAP IF NEGATE THEN
9        \ mantissa is input as fixed 9 digits
SWAP - 10EXP ! 0 2EXP !
                 \ reduce 10EXP to zero
BEGIN
  10EXP @ DUP
  IF             \ 10EXP not zero
    8000 AND
    IF   B-STEP>   *10    1 10EXP +!
    ELSE B-<STEP   /10   -1 10EXP +!
    THEN 0
  ELSE LNOT            \ done
  THEN
UNTIL

BEGIN
  DUP  0080 AND     \ test if normalized
  OVER FF00 AND LNOT LAND
    IF 1
    ELSE
      DUP FF00 AND  \ test if any upper guardbit set
      IF   1DSHIFT> -1 2EXP +!
      ELSE 1<DSHIFT  1 2EXP +!
      THEN 0
    THEN
UNTIL

7F AND
D% 23 2EXP @ -
7F +
8<SHIFT 1SHIFT>
OR                \ insert in mantissa
ROT OR            \ insert sign
;


\ F.  output scien. notation

\ --------------------------------------------------
\ include 1<DSHIFT 1DSHIFT> 4<DSHIFT 4DSHIFT> (D.)

2 ZVARIABLE  2EXP
2 ZVARIABLE 10EXP

: /10  1 0 A 0 UD*/ ;
: *10  A 0 1 0 UD*/ ;

: ABS           \ (  N1 ---  N2 )
DUP 8000 AND IF NEGATE THEN ;

: SIGN.             \ ( N1 --- ) 0="+" 1="-"
8000 AND IF 2D ELSE 2B THEN EMIT ;

: F.                \ ( UD1 ---  )
DUP SIGN.           \ print sign mantissa
D% 23 OVER 1<SHIFT 8SHIFT> 7F - - 2EXP ! 0 10EXP !
00FF AND 0080 OR    \ --- LW HW' ) insert hidden bit

4<DSHIFT            \ renormalize mantisse
BEGIN
  2EXP @ DUP
  IF         \ not zero
    8000 AND
    IF   1<DSHIFT  1 2EXP +!
         DUP 8000 AND     IF /10 -1 10EXP +! THEN
    ELSE 1DSHIFT> -1 2EXP +!
         DUP FF00 AND LNOT IF *10  1 10EXP +! THEN
    THEN 0
  ELSE LNOT \ done: =0
  THEN
UNTIL
4DSHIFT>            \ renormalize mantisse

BEGIN
  967F 0098 2OVER D- SWAP DROP 8000 AND
  IF   /10 -1 10EXP +!
       0
  ELSE 1
  THEN
UNTIL

30 EMIT 2E EMIT \ "0,"
7 (D.)
45 EMIT         \ "e"
7               \ number of digits printed
10EXP @ -
DUP SIGN.
ABS 0 2 (D.)
;
```

Kontrollflussmanipulation mit Trampoline

Ulrich Hoffmann

In der vergangenen Ausgabe 2/2015 unseres Forth–Magazins hat uns Albert Nijhof mit seinem Artikel GATE — Eine Art GOTO in Forth *[1] in die Geheimnisse der portablen, beliebigen Kontrollflussänderungen eingeweiht und dabei eine Technik vorgestellt, die sich in seinem Wort* `GATE` *verbirgt.*

Dieser Artikel greift die von Albert präsentierte Technik auf und setzt sie in einen generelleren Zusammenhang.

Motivation und Grundlagen

Eine der Stärken von Forth ist, dass der Programmierer den Kontrollfluss seiner Programme weitgehend beeinflussen kann, etwa dadurch, dass er neue Kontrollstrukturen definiert. So ist in Forth klassisch keine `CASE`–Kontrollstruktur vorhanden, lässt sich aber eben leicht hinzufügen, wie etwa der grundlegende Artikel von Charles Eaker zeigt [2].

Will man den Kontrollfluss über Wortgrenzen hinweg bestimmen, so bieten einige historische Forth–Systeme die Möglichkeit, Kontrollstrukturen in einem Wort zu beginnen und in einem anderen Wort zu beenden — der Einsatz dieser Technik gilt jedoch als anrüchig und schwer zu verstehen — etwa:

```
: max ( n1 n2 -- n3 )
   2dup < BEGIN
   IF swap THEN drop ;

: min ( n1 n2 -- n3 )
   2dup > AGAIN ;

10 3 max .   10 ok
10 3 min .    3 ok
```

Das Wort `min` springt nach dem Test in das Wort `max` und verwirft dann dort den größeren Wert.

Ein solches Vorgehen ist in modernen Systemen durch die *Compiler Security* normalerweise nicht mehr möglich und auch die Forth–Standards sichern nicht zu, dass ein Standard–System solche Funktionalität unterstützt.

Eine weitere Art, den Kontrollfluss zu ändern, ist die Manipulation des Return–Stacks. Wird ein Wort w aufgerufen, so befindet sich in traditionellen, durch threaded Code implementierten Forth–Systemen die Rückkehradresse als oberstes Element auf dem Return–Stack. Die Rückkehradresse ist diejenige IP–Adresse (IP=Instruction–Pointer des Inner Interpreters), die der Adresse des Aufrufs von w unmittelbar folgt.

Durch Manipulation der Rückkehradresse auf dem Returnstack kann erreicht werden, dass im Aufrufer Programmteile übersprungen werden (die klassische Implementierung von `COMPILE` als `: COMPILE ( -- ) R> DUP @ , CELL+ >R ;` etwa überspringt das (kompilierte) Wort, das `COMPILE` folgt) oder zusätzliche Programmteile ausgeführt werden. Als Beispiel sei hier das Zurücksetzen von Variablen auf vorher gesicherte Werte genannt:

```
Create pull ] r> r> ! ;
: local ( addr -- )
   r>  rot dup >r @ >r pull >r >r ;

: .hex ( u -- )   base local hex . ;

decimal
10 1 - .  9 ok
42 .hex   2A ok
10 1 - .  9 ok
hex
10 1 - .  F ok
```

Auch Return–Stack–Manipulationen sind systemspezifisch (Native–Code–Compiler und ihre –Optimierer geben keine Zusicherung, dass sich immer die Rückkehradresse auf dem Return–Stack befindet, Stichwort *Tail–Call–Optimierung*) und daher auch kein Verhalten eines Standard–Programms.

In den Standards können ganze Forth–Worte über ihr *Execution–token* angesprochen werden. Dieses kann ausgeführt (`EXECUTE`) oder kompiliert werden (`COMPILE,`).

Teile eines Forth–Wortes kann man im Rahmen der Standards nicht weiter referenzieren. Die exakte Bedeutung der Kontrollstruktur–Steuerwerte (`BEGIN`, `IF`, `REPEAT`, . . .), die diese zur Übersetzungszeit verwalten, ist nicht bekannt und kann von System zu System variieren. Ebenso wenig ist bekannt, ob sie auf dem Datenstack oder einem eigenen Kontrollfluss–Stack liegen. Daher gibt es sogar eigene Stack–Operatoren (`CS-PICK`, `CS-ROLL` aber kein `CS-DROP` oder `CS-DEPTH`), um ihre Reihenfolge auf dem Stack zu verändern.

Standard–konform können Execution–Tokens von Worten mit `'` oder `[']` ermittelt werden, wenn der Name des Wortes bekannt ist. Definitionen ohne Namen lassen sich mit `:NONAME` erzeugen. Ist ein Execution–Token bekannt so kann es standardkonform nur **aufgerufen** werden (`EXECUTE` ruft es direkt auf, mit `COMPILE,` wird es zunächst kompiliert und dann später ggf. aufgerufen). Dabei wird zwangsweise die Rückadresse auf dem Return–Stack abgelegt. Was eben insbesondere nicht möglich ist, ist das **GOTO** zu einem Wort. Baut man Ketten von Aufrufen auf, so wird über kurz oder lang der Returnstack überlaufen.

Dieser Weg scheint also auch nicht gangbar zu sein. Müssen wir also auf portable Änderungen des Kontrollflusses verzichten?

GATE — Eine Art GOTO in Forth

Nein! Denn hier greift nun Alberts GATE–Technik. In seiner Implementierung aus dem letzten Heft finden wir:

```
: gate (ccc--)
  create ['] 0-gate , syntax ,
  does>
   begin @ execute ?dup 0= until ;
```

Uns interessiert hier der Does>–Teil: In einer Schleife wird @ EXECUTE aufgerufen, bis das aufgerufenen Wort eine 0 auf dem Stack hinterlässt. Ansonsten muss es eine Adresse hinterlassen, mit der das Spiel von vorne beginnen kann. Die durch diese Schleife aufgerufenen Worte müssen also alle den Stack–Effekt

```
( i*x -- j*x 0 | k*x a-addr2 )
```

haben.

Wie sieht es aber mit dem Return–Stack aus? Innerhalb der Schleife wird nur immer eine Rückkehradresse auf den Return–Stack gelegt (die Adresse vom ?DUP), also der Return–Stack nicht wirklich belastet, obwohl der Kontrollfluss zwischen den Worten wechselt. Um dies zu erreichen, müssen die Worte eben eine Adresse auf den Stack legen, unter der das Execution–Token des als nächstes auszuführenden Wortes zu finden ist, und zum Aufrufer zurückkehren. Bei Albert sind diese gelieferten Adressen die Adressen seiner GATEs, in denen sich das Execution–Token der bei ihm durch == definierten anonymen Worte befindet.

Alberts GOTO legt diese Adresse auf den Stack und verlässt das ==–Wort. Damit bekommt es den obigen Stack–Effekt. Die Folge ist, dass zwischen den ==–Worten hin– und–hergewechselt wird, bis schließlich ein READY eine 0 auf dem Stack hinterlässt und die EXECUTE–Schleife beendet wird. Was immer das letzte Wort ansonsten für Werte auf dem Stack hinterlässt, ist dann das Ergebnis des Aufrufs des ursprünglichen GATEs.

Der Kontrollfluss wird hier also explizit auf dem Daten–Stack verwaltet und kann dementsprechend manipuliert werden. Diese Technik ist unter dem Namen *Continuation Passing Style (CPS)* [4] bekannt und die EXECUTE–Schleife wird dann als *Trampolin* [5] bezeichnet. Die Continuations sind hier durch Execution–Tokens realisiert. Alberts GATE–Realisierung ist, soweit dem Autor dieses Artikels bekannt, die erste Implementierung eines Trampolins in Forth. (Anmerkungen und Richtigstellungen sind herzlich willkommen.) Interessanterweise hat das Trampoline Ähnlichkeit zur inneren Schleife von Umbilical–Forth–Implementierungen, in der das Target–System Execution–Tokens vom Host–System empfängt und sie dann mit EXECUTE ausführt.

Doch zurück zu Alberts Implementierung: Sein Wort GOTO ist ein Compiling–Wort, das in den ==–Worten allein oder zusammen mit -IF- verwendet werden kann, und muss dann unterschiedlichen Code generieren (GATE–Adresse ; bzw. GATE–Adresse EXIT THEN).

Insgesamt realisiert Albert mit seinem GATE–==–-IF-–GOTO–READY–Konstrukt sehr geschickt Zustandsautomaten (*Finate State Machine*) [3]. Ein wichtiges Werkzeug insbesondere der hardware–nahen Programmierung. Seine Automaten betten sich nahtlos in Forth–Programme ein: Automaten können dabei nämlich über ihre Zustände (die GATEs) wie ganz normale Forth–Worte aufgerufen werden. Das startet das Trampolin (die EXECUTE–Schleife), das durch die Zustände des Automaten springt. Ist der Automat in seinem Endzustand angekommen (READY), wird mit dem aufrufenden Forth–Programm fortgefahren. Elegant!

```
\ State machines via trampoline, based on Albert Nijhof's GATE, uho 2015-05-07

: trampoline ( i*x xt -- j*x )    BEGIN execute ?dup 0= UNTIL ;

: State: ( <name> -- )    Create 0 , Does> @ trampoline ;

: ==  ( <name> <def> -- 'state i*x ff )   ' >body :noname false ;

: hop ( tf | addr ff -- )
   IF postpone exit  postpone then   \ if following -if-
   ELSE  postpone ; swap ! THEN ;    \ at end of state

: goto ( <name> f -- )
   ' >body  postpone Literal  postpone @  hop ; immediate

: ready ( f -- )
   0  postpone Literal  hop ; immediate

: -if- ( f -- ) ( compile: i*x -- j*x tf )  postpone if  true ; immediate
```

Listing 1: Die Implementierung von Zustandsautomaten mit trampoline

```
1  \ Portable threaded code with trampoline          uho 2015-11-21
2
3  : trampoline ( i*x xt -- j*x )   BEGIN execute ?dup 0= UNTIL ;
4
5  :noname ( i*x addr1 -- i*x addr2 xt )  cr .s dup cell+ swap @ ; Constant NEXT
6
7  : run ( i*x -- j*x )   NEXT trampoline drop ;
8
9  0 Constant DONE
10
11 : dupp ( x addr -- x x addr xt )   over swap NEXT ;
12 : dropp ( x addr -- addr xt )      nip  NEXT ;
13 : show  ( x addr -- addr xt )      swap  ." -> ". NEXT ;
14 : dec   ( x1 addr -- x2 addr xt )  swap 1- swap NEXT ;
15 : zeq   ( x addr -- f addr xt )    swap 0= swap NEXT ;
16 : ?jump  ( x addr1 -- x addr2 xt )  swap IF  cell+ ELSE @ THEN  NEXT ;
17
18 Create countdown
19    ' dupp ,  ' show ,  ' dec ,
20    ' dupp , ' zeq ,  ' ?jump , countdown ,
21    ' dropp ,  DONE ,
```

Listing 2: Portabler threaded Code mit `trampoline`

Implementierung

Alberts Implementierung im letzten Heft verwendet *Compiler Security* durch die Worte `syntax` und `?syntax`. Das Listing 1 auf der gegenüberliegenden Seite verzichtet hingegen darauf, um die Implementierung kürzer zu halten. Es verwendet eine leichte Variation des Trampolins im Wort `trampoline`, der das `@` fehlt. Die aufgerufenen Worte haben nun die Aufgabe, gleich das neue Execution–Token zu liefern und nicht erst eine Adresse in der es sich befindet. Sie haben also den Stack–Effekt

`i*x -- j*x xt` .

Um Alberts Syntax beizubehalten, steuern `==` und `-IF-` über ein Flag, wie sich `GOTO` verhalten soll (vgl. `HOP`).[1] So lassen sich auch die Beispiele des letzten Heftes direkt mit der hier vorgestellten Implementierung verwenden.

Anwendung

Ist die Idee des Trampolins ersteinmal isoliert, kann es auch für weitere Kontrollstrukturen eingesetzt werden. Eine Möglichkeit ist es, portabel Threaded Code wieder verfügbar zu machen. der immer dann zum Einsatz kommen kann, wenn man darauf angewiesen ist, die Struktur des Codes genau zu kennen. Listing 2 zeigt, wie das geht. Für die Ausführung von Threaded Code — einem Vektor von Adressen — muss es einen Instruktionszeiger geben. Hier ist die Entscheidung getroffen worden, ihn als oberstes Element des Datenstacks zu halten. Damit kann der Instruktionszeiger von Worten verändert werden. Die Definition in Zeile 5 (`NEXT`) erklärt, wie der Instruktionszeiger fortschreitet und wie das als nächstes auszuführende Wort ermittelt wird. Dieses muss den Stack–Effekt

(i*x addr1 -- j*x addr2 xt)

haben. Neben seiner eigentlichen Aufgabe (`i*x` nach `j*x` zu transformieren), kann es den Instruktionszeiger verändern und muss (für das Trampolin) ein Execution–Token auf den Stack legen. Die hier definierten Worte legen immer `NEXT` auf den Stack, damit der nächste Befehl ausgeführt wird (könnten aber auch andere Excution–Tokens liefern, um die Ausführung ganz anders zu beeinflussen). Das Trampolin führt dadurch abwechselnd ein Wort aus, dann `NEXT`, dann wieder ein Wort, und so weiter, bis `DONE` (=`0`) die Ausführung beendet. In Zeile 11 bis 16 werden ein paar Forth–Worte definiert, die dieser Konvention genügen. `DUPP` und `DROPP` sind die Varianten, die `DUP` und `DROP` entsprechen, aber die Ausführung vorantreiben. `SHOW`, `DEC` und `ZED` entsprechen `.`, `1-` und `0=`. `?JUMP` entspricht dem Wort `?BRANCH` vieler Implementierungen. Es prüft das oberste Element des Stacks (unterhalb des Instructionszeigers) und springt zur Adresse, die in der nachfolgenden Zelle im Code–Vektor steht, falls das oberste Element `0` ist. Ansonsten wird die nachfolgende Zelle übersprungen und die Ausführung fährt mit der übernächsten Zelle fort.

Mit diesen Worten kann man nun den Threaded Code für `COUNTDOWN` definieren. Er wird hier per Hand konstruiert, aber man kann sich gut vorstellen, dass der Forth–Compiler auch leicht passenden Code generieren kann. `COUNTDOWN` entspricht der Definition

```
: countdown ( n -- )
   BEGIN dup . 1- dup 0= UNTIL drop ;
```

[1] Würde man von Alberts Syntax abweichen und stattdessen Forth–Standard–`IF`–`THEN` und ein explizites Ende des Zustands `==;` verwenden, so würde das Flag nicht mehr benötigt und die Implementierung ließe sich dann weiter verkürzen. Der geneigte Leser ist zu diesbezüglichen Experimenten eingeladen.

nur dass es über `RUN` aufgerufen werden muss und über `TRAMPOLINE` bzw. `NEXT` ausgeführt wird. `NEXT` enthält Ausgaben, die die Ausführung protokollieren:

```
3 countdown run

<2> 3 4312102448
<3> 3 3 4312102456 -> 3
<2> 3 4312102464
<2> 2 4312102472
<3> 2 2 4312102480
<3> 2 0 4312102488
<2> 2 4312102448
<3> 2 2 4312102456 -> 2
<2> 2 4312102464
<2> 1 4312102472
<3> 1 1 4312102480
<3> 1 0 4312102488
<2> 1 4312102448
<3> 1 1 4312102456 -> 1
<2> 1 4312102464
<2> 0 4312102472
<3> 0 0 4312102480
<3> 0 -1 4312102488
<2> 0 4312102504
<1> 4312102512  ok
```

Das oberste Stack–Element ist hierbei jeweils der Instruktionszeiger, die Elemente darunter die Daten, auf denen `COUNTDOWN` arbeitet.

Die Struktur des Code–Vektors `COUNTDOWN` ist bekannt und so kann er auch portabel (auch während der Ausführung) manipuliert werden.

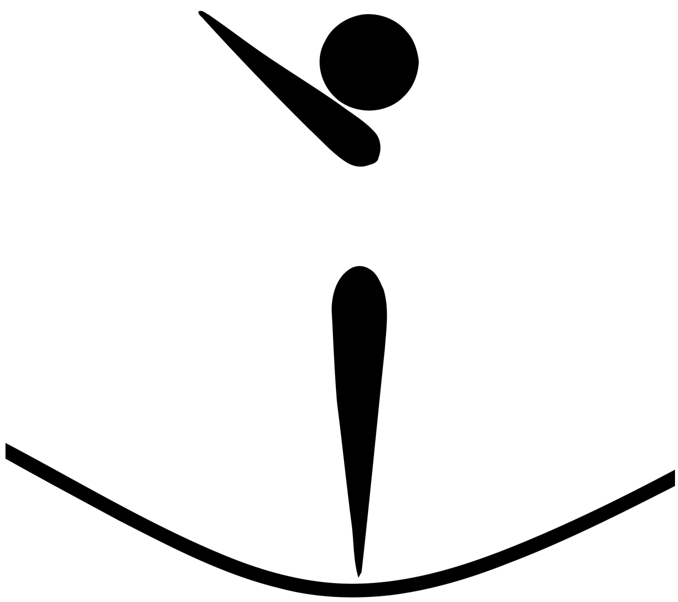
Ausblick

Mit Hilfe des Trampolins sollten sich viele, auch ungewöhnliche Kontrollstrukturen realisieren lassen. Backtracking, wie es beispielsweise für die Suche in Bäumen verwendet wird, ist sicher ein untersuchenswertes Thema. Oder, wie sieht es mit einem portablen Multitasker aus? Wenn wir den Kontrollfluss in der Hand haben, sollten wir auch einen Kontext–Wechsel portabel programmieren können. Man darf auf weitere Erkenntnisse und Einsatzgebiete des Trampolins gespannt sein.

May the Forth be with you.

Referenzen

1. *GATE — Eine Art GOTO in Forth*, Albert Nijhof, Vierte Dimension 2/2015
2. `http://www.forth.org/fd/FD-V02N3.pdf` (Charles Eaker, *Just in Case*, Forth Dimensions II/3, p. 37)
3. `https://de.wikipedia.org/wiki/Endlicher_Automat`
4. `https://de.wikipedia.org/wiki/Continuation-passing_style` und `https://en.wikipedia.org/wiki/Continuation-passing_style`
5. `https://en.wikipedia.org/wiki/Trampoline_(computing)` und `https://en.wikipedia.org/wiki/Tail_call#Through_trampolining`



Von Universal Time zu Epochensekunden und zurück

Erich Wälde

Letztes Jahr verbrachte ich einige Zeit mit dem Thema Uhr und Uhrzeit in Verbindung mit meinen amforth-Mikrocontroller-Basteleien. In diesem Zusammenhang ergab sich die Aufgabe, die bürgerliche Zeit in Epochensekunden umzurechnen und umgekehrt.

Damit wird hier sozusagen der erste Teil eines größeren Projekts vorgestellt[1]*. Ein Funkfreund sagte einmal, er hätte gerne eine Uhr, die per DCF77–Signal gestellt wird, dann aber bitte die Uhrzeit in UT (Universal Time) anzeigt ohne Sommerzeit–Firlefanz und sowas. Das war der Auslöser. Selbstverständlich baue ich so eine Uhr mit einem Atmega-Mikrokontroller, welcher mit AmForth programmiert wird - auch wenn das in unwissenden Kreisen gelegentlich für merkwürdige Diskussionen sorgt.*

Intro

In der Unixwelt wird die Zeit normalerweise als *Anzahl der Sekunden seit dem 1.1.1970 0h UT* in der Zeitbuchhaltung des Kernels geführt. Der genannte Zeitpunkt markiert den Beginn der *Epoche*. Der Unix–Befehl `date` kann die Zeit in Epochensekunden anzeigen:

```
$ date +%s
1446650000
```

Auch die Umkehrung gelingt leicht:

```
$ date -u -d @1445566000
Fri Oct 23 02:06:40 UTC 2015
```

Um die Uhrzeit von MEZ/MESZ in UT umzurechnen, beschloss ich, die entsprechenden Berechnungen durch Wandlung in Epochensekunden zu bewerkstelligen. Die Funktionen wurden ein wenig getestet, und zwar sowohl mit *gForth* als auch mit *AmForth*.

Implementierung

Schaltjahr

Die Schaltjahresregel heißt heutzutage: Jahre, die ohne Rest durch 400 teilbar sind, sind Schaltjahre. Jahre die ohne Rest durch 100 teilbar sind, sind **keine** Schaltjahre. Jahre die ohne Rest durch 4 teilbar sind, sind Schaltjahre. Alle übrigen Jahre sind keine Schaltjahre.

```
: leapyear? ( yyyy -- t/f )
  dup &4 mod 0=
  over &100 mod 0<> and
  swap &400 mod 0= or
;
```

Ein kleiner Test kann bekanntlich nichts schaden:

```
$ gforth
Gforth 0.7.9_20150825, Copyright (C) 1995-2015
Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY;
for details type ‘license’
Type ‘bye’ to exit
: leapyear? ( yyyy -- t/f ) compiled
dup &4 mod 0= compiled
over &100 mod 0<> and compiled
swap &400 mod 0= or compiled
; ok
include test/tester.fs
/usr/local/share/gforth/0.7.9_20150825/test/
tester.fs:8: warning: redefined { ok
decimal ok
t{ 1970 leapyear? ->  0 }t ok
t{ 1972 leapyear? -> -1 }t ok
t{ 1999 leapyear? ->  0 }t ok
t{ 2000 leapyear? -> -1 }t ok
t{ 2001 leapyear? ->  0 }t ok
t{ 2004 leapyear? -> -1 }t ok
t{ 1900 leapyear? ->  0 }t ok
t{ 3900 leapyear? ->  0 }t ok
t{ 3996 leapyear? -> -1 }t ok
t{ 4000 leapyear? -> -1 }t ok
```

Die getesteten Fälle sehen gut aus.

Von Epochensekunden bis zur Uhrzeit

Zuerst soll die Umrechnung von Epochensekunden in *normale Zeit* (UT) mit Minuten, Stunden usw. berechnet werden. Die Funktion `ud/mod` wird uns dabei helfen. In *gForth* reicht auf meinem Rechner eine Zelle, aber auf dem Mikrokontroller mit AmForth muss eine doppelt lange Zahl benutzt werden. Deswegen wird das hier teilweise mit doppelt langen Zahlen und ihren Operatoren vorgeführt:

```
$ gforth
Gforth 0.7.9_20150825, Copyright (C) 1995-2015
Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY;
for details type ‘license’
Type ‘bye’ to exit
decimal  ok
1445566000. 60 ud/mod .s <3> 40 24092766 0  ok
```

Die 40 sind die Sekunden, die große Zahl ist der Rest der Zeit in Minuten. Die Operation `ud/mod` wird wiederholt.

[1] *Ich liebe Serien! (der sätza)*

```
60 ud/mod .s <4> 40 6 401546 0 ok
```

Die 6 sind die Minuten, die große Zahl ist der Rest der Zeit in Stunden. Auch die nächste Operation ist noch einsichtig.

```
24 ud/mod .s <5> 40 6 2 16731 0 ok
```

02:06:40 als Uhrzeit ist schon mal korrekt. Es sind jetzt noch 16731 Tage in ein Datum (Tag-Monat-Jahr) umzurechnen. Schön wäre es, wenn wir über eine ähnliche Funktion wie `um/mod` verfügen würden, etwa `months/mod` und `years/mod` . Also bauen wir uns diese Funktionen.

`years/mod` und `months/mod`

Zuerst definieren wir den Beginn der Epoche. Danach schreiben wir eine Funktion, die für ein gegebenes Jahr die Jahreslänge in Tagen berechnet. Das gestaltet das weitere Programm besser lesbar.

```
&1970 constant __Epoch
: 365+1 ( year -- 365|366 )
    &365 swap leapyear? if 1+ then
;
```

Damit lässt sich die Funktion `years/mod` aufschreiben, welche aus einer Anzahl von Tagen die ganzen Jahre ausrechnet und abzieht und den Rest in Tagen auf dem Stapel liegen lässt. Zunächst wird festgestellt, ob die übergebene Zahl überhaupt ein ganzes Jahr ergibt. Wenn ja, dann werden so lange die korrekten Jahreslängen in Tagen abgezogen, bis kein ganzes Jahr mehr übrig ist. Man beginnt die Schleife mit dem Jahr der Epoche, also 1970.

```
: years/mod ( T/day -- years T/day' )
  dup &365 u>= if          \ -- T
      __Epoch swap         \ -- year T
      begin
          over 365+1
          -
          swap 1+ swap     \ -- T-365 year+1
          over 365+1       \ -- year' T' 365
          over swap        \ -- year' T' T' 365
      u>= 0= until
  else
      __Epoch swap
  then
;
```

Das ist vielleicht keine zahlentheoretisch elegante Angelegenheit, aber sie funktioniert, wie man überprüfen kann.

```
$ gforth
...
t{     0 years/mod -> 1970   0 }t  ok
t{     1 years/mod -> 1970   1 }t  ok
t{    31 years/mod -> 1970  31 }t  ok
t{   364 years/mod -> 1970 364 }t  ok
t{   365 years/mod -> 1971   0 }t  ok
t{   366 years/mod -> 1971   1 }t  ok
t{   730 years/mod -> 1972   0 }t  ok
t{  1094 years/mod -> 1972 364 }t  ok
t{  1095 years/mod -> 1972 365 }t  ok
t{  1096 years/mod -> 1973   0 }t  ok
t{  1097 years/mod -> 1973   1 }t  ok
t{ 11322 years/mod -> 2000 365 }t  ok
t{ 11323 years/mod -> 2001   0 }t  ok
...
.s <5> 40 6 2 16731 0  ok
d>s .s <4> 40 6 2 16731  ok
years/mod .s <5> 40 6 2 2015 295  ok
```

Der Datumsteil ergibt Jahr 2015 und 295 Tage Rest. Aus dem Rest sind jetzt noch Monat und Tag zu berechnen. Um den Monat zu berechnen, habe ich eine Liste angelegt, die die Anzahl der Tage bis zum Ende des Monats N enthält. Die Funktion `months/mod` benötigt auch das Jahr, um Schaltjahre korrekt zu behandeln.

```
create __acc_days
0 ,
&31 , &59 , &90 , &120 , &151 , &181 ,
&212 , &243 , &273 , &304 , &334 , &365 ,

: months/mod ( year T/day -- year month T/day' )
  dup 0= if
    drop 1 1
  else
    &12 swap   \ -- year month T
    begin
      \ over __acc_days + @i     \ amForth
      over cells __acc_days + @  \ gForth
      \ -- year month T acc_days[month]
    \ correct acc_days for leap year ...
    \ ... and months > 1 (January)
      3 pick leapyear?
      3 pick 1 > and if 1+ then
    \ -- year month T
    \    acc_days[month] acc_days[month] T
      over over swap
      u>
    while  \ -- year month T
      drop swap 1- swap
        \ -- year month-1 T
    repeat \ -- year month' T acc_days[month']
        -  \ -- year month' T-acc_days[month']
    swap 1+
    swap 1+
  then
;
```

Diese Funktion ist möglicherweise etwas umständlich geraten. Man könnte auch zwei Listen machen, eine für Gemeinjahre, eine für Schaltjahre. Ob das die Funktion besser lesbar macht, kann ich im Moment nicht beurteilen. Jedenfalls funktioniert sie zufriedenstellend.

```
gforth
...
t{ 1970   0 months/mod -> 1970  1  1 }t  ok
t{ 1970   1 months/mod -> 1970  1  2 }t  ok
t{ 1970  30 months/mod -> 1970  1 31 }t  ok
t{ 1970  31 months/mod -> 1970  2  1 }t  ok
t{ 1970  59 months/mod -> 1970  3  1 }t  ok
t{ 1970  90 months/mod -> 1970  4  1 }t  ok
t{ 1970 120 months/mod -> 1970  5  1 }t  ok
```

```
t{ 1970 151 months/mod -> 1970  6  1 }t  ok
t{ 1970 181 months/mod -> 1970  7  1 }t  ok
t{ 1970 212 months/mod -> 1970  8  1 }t  ok
t{ 1970 243 months/mod -> 1970  9  1 }t  ok
t{ 1970 273 months/mod -> 1970 10  1 }t  ok
t{ 1970 304 months/mod -> 1970 11  1 }t  ok
t{ 1970 334 months/mod -> 1970 12  1 }t  ok
t{ 1970 364 months/mod -> 1970 12 31 }t  ok
t{ 1996   0 months/mod -> 1996  1  1 }t  ok
t{ 1996   1 months/mod -> 1996  1  2 }t  ok
t{ 1996  30 months/mod -> 1996  1 31 }t  ok
t{ 1996  31 months/mod -> 1996  2  1 }t  ok
t{ 1996  60 months/mod -> 1996  3  1 }t  ok
t{ 1996  91 months/mod -> 1996  4  1 }t  ok
t{ 1996 121 months/mod -> 1996  5  1 }t  ok
t{ 1996 152 months/mod -> 1996  6  1 }t  ok
t{ 1996 182 months/mod -> 1996  7  1 }t  ok
t{ 1996 213 months/mod -> 1996  8  1 }t  ok
t{ 1996 244 months/mod -> 1996  9  1 }t  ok
t{ 1996 274 months/mod -> 1996 10  1 }t  ok
t{ 1996 305 months/mod -> 1996 11  1 }t  ok
t{ 1996 335 months/mod -> 1996 12  1 }t  ok
t{ 1996 365 months/mod -> 1996 12 31 }t  ok
...
.s <5> 40 6 2 2015 295  ok
months/mod .s <6> 40 6 2 2015 10 23  ok
swap rot .s <6> 40 6 2 23 10 2015  ok
```

Epochensekunden in UT umwandeln

Die Umwandlung von Hand ist also erfolgreich. Jetzt kann man die gewünschte Funktion `s>ut` fertigstellen.

```
: s>ut
  ( d:EpochSeconds --
  (   sec min hour day month year/UT )
  &60 ud/mod  \ -- sec d:T/min
  &60 ud/mod  \ -- sec min d:T/hour
  &24 ud/mod  \ -- sec min hour d:T/day
  d>s
  years/mod   \ -- sec min hour year T/day
  months/mod  \ -- sec min hour year month day
  swap        \ -- sec min hour year day month
  rot         \ -- sec min hour day month year
;
```

Es war aufwendiger, die zugehörigen Tests aufzuschreiben, als die Funktion selbst zu definieren. Und ja, die Tests haben sich gelohnt, weil ich zuerst einen subtilen Fehler in `years/mod` eingebaut hatte.

```
gforth
...
1445566000. s>ut .s <6> 40 6 2 23 10 2015  ok
...
t{          0. s>ut ->  0  0  0  1  1 1970 }t ok
t{       3600. s>ut ->  0  0  1  1  1 1970 }t ok
t{      86400. s>ut -> 00 00 00 02 01 1970 }t ok
t{   31536000. s>ut -> 00 00 00 01 01 1971 }t ok
t{  100000000. s>ut -> 40 46 09 03 03 1973 }t ok
t{  951782400. s>ut -> 00 00 00 29 02 2000 }t ok
t{ 1000000000. s>ut -> 40 46 01 09 09 2001 }t ok
  ok
t{ 1044057600. s>ut -> 00 00 00 01 02 2003 }t ok
t{ 1044144000. s>ut -> 00 00 00 02 02 2003 }t ok
t{ 1046476800. s>ut -> 00 00 00 01 03 2003 }t ok
t{ 1064966400. s>ut -> 00 00 00 01 10 2003 }t ok
\ leap year, end of February  ok
t{ 1077926399. s>ut -> 59 59 23 27 02 2004 }t ok
t{ 1077926400. s>ut -> 00 00 00 28 02 2004 }t ok
t{ 1077926410. s>ut -> 10 00 00 28 02 2004 }t ok
  ok
t{ 1078012799. s>ut -> 59 59 23 28 02 2004 }t ok
t{ 1078012800. s>ut -> 00 00 00 29 02 2004 }t ok
t{ 1078012820. s>ut -> 20 00 00 29 02 2004 }t ok
  ok
t{ 1078099199. s>ut -> 59 59 23 29 02 2004 }t ok
t{ 1078099200. s>ut -> 00 00 00 01 03 2004 }t ok
t{ 1078099230. s>ut -> 30 00 00 01 03 2004 }t ok
  ok
t{ 1078185599. s>ut -> 59 59 23 01 03 2004 }t ok
t{ 1096588800. s>ut -> 00 00 00 01 10 2004 }t ok
  ok
t{ 1413064016. s>ut -> 56 46 21 11 10 2014 }t ok
t{ 1413064100. s>ut -> 20 48 21 11 10 2014 }t ok
  ok
\ 31 bit max  ok
t{ 2147483648. s>ut -> 08 14 03 19 01 2038 }t ok
t{ 2147483649. s>ut -> 09 14 03 19 01 2038 }t ok
\ 32 bit max  ok
t{ 4294967295. s>ut -> 15 28 06 07 02 2106 }t ok
\ this is still working because I use  ok
\ Epoch seconds as 32 bit *unsigned* integer  ok
\ in disagreement with the standard definition  ok
\ overflow here :-) with AmForth, gForth ok
t{ 4294967296. s>ut -> 16 28 06 07 02 2106 }t ok
```

Das ist jetzt insofern alles gemogelt, weil Zeitzonen-Information bis zu dieser Stelle nicht berücksichtigt wurde.

UT in Epochensekunden unwandeln

Die umgekehrte Funktion ist eine reine Fleißaufgabe. Mit dem Jahr beginnend rechnet man die Zahlen auf dem Stapel in immer kleinere Zeit-Einheiten (Tage, Minuten, Sekunden) um und addiert die darunter liegende Zahl. Aber wir können von den oben verwendeten Hilfsmitteln profitieren. Damit man hier nachvollziehen kann, was da passiert, wurde für jeden wichtigen Schritt ein voller Stack-Kommentar angefügt[2]

```
: ut>s ( s m h d m y -- d:T/sec )
    \ add start value T=0
    0 over        \ -- s m h d m y T=0 y
    __Epoch       \ -- s m h d m y T y E
    ?do
i 365+1 +
    loop          \ -- s m h d m y T/days
    2 pick 1-     \ -- s m h d m y T/days month-1
    \ __acc_days + @i
    \ -- s m h d m y T/days acc_days[month] \ amForth
    cells __acc_days + @
```

[2] Im Original wurden die vollen Namen der Werte benutzt. Das hätte aber die Spaltenbreite überschritten. Daher wurde gekürzt: `sec min hour day month year Epoch` wurden zu `s m h d m y E` . Ihr werdet das mental schon hinbekommen.

```
    \ -- s m h d m y T/days acc_days[month] \ gForth
    +             \ -- s m h d m y T/days
    swap          \ -- s m h d m T/days y
    leapyear? rot 2 > and if 1+ then
    \             \ -- s m h d T/days
    swap 1- +     \ -- s m h T/days
    s>d
    24 1 m*/ rot s>d d+  \ -- s m T/hours
    60 1 m*/ rot s>d d+  \ -- s T/minutes
    60 1 m*/ rot s>d d+  \ -- T/sec
;
```

Das gerade berechnete Datum lässt sich wieder in Epochensekunden zurückwandeln.

```
gforth
...
.s <6> 40 6 2 23 10 2015  ok
ut>s .s <2> 1445566000 0  ok
```

Zeitzone MEZ/MESZ

Ich brauchte keine generelle Behandlung von Zeitzonen, weil ich lediglich die Zeit, die der Sender DCF77 ausstrahlt, in UT umwandeln wollte. Daher ist dieser Teil etwas schmal geraten, das geht sicher besser. Hier wird als unterstes Argument auf dem Stapel die Zeitdifferenz zwischen der Zeitzone und UT verlangt.

```
3600 constant CET
7200 constant CEST

: dt>s ( tzoffset s m h d m y -- d:epochsec )
  ut>s
  rot s>d d-
;
```

```
$ gforth
...
CET  0 0 0 28 10 2015 dt>s 2dup ud. 1445986800  ok
CEST 0 0 0 28 10 2015 dt>s 2dup ud. 1445983200  ok
d- ud. 3600  ok
```

Auch dieser Teil verdient ein paar Tests.

```
$ gforth
...
decimal  ok
t{ CET   0  0  0  1   1 1970 dt>s -> -3600. }t  ok
t{ CET   0  0  1  1   1 1970 dt>s ->     0. }t  ok
t{ 0. s>ut ->  0  0  0  1   1 1970 }t  ok
t{ CET  59 59 23  1   1 1970 dt>s -> 82799. }t  ok
t{ CET  59 59  0 02  01 1970 dt>s -> 86399. }t  ok
t{ CET   0  0  1  2   1 1970 dt>s -> 86400. }t  ok
t{ 86400. s>ut -> 00 00 00 02 01 1970 }t  ok
t{ CET   1  0  1  2   1 1970 dt>s -> 86401. }t  ok
t{ CET  59 59 23 31   1 1970 dt>s ->    2674799. }t  ok
t{ CET   0  0  0  1   2 1970 dt>s ->    2674800. }t  ok
t{ CET   1  0  0  1   2 1970 dt>s ->    2674801. }t  ok
t{ CET  59 59 23 28   2 1970 dt>s ->    5093999. }t  ok
t{ CET   0  0  0  1   3 1970 dt>s ->    5094000. }t  ok
t{ CEST 30 15 12  1   6 1971 dt>s ->   44619330. }t  ok
t{ CEST  6  3 17 12  10 2014 dt>s -> 1413126186. }t  ok
t{ CEST 00 00 00 29  06 2000 dt>s ->  962229600. }t  ok
t{ CET  00 00 00 29  01 2000 dt>s ->  949100400. }t  ok
t{ CET  00 00 00 28  02 2000 dt>s ->  951692400. }t  ok
t{ 951692400. s>ut -> 00 00 23 27 02 2000 }t  ok
t{ CET  00 00 00 29  02 2000 dt>s ->  951778800. }t  ok
t{ CET   0  0  1  1   1 1970 dt>s ->          0. }t  ok
t{ 0. s>ut ->  0  0  0  1   1 1970 }t  ok
t{ CET   0  0  1 29   2 1972 dt>s ->   68169600. }t  ok
t{ 68169600. s>ut ->  0  0  0 29   2 1972 }t  ok
t{ CET  00 00 01 28  02 1972 dt>s ->   68083200. }t  ok
t{ 68083200. s>ut -> 00 00 00 28 02 1972 }t  ok
t{ CEST 40 46 03 09  09 2001 dt>s -> 1000000000. }t  ok
t{ 1000000000. s>ut -> 40 46 01 09 09 2001 }t  ok
t{ CET  00 00 01 01  01 2004 dt>s -> 1072915200. }t  ok
t{ 1072915200. s>ut -> 00 00 00 01 01 2004 }t  ok
```

So Sachen

Dieses Projekt fing klein an, zog allerdings lustige und interessante Ideen an, wie Licht die Motten anzieht. Etwa in Sachen Hardware: LED-Ziffern wären ja eine naheliegende Lösung. Normalerweise beleuchtet man die Ziffern im Zeit-Multiplex-Verfahren. Aber es gibt so grottige Implementierungen, dass ich's da blinken sehe. Deswegen werden alle Ziffern flackerfrei mit Schieberegistern gesteuert. Wollte ich die Helligkeit der Ziffern regeln, dann würde ich das über eine PWM-Steuerung in der Stromversorgung der Ziffern regeln, die mit etwa 1kHz läuft - schnell genug, dass ich es nicht flackern sehe.

Andere Anzeigen sind selbstverständlich denkbar, andere Uhrzeitdarstellungen oder Zeiten ebenfalls. Man könnte die Epochensekunden für einen erhöhten Geek-Faktor anzeigen, oder die Sternzeit. Ach, und eine Weckfunktion ist natürlich auch wünschenswert. Und spätestens jetzt entgleist die ganze Angelegenheit in unkontrollierbare Kreativität.

Ceterum censeo ... ja, im Übrigen bin ich entschieden der Meinung, dass der Sommerzeit-an-der-Uhr-dreh-Zirkus dringend und ersatzlos gestrichen gehört.

Referenzen

1. http://amforth.sourceforge.net/
2. https://de.wikipedia.org/wiki/Universal_Time
3. https://de.wikipedia.org/wiki/DCF77
4. https://de.wikipedia.org/wiki/Sommerzeit
5. https://de.wikipedia.org/wiki/Unixzeit

Forth–Gruppen regional

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

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

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

Ruhrgebiet **Carsten Strotmann**
`ruhrpott-forth@strotmann.de`
Treffen alle 1–2 Monate Freitags im Unperfekthaus Essen
`http://unperfekthaus.de`.
Termine unter: `http://forth-ev.de`

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

Forth-Hardware in VHDL microcore (uCore)	**Klaus Schleisiek** Tel.: **(0 75 45) – 94 97 59 3**$_p$ kschleisiek@freenet.de
KI, Object Oriented Forth, Sicherheitskritische Systeme	**Ulrich Hoffmann** Tel.: **(0 41 03) – 80 48 41** `uho@forth-ev.de`
Forth–Vertrieb volksFORTH ultraFORTH RTX / FG / Super8 KK–FORTH	Ingenieurbüro **Klaus Kohl–Schöpe** Tel.: **(0 82 66)–36 09 862**$_p$

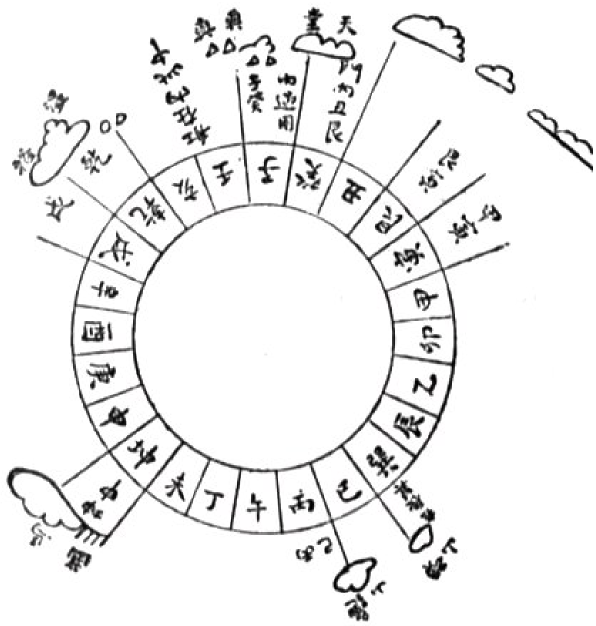
Termine

Donnerstags ab 20:00 Uhr
Forth–Chat IRC `#forth-ev`

27.–30. Dezember 2015:
32C3 — 32. Chaos Communication Congress, Hamburg
`https://events.ccc.de`

12.–13. März 2016:
1. Maker Faire Ruhr
`http://www.makerfaire-ruhr.com`

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

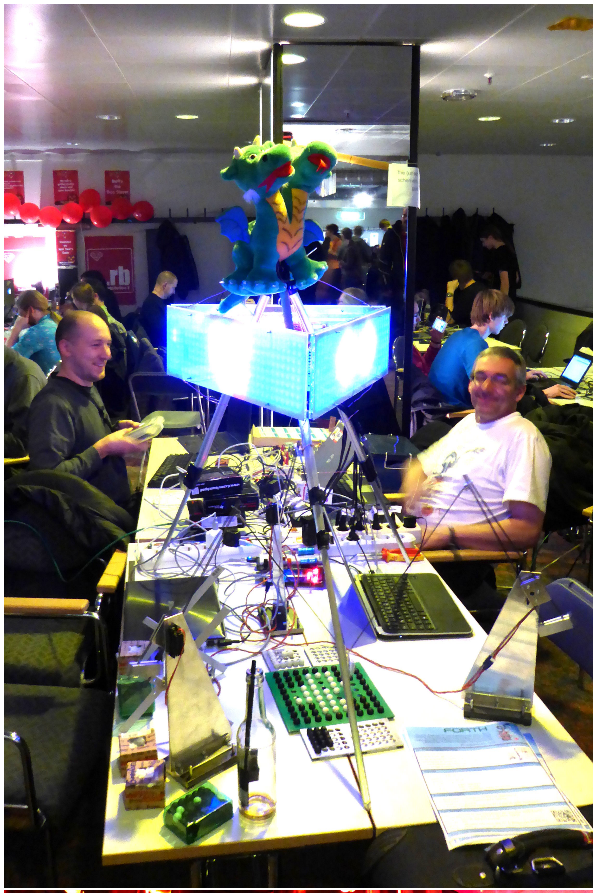


Möchten Sie gerne in Ihrer Umgebung eine lokale Forthgruppe gründen, oder einfach nur regelmäßige Treffen initiieren? Oder können Sie sich vorstellen, ratsuchenden Forthern zu Forth (oder anderen Themen) Hilfestellung zu leisten? Möchten Sie gerne Kontakte knüpfen, die über die VD und das jährliche Mitgliedertreffen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

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

32c3 vom 27. bis 30. Dezember in Hamburg

Die Forth–Gesellschaft trägt auch dieses Jahr zum Chaos Communication Congress in Hamburg bei. Für das Standpersonal durchaus anstrengend, denn Hacker kommen zwar frühestens mittags aus dem Bett, gehen aber nie vor 3 Uhr nachts schlafen — aber nur, wenn sie vorher eine komplette Einweisung in Forth bekommen haben.



Assembly

Auf unserer Assembly stellen wir die Bitkanone, den Triceps, b16, Gforth (insbesondere auf Android) und natürlich Mecrisp vor, diskutieren mit Besuchern und werben für Forth.

Vortrag "Compileroptimierungen für Forth im Microcontroller"

Matthias Koch hält einen Vortrag über seinen optimierenden Compiler.

Session #wefixthenet "net2o — make it userfriendly"

Bernd Paysan hält voraussichtlich einen net2o–Vortrag in einer selbst–organisierten Session; für genaue Details ist dieses Heft zu früh fertig geworden.

Mehr

Der CCC ist eine riesige Veranstaltung, die wahnsinnig viel Spaß macht: Von Tech–Talks über Politisches, humorvolles (Fnord–Jahresrückblick!) bis zur Kunst. Mehr auf der Congress–Homepage:
`https://events.ccc.de/congress/2015/wiki/Main_Page`