Vierte Dimension

3/2022

Das Forth–Magazin

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

In dieser Ausgabe:

Titelbild: ASCII–Art Pusteblume AUTOR mk

Quelle: Anregung aus dem Internet; eigenes Werk.

Impressum

Name der Zeitschrift

Vierte Dimension

Herausgeberin

Forth–Gesellschaft e. V.
Postfach 1030
48481 Neuenkirchen
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,

Das Forth–Sommertreffen im Linuxhotel Essen vom 15.–17. Juli diesen Jahres war sehr angenehm. Sich so ungezwungen austauschen zu können, finde ich immer noch am besten. Da wir alle in der Welt verstreut arbeiten, ist Telekommunikation nötig, das ist klar. Doch ein persönlicher Kontakt hin und wieder verbindet besser, finde ich. Daher bin ich gespannt, ob wir auch 2023 wieder eine Zusammenkunft haben werden. Gewünscht wurde es in der Runde, wieder im früheren Rhythmus soll es auch sein, also so um Ostern herum. Wer es dann wo macht, ist noch offen. Haltet die Augen offen. Auf `www.forth-ev.de` wirds stehen.

RISC–V zieht Kreise auf FPGAs. „Willst du einen Prozessor richtig kennenlernen, schreib einen Disassembler dafür!“, hat mal ein weiser Programmierer gesagt. Klaus Kohl–Schöpe hat’s gemacht, in Forth versteht sich, Basisarbeit sozusagen. Damit Forth überall Verbreitung findet.

Künstler sind oft Vordenker, Wegbereiter. Matthias Koch ließ sich inspirieren von der ASCII–Art. Und fand, dass man auf diese Weise auch prima technische Darstellungen schaffen kann, ganz ohne graphische Oberflächen.

Die Colaboration in Forth Works, Albert Nijhof and Ulrich Hoffmann haben einen Weg gefunden, auf sehr kleinen MCUs die Ziffenausgabe zu ermöglichen. Nur HEX, aber erstaunlich einfach!

Die kompliziertere Variante der Zifferndarstellung mit allem Drum und Dran hat Gforth. Dem bin ich mal nachgegangen und habe dabei Erstaunliches gelernt: Von der Vorstellung einer Zahl im menschlichen Hirn zu deren Darstellung im Elektronengehirn und die Übermittlung von Zahlen zwischen Hirnen, auch elektronischen, war ein langer Weg. Dieser kulturelle Akt ist nicht trivial. Aber fundamental. Systemrelevant, sagt man da heute wohl, hüben wie drüben. Das zu verstehen, dazu bietet das aktuelle Gforth 0.7.9 prima Werkzeuge.

Eine besondere Anerkennung sprach Philip Zembrod aus: „Nachdem ich die letzte VD dann in der Hand hielt und meinen Beitrag *Unboxing Swappy* nochmal gelesen hatte, fand ich, daß eine Sache nicht so ganz deutlich herauskam, nämlich was für eine herausragende Arbeit Wolfgang Strauss mit der neuen Drachen–Transportbox geleistet hat. Das ist wirklich, wie man im Englischen so schön sagen kann, above and beyond, und dafür möchte ich dir, lieber Wolfgang, an dieser Stelle noch einmal ganz herzlich und im Namen aller danken!“ (Du sprichst mir aus der Seele, Philip, zumal ich ja derjenige war, der damals Swappy so leichtfertig im Pappkarton der Post überlassen hat — mit den bekannten Folgen.)

Und eine traurige Nachricht gibt es leider auch: Dr. Chen–Hanson Ting ist verstorben. Er hinterlässt uns eForth, bestens dokumentiert. Ein kultureller Schatz. Im Heft mehr dazu.

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-2022-03`

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

Ulrich Hoffmann Kontakt: `Direktorium@Forth-ev.de`
Bernd Paysan
Gerald Wodni

Mr. Chen–Hanson Ting verstorben

August 29, 1939 — May 30, 2022

Abbildung 1: Chen–Hanson Ting

Die Nachricht von Dr. Tings Ableben erreichte uns kurz nach Redaktionsschluss des vorherigen Heftes, zunächst über Facebook, dann aber auch über die Veröffentlichung des Bestatters.

> „Obituary for Mr. Chen-Hanson Ting.
>
> It is with great sadness that we announce the passing of Chen–Hanson Ting on May 30, 2022, in San Mateo, California. Mr. Ting was born in China and is survived by a spouse and children.
>
> Arrangements have been entrusted to the care of Skylawn Funeral Home & Memorial Park, Hwy 92 at Skyline Blvd., San Mateo, CA 94402.“ [Quelle: `https://www.skylawnmemorialpark.com/obituaries/Chen-Hanson-Ting/#!/Obituary`]

So mussten wir es also glauben. Ein erster Nachruf auf `comp.lang.forth` kam von Dirk Brühl:

> „In Memory to Dr. Chen–Hanson Ting.
>
> Do you Remember RSC–FORTH in 1984? Shortly after there was Chuck Moore's NOVIX 16–bit microprocessor available, and the first story about this microprocessor was written by Dr. Chen–Hanson Ting: *Footsteps in an empty valley*, 1986, eloquently written, a book everybody should have red to discover what it is all about. Over the years Dr. Chen–Hanson Ting supported the Forth–Community with all his heart. He even did the last international Forth Zoom conferences from the hospital room. He filled a bookshelf with Forth literature, especially with his eForth and all it's portations to different microprocessors. He was a genius. He got all the skills to do this work.
>
> He is one of the giants on whose shoulders we are standing now.
>
> He left us too early, fighting for his life and for his Forth Community with his goal *Zen of Forth* until his end.
>
> I invite everybody who worked with him to tell our Forth Community about this experience.
>
> Rest In Peace, Chen!“

Wir haben uns nicht persönlich getroffen, Deutschland und Kalifornien liegen weit auseinander. Im Zeitalter des Internet und von E–Mail fand aber ab und zu ein Austausch statt über sein eForth für den MSP430G2553. Es ging damals um die Frage, welche Implementation die schnellsten Ausführungszeiten und den dichtesten Code erlauben würden, und wir kamen überein, dass es ein „direkt threaded Forth code“ sein müsste. Das `next` wird dann kurz und schnell, weil der Instruktionssatz dieser 16–Bit–Maschine dafür wie gemacht ist und ein Forth–Token nur 16 Bit erfordert.

```
mov @ip+,pc ; fetch code address into PC
```

Und weil er der Auffassung war, alles so einfach wie möglich zu halten, gab es daraufhin den eForth–Kern als ein simples Assembler–File, nichteinmal die Header der Forthworte wurden von Macros generiert, nein, alles „handcoded“, frei von den Abhängigkeiten eines speziellen Assemblers, er wechselte von IAR nach CCS und von dort ging es auch ohne IDE mit einfachem Werkzeug wie dem Naken–Assembler zu übersetzen, anstandslos. Der Programmierer selbst muss allerdings wissen, wie sein eForth werden soll und sich darin gut auskennen. Doch weil eForth so unkompliziert gemacht ist, kann man es gut schaffen, das vollständig zu durchschauen.

So denke ich immer an Dr. Ting, sobald es ans Werkeln mit dem MSP430 geht, und danke ihm für dieses große Geschenk.

Im *Zen of LaunchPad*[1] schrieb er dazu:

> „The most interesting aspect of Forth is that it retains names of commands in memory. Most other programming languages throw away the names after code is compiled. Names were only temporary devices used for the convenience of the programmer in the process of programming, and they are of no value in the final product. In a living and growing computing system, just like in natural languages, names are representations of intelligence and are the only vehicles for thinking, reasoning, abstraction, communication, and accumulation of knowledge and technology. Lao–Tze was the first person who realized the significance of names along with Tao. He had profound understanding of nature, and perhaps, computers. He opened his Tao–Te Ching with some profound statements.

[1] Version 4.3, Offete Enterprises, Inc., 2015

> Nobody really knows what he's talking about, but I do, because I think he was talking about computers. I will show you my translation and my interpretation. The Chinese text was based on a recently (1973) excavated version of Tao–Te Ching, similar to but not the same as you find in your local library.
>
> 道可道也，非恆道也。 Eternal Tao cannot be spoken,
> 名可名也，非恆名也。 Eternal name cannot be named.
> 無名萬物之始也； In the beginning, Tao has no name,
> 有名萬物之母也。 But, Name is the mother of everything.
> 故恆無欲也，以觀其眇； Tao is manifested by its actions,
> 恆有欲也，以觀其所噭。 Name reveals the nature of Tao.
> 兩者同出，異名同胃， Tao and Name are one and same,
> 玄之有玄，眾眇之門。 Uttermost profound, mystery of mysteries!
>
> Lao–Tze was talking about Tao and Name. Tao as Nature, can be observed. To understand Tao, and to communicate Tao, we need Names. I think he was actually talking about computers and firmware engineering. A computer has two components. It has programs which can do wonderful things beyond human comprehension. Programs are best modularized and constructed in nested lists. If all modules and lists are given proper names, programs can be constructed, debugged, understood, and used most efficiently by human."

Wer verstehen will, wie man (s)ein Forth machen kann, wird bei ihm fündig. mka

Meta–, Cross–, Target–Compiler . . . ?

Beginnen wir mit einem Zitat:

> „This leads to possibilities of fully interactive metacompilation, where words can be compiled one at a time in the target, tested individually, forgotten, and redefined . . . making the metacompiler environment every bit as interactive as a normal Forth system! . . . First Rule of Metacompiler Design: Always keep in mind what the result should look like!" [FD V05N3[2] Rodriguez, R. J.]

So gibt es nicht *den* einen Metacompiler, sondern immer nur *den speziell angefertigten*, der genau das erzeugt, was man im Zielsystem (target) dann haben will.

Metacompiling ist also ein Konzept, das variiert werden muss für den jeweiligen Zweck.

Was die Terminologie angeht, scheint die bis heute in der Forth Community recht konsistent zu sein. Henry Laxen:

> „In one sentence, Meta Compiling in FORTH is a process in which FORTH code is compiled in one environment and executed in another. The environment in which the code is compiled is called the HOST system. The environment in which the code compiled by the Meta Compiler will finally execute is called the TARGET system." [FD V04N6, Techniques Tutorial, Meta Compiling I, Henry Laxen]

Die Begriffe HOST und TARGET haben wohl alle Autoren beibehalten. Die Bezeichnungen für *ihren* jeweiligen Metacompiler hingegen variieren: Forth Inc. nennt es einen *Target Compiler*, Brad Rodriguez' ist ein *Image Compiler*, und *Cross Compiler* hab ich auch schon mal irgendwo gelesen. Mit der Bezeichnung *Cross* wird da besonders betont, dass das neue Zielsystem ja eine ganz andere Hardware hat, mit völlig anderem Befehlssatz als der des Host, weshalb man auch noch einen eigenen Assembler für das neue Target machen muss, damit die basalen Forth–Routinen auch im neuen Forth–Kernel schnell werden.

Also, was wird da nun gemacht? Es sind immer diese vier Schritte:

– man studiere, was erzeugt werden soll (Target)

– man nehme sein Lieblingsforth (Native System)

– damit programmiert man all das, was es braucht (Host), um das Image im Target zu erstellen (und weil man in Forth beim Programmieren ja compiliert, ist das, was man damit dann für das Target compiliert, eben „meta")

– man erzeugt damit ein neues „Image" für das Target.

Beim direkten Zusammenstellen von Maschineninstruktionen spricht man traditionell von „assemblieren", aber „compilieren" wird dafür genommen, wenn man Macros zusammenstellt. Und höhere Programmiersprachen gehen dann dazu über, solche Module in hübsche Anweisungen für ihren darunter liegenden „Compiler" zu verpacken — der „Syntactical Sugar" [Wil Baden].

Solche Compiler–Compiler sind durchaus Trend: modulare Systeme, mit deren Hilfe sich Compiler machen lassen. In Forth ist das ein alter Hut, oder?

mka

Links

Klaus Kohl–Schöpe[3] hat eine Literaturliste dazu beigesteuert. Herzlichen Dank, Klaus.

```
https://arduino-forth.com/article/FORTH_
metacompilation_intro
http://www.ultratechnology.com/meta.html
https://arduino-forth.com/article/FORTH_
metacompilation_intro
https://www.bradrodriguez.com/papers/moving4.
```

[2] Forth Dimension, Volume 05, Number 3

[3] Was immer du zu Forth wissen willst, frag Klaus. Was in seinem Archiv nicht ist, gibt's auch nicht auf der Welt, hab ich so den Eindruck. :)

htm
https://howerj.github.io/embed/meta.htm
https://www.reddit.com/r/Forth/comments/8e4j57/metacompiler_howto_and_small_eforth_interpreter/
http://www.mcforth.net/ – siehe SmallFORTH

Compiler–Compiler: CLANG und LLVM

Bei Texas Instruments ist man derzeit optimistisch bis euphorisch:

> „TI Arm *Clang* is a new set of compiler tools for TI Arm Cortex microcontrollers and represents the future of the TI Arm compiler. This new toolchain is based on the *LLVM* project and uses Clang as the C/C++ front end. Ultimately this new toolchain will help customers produce more efficient applications . . .“ [https://e2e.ti.com/blogs...]

Aha, LLVM?

> „LLVM is an open–source project that is a collection of modular and reusable compiler toolchain technologies. In essence, it is an initiative consisting of building blocks for creating compilers. . . . The LLVM Foundation is a 501(c)(3) nonprofit[4] whose mission is to support education and advancement of the field of compilers and tools through educational events, grants and scholarships, and increasing diversity with the field of compilers, tools, and the LLVM project. We have established 3 main programs: Educational Outreach, Grants & Scholarships, Diversity and Inclusion. . . . “ [https://foundation.llvm.org/]

Eine interessante Sache, so ein Compiler–Compilerbau.
mka

Neues von GreenArrays

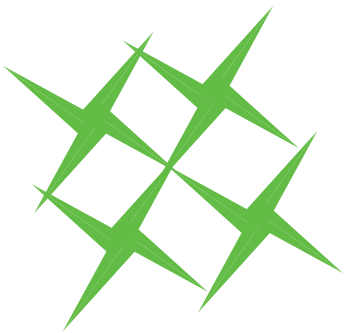


Abbildung 2: Logo der Firma „GreenArrays, Inc.“

In dem Thread: „Ist es Zeit für einen weiteren Forth–Chip?“ auf de.comp.lang.forth hat sich am 24.5.2022 auch Greg Bailey geäußert.

> „Heute Morgen erhielt ich eine E–Mail von Wayne [Morellini] und dachte mir: Schau mal nach, woher er kommt. Das hat mich zu dieser Newsgroup geführt. Nachdem ich den Thread gelesen hab, bemühe ich mich um eine passende Antwort. Ich schreibe genau *einen* Beitrag, der die hoffentlich wichtigen Punkte anspricht. Ich werde nicht an weiteren Diskussionen hier teilnehmen. Wendet euch an greg@greenarraychips.com, wenn ihr Fragen habt. Ich versuche, mich an die Fakten zu halten und Meinungen zu vermeiden:
>
> *Erstens* konnten wir uns noch nicht Vollzeit um die Finanzierung von POLYSANCE kümmern. Die 501(c)(3) [steuerlich begünstigte] gemeinnützige Organisation wurde von uns in Wyoming [Standort von Green Arrays] gegründet. Sie soll laut Satzung jungen Menschen das Programmieren und Entwerfen von Halbleitern in unserem Stil beibringen. Mit jung meinen wir hauptsächlich aufgeweckte Jugendliche, die noch nicht aus der High School heraus sind. (1) Die verstehen bereits, was nötig ist. (2) Denen ist noch nicht eingeredet worden, dass FORTH Mist ist und dass asynchrone Computer unpraktikabel sind. Die Kids werden eine Form von GLOW, dem Nachfolger von OKAD, verwenden. Der [HW–] Prozess wird auf die verfügbare Finanzierung abgestimmt. Sollte jemand von euch jemanden mit einem Haufen Geld kennen, der an eine ehrliche, unpolitische, steuerlich absetzbare Wohltätigkeitsorganisation spenden möchte, macht ihn bitte mit mir bekannt!
>
> *Zweitens* hat OKAD Chips in 180– und 130–nm–CMOS produziert, letzteres mit 5V–I/O. GLOW hat 180–nm–CMOS– und 28–nm–Designs produziert, die alle Designregelprüfungen bestanden haben und erfolgreich simuliert wurden. Der Zeitaufwand, um unsere Low–Level–Designmethoden an neue Geometrien anzupassen, war minimal. Der Aufwand liegt hauptsächlich in den immer bizarreren High–Level–Design–Rules. Die können wegen der Geheimhaltungsvereinbarung mit der Foundry hier nicht diskutiert werden.
>
> *Drittens* sind moderne Prozesse sagenhaft teuer. Jedenfalls bis FinFETs kommen. Auch Leckstrom nimmt erheblich zu. Die meisten von uns können sich die $80k (Preise von anno 2010) für 180–nm–Masken nicht leisten, geschweige denn die $6M6 für 28 nm. Als ich Global Foundries nach den Design Rules für 14–nm–FinFETs fragte, sagten sie mir, sie würden mir erlauben, die zu sehen, wenn ich ihnen 25 Millionen Dollar für einen Shuttle Run [Produktionslauf] bezahle. POLYSANCE wird unter anderem nach einem guten komplementären Prozess suchen, der mit geeignetem Tintenstrahldrucker darstellbar ist. [Gemeint sind eventuell organische Halbleiter, die dann aber wohl erst einmal nur die Komplexität digitaler SSI–ICs[5] erreichen].

[4] Steuerbegünstigtes Unternehmen.

[5] SSI small scale integration; unter 12 Gatter Äquivalente.

Für pädagogische Zwecke ein gutes Werkzeug. GDSII [Calma Format[6] der 70er Jahre, Standard für IC–Layouts], das man über Nacht testen kann, ist besser als sechs Monate warten. Wenn jemand einen Forscher kennt, der ein solches Verfahren im Rahmen seines Studiums aufgezogen hat: seid so nett und bringt diese Person mit mir in Kontakt; persönliches Gespräch ist besser als kalte E–Mails.

Viertens, ein wichtiger Grund, warum die SWATCH–Gruppe [d. h. EM Microelectronic in der Schweiz, die Multi–Project–Wafer anbieten] eine eigene 180–nm–Fertigung besitzt, ist derselbe Grund, warum auch wir diesen Prozess mögen: Solide Leistung, geringer Leckstrom, niedrige Kosten. Wir haben uns mit einer Designgruppe von SWATCH in Colorado Springs getroffen, die Chips für Logitech und andere Kunden entworfen hat, die ebenfalls Wert auf geringe Leckage legen.

Fünftens, ja, es ist so: OKAD und vor allem GLOW *sind* unsere Kronjuwelen. Wir haben diese Tools von Grund auf geschrieben und müssen daher im Gegensatz zu den meisten anderen nicht in der Größenordnung von $100k pro Jahr und Platz zahlen. Für unsere asynchronen Designs ist auch analoge Simulation erforderlich. Wir haben $12k für eine kommerzielle TSpice–Lizenz gezahlt. Die dann vier *Monate* benötigte, um das Verhalten einer Handvoll Transistoren zu simulieren, um die UI–Kurven zu erhalten. Ehedem habe ich gesehen, wie Intellasys $100k pro Jahr für die 1–Platz–Lizenz des analogen ANASIFT–Simulators bezahlt hat. Weiß nicht, wie gut der funktionierte, konnte es mir nicht leisten, es herauszufinden. Unser Simulator läuft schnell genug, um mehrere [GA144–] Nodes beim Ausführen von Programmen zu simulieren, während wir dabei zuschauen. Da liegen Welten zwischen! Wir haben also, was ein Startup braucht, um Chips herzustellen. Und zwar, ohne *mehr als die Kosten für das Silizium* zu bezahlen, um Designsoftware zu mieten. Wir haben alles, was es braucht, um Leuten das Know–how für den Beruf beizubringen. Ohne, dass jemand wie Mentor Graphics uns kostenlose Lizenzen geben muss, damit wir Leuten beibringen, wie sie deren Tools zu bedienen haben.

Sechstens können unsere [GA144–] Chips sehr wohl SDRAM ansteuern. John Ribles Code dafür ist vorhanden und zwar im ROM in den Node 7, 8, 9, 107, 108. Der Quellcode für diese ROMs ist in der kostenlosen Distribution von arrayForth–3 ab Block 2241 enthalten. Man muss da nur reinschauen.

Siebtens verwenden wir routinemäßig externe Taktgeber (Quarze) aus genau den zuvor in diesem Thread genannten Gründen, wenn die Anwendung eine genau spezifiziertes Timing erfordert. Siehe zum Beispiel AN002 und AN012 [Application Notes GA144].

Achtens, jemand [im clf–Thread] sagte, er habe uns nach Timing–Details zu den Speicherschnittstellen–Nodes gefragt im Zusammenhang mit dem Wunsch, DRAMs anzusteuern. Und ihm sei geraten worden, einfach zu probieren. Ich war die ganze Zeit hier, schon als wir GreenArrays, Inc. hießen, und kann mich nicht an eine solche Anfrage erinnern. Ich weiß nicht, wer diese Antwort gegeben hat; derjenige muss zu diesem Zeitpunkt zumindest abgelenkt gewesen sein. Ich entschuldige mich also, sollte GreenArrays diesen unpassenden Ratschlag gegeben haben. Wer heute per E–Mail bei `hotline@greenarraychips.com` nachfragt, erhält bessere Antworten. Denn es gibt beim EVB001 [Evaluation Board GA144] Testpunkte auf vier zeitkritischen Leitungen der Speicherschnittstelle, die solche Untersuchungen vereinfachen.

Abschließend lädt GreenArrays euch zum konstruktiven Dialog direkt per E–Mail ein. Wir haben früh festgestellt, dass wir nicht so lange leben, um Zeit für Chat–Gruppen zu haben. Ich habe `comp.lang.forth` lange genug gelesen, um das einzusehen. Alles Gute euch allen! Und Grüße an Anton [Ertl] und andere hier, die ich seit vielen Jahren kenne.

Greg Bailey, Präsident, Greenarrays, Inc. `https://www.greenarraychips.com/`“

In eckigen Klammern stehen Anmerkungen zum besseren Verständnis dessen, worauf Greg sich jeweils bezieht.

Und die folgende Liste gibt einen Überblick über die historischen Prozessoren von Chuck Moore und wann GreenArrays dazu gekommen ist.

Chuck Moores Chips

Novix NC4016

1983 Fab: Mostek, 4000 (geschätzt) Gates, 3 µm CMOS — Moore „several hundreds boards sold“ [1].

Harris RTX2000 d. h. redesign Novix

1988 Fab: Harris, Standardzelle, 2 µm CMOS, vermutlich >1k produziert.

Shboom und das Resdesign PSC1000 JAVA–Prozessor

1988 Fab: Oki, 8000 Gates, 1,2 µm CMOS Samples.

[6] GDSII (ursprüngl. Graphical Design Station II oder Graphic Data System II) bezeichnet das De–facto–Standard–Datenformat für Layoutdaten von integrierten Schaltkreisen im EDA–Bereich. (Electronic Design Automation (EDA) bezeichnet Software für den Entwurf von Elektronik, insbesondere Mikroelektronik. Es ist ein Teilgebiet des Computer–Aided Design (CAD))

1999 Patriot Scientific, Beginn der PSC1000 Produktion.

2002 Patriot sucht Investoren für PSC1000, ca. Ende Fertigung

2005 AMD erwirbt Lizenz, Patriot & Moore sammeln ingesamt von mehreren Halbleiterherstellern $125M Lizenzgebühren für ShBoom–Patente ein.

Generation „21“

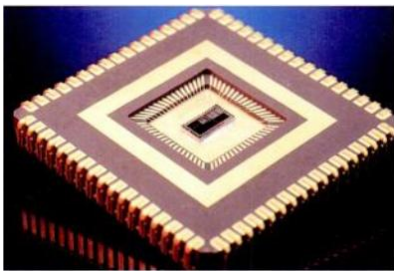


Abbildung 3: Der i21–Prototyp

1990 MuP21 Fab: Orbit, Fab: Hewlett Packard / Mosis — 12. . . 25 Muster–Chips für $3k. . . $6k bei 2 Monaten Durchlaufzeit. [1] [MOSIS war der Musterservice von DARPA/Pentagon. Anfangs verteilte er Fertigung auf 11 Fabs, aber ab 1989 eigentlich nur noch auf Hewlett-Packard, Orbit, IMP und VLSI Technology.]

1993 MuP21, Offete (= Dr. Ting), 1,2 µm CMOS Produktion zu $25/1 DIL40 [2] — Moore: „Some 20k were manufactured“.

1996 iTVc i21 Muster „400 MIPS processor“ — Mit Unterstützung NASA / Ames Research Center und $0,9M gegründet. Pro forma für Anwendung von NASA.

1997 iTv Corporation erhält mehrere Mio$ Venture Capital. Ziel ist nun ein ultra low cost Internet Access Terminal. D. h. BTX auf amerikanisch. Fabs wären Korea/Japan gewesen [3]. — Neben den technischen Problemen hat das Platzen der Dotcom–Blase 2000 das Projekt beendet.

1998 F21, UltraTechnology (= Jeff Fox), Fab: HP / Mosis 0,8 µm Muster.

GreenArrays

2009 Gründung des Unternehmens

[7] integrated circuit

[8] ASIC: application–specific integrated circuit.

Abbildung 4: G144A12 Chip

[1] Bergin, „History of Programming Languages“, Addison Wesley, 1996
[2] `www.ultratechnology.com/cowboys.html`
[3] „Spinoff 1998“, NASA–Broschüre.

Rafael Deliano

IC–Designs

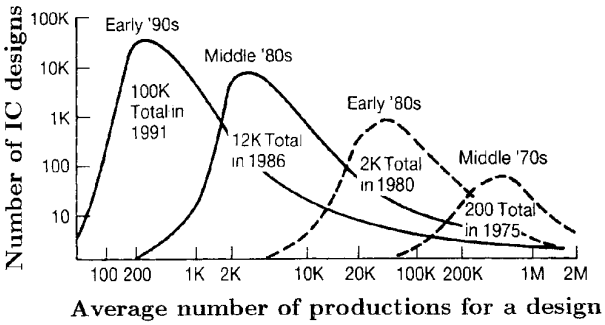


Abbildung 5:

Wenn, wie in Abb. 5 dargestellt, historisch die Zahl der Designs steigt und die typischen Stückzahlen sinken, warum ist es dann so schwer, ein IC[7] zu entwickeln? Die Chips mit den kleinen Stückzahlen sind ASICs[8], auf bestimmte Anwendung und einen Kunden zugeschnitten. Den Mikroprozessor als Standardbauteil am freien Markt im Stil der 70er Jahre gibt es fast nicht mehr. Das Marktvolumen ist zwar größer geworden, aber die Fertigung von ICs wurde nicht billiger. Kleine Stückzahlen waren bei dem ehedem höheren Preisniveau gangbarer. Indikator für die wirtschaftlichen Gegebenheiten ist, dass sich die Zahl der IC–Hersteller durch Fusionen kontinuierlich vermindert hat.

Anmerkung zu Abb. 5: J. H. Lau zählte wohl auch Masken–ROMs in Controllern als Design.

Quelle: Ball Grid Array Technology (Electronic Packaging and Interconnection Series), 1995, englische Ausgabe von John H. Lau (Herausgeber). Rafael Deliano

CH56x, Mecrisp Quintus und ein Board dazu

> „Hi, I thought this would be interesting for you, maybe you could include it in the newsletter: I created a working port of Mecrisp Quintus for the WCH–Tech CH56x series of USB3 microcontrollers:
> `https://github.com/hansfbaier/mecrisp-quintus`
>
> I also designed an open–hardware board with that chip:
> `https://github.com/hansfbaier/ch569-usb3-board`
> It can be cheaply manufactured on demand by JLCPCB, complete production files are included.
>
> Best regards, Hans Baier“

Und ob! Das sind sehr interessante RISC–V MCUs, welche da von der Design–Company *WCH Nanjing Qinheng Microelectronics* in China gefertigt werden, „full stack“. `www.wch-ic.com`.[9]

Anmerkung

Neugierig geworden, wollte ich wissen, was diese Firma so alles macht. Sie entwerfen MCUs, MCU–Networking, USB Connections und Charging Protocolls.

> „Das Unternehmen betreibt ein Full–Stack–Entwicklungsmodell, das auf selbst entwickelten Transceiver–PHYs und Prozessor–IP basiert, anstatt herkömmliche IP–Integrationsmodelle zu nehmen.“

In der Korrespondenz mit Bernd Paysan war zu erfahren, dass ihm als Chip-Designer das alles „völlig klar wie Klosbrühe“ war, was sie damit meinten. Mir aber nicht. So habe ich nun gelernt, dass PHY die physikalische Schicht der Transceiver ist, und IP die „Intellectual Property“ fertiger MCU–Blöcke, die man in Lizenz für sein Design dazukaufen kann. Und das mit „Full Stack“ die Leute, die Hardware designen, sich damit auf ihren kompletten Schaltungsentwurf beziehen. Sie meinen, dass sie alle Schaltpläne und Designs selber machen, also sowohl analog als auch digital, und keine Komponenten hinzukaufen, außer denen, die Teil des Design–Kits sind, den der Chip–Fertiger zur Verfügung stellt. Und das sind ganz basale Dinge, einfache analoge Komponenten wie Widerstände, Transistoren oder Kapazitäten und einfache digitale Komponenten, wie Gatter oder SRAM–Zellen bzw. Arrays. Irgendwie forthig, oder?

Ansässig ist die Firma übrigens in No. 18, Ningshuang Road, Nanjing, China — laut GoogleMaps im: 新中心4幢, „Innovationszentrum, Gebäude 4“. mka

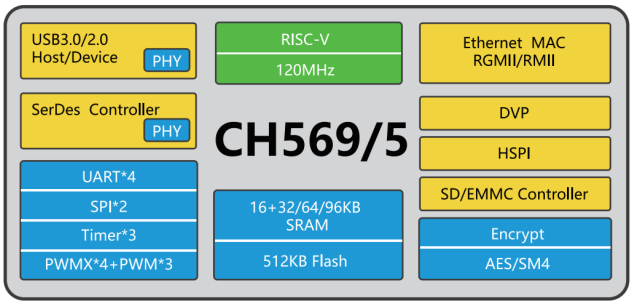


Abbildung 6: CH56x series of USB3 microcontrollers

`floor` — integer oder integral ?

Beschäftigt man sich mit Zahlenkonversionen, stolpert man früher oder später auch über `fmod` und will dann wissen, was es tut. Gforth `locate fmod` liefert Folgendes:

```
: fmod ( r1 r2 -- r ) \ modulus of r1/r2
  fover fover f/ floor f* f- ;
```

Gut, da passiert einiges auf dem *flointing point stack*, dem ich folgen kann, aber was macht das `floor` dort? Ein Blick in den Quellcode zeigt:

```
floor ( r1 -- r2 ) float
""Round towards the next smaller integral value,
  i.e., round toward negative infinity.""
CLOBBER_TOS_WORKAROUND_START;
r2 = floor(r1);
CLOBBER_TOS_WORKAROUND_END;
```

Aha, ein Primitive des Forth, Maschinencode, in C.

```
123.456e0 floor cr f. <enter>
123.
```

Man sieht sogleich, was es macht: runden auf die nächst kleinere ganze Zahl.

Doch was meint „integral“ da im Kommentar? Anton Ertl wies mich auf die Bedeutung hin, die das *wiktionary* nennt. Danach wurde die heutige englische Bedeutung ausgeliehen aus dem französischen *integral*, die es wiederum vom lateinischen *integer* im Sinne von „vollständig“ übernommen hatte. Die beiden Begriffe liegen, was den sprachlichen Wortsinn angeht, gleichauf. Aber in deiner Rechenmaschine in Gforth liegen Welten dazwischen!

So bezeichnet der Begriff „integer“ in der Computerei die ganzen Zahlen vom „single–cell type“ und „integral“ meint eine ganze Zahl, hinterlegt im Gleitkommaformat. Und die Handhabung dieser Zahlentypen erfolgt in Gforth auf verschiedenen Stacks! Die *Integer–Zahlen* werden auf dem Daten–Stack jongliert, die *integralen Zahlen* hingegen auf dem Floatingpoint–Stack. Man muss also wissen, auf welchem Stack man operiert. Daher finde ich es sehr sinnvoll,

[9] Das Mecrisp Quintus und weitere Forthe von Mattias Koch findest du dort:
`https://sourceforge.net/projects/mecrisp/files/`

das auch sprachlich auseinanderzuhalten, so, wie es in der Gforth–Quelle auch erfolgt ist.[10]

Der Screenshot (Abb. 7) zeigt meine Fingerübung dazu. Man sieht da schön, dass Gforth so nett ist, hinter dem `ok` auch anzugeben, was auf welchem Stack liegt. Die `1` hinter dem `ok` bedeutet: „Ein Item ist auf dem Datenstack“. Und die `1` hinter dem `f:` bedeutet: „Ein Item ist auf dem Floatingpoint–Stack“ — hübsch, oder?

Was ergibt dann die folgende Eingabe?

```
123.345e0 10 fmod
```

Ja, genau, eine Fehlermeldung!

```
*the terminal*:9:14: error:
Floating-point stack underflow
123.345e0 10 >>>fmod<<<
```

Denn `fmod` braucht den Teiler `10` bei sich auf dem *floating point stack*. Einfach so eingegeben, packte die Ziffernkonversion die `10` aber auf den Datenstack, so geht's also nicht. Es braucht `10 s>f` , um die 10 rüber auf den anderen Stack zu schieben, oder eben gleich die Eingabe `10.00e0` an jener Stelle. mk

Abbildung 7: Integer vs. integrale Zahl

Aktualisiertes VolksForth für den Commander X16

Der *Commander X16* — jener neue, Commodore–inspirierte 65C02–Computer (siehe VD 3/2021 S. 32 ff.) — hat im Frühjahr 2022 ein neues Prototyp–Board bekommen, das mit dem ersten Prototypen inkompatibel ist, und parallel dazu drei neue Versionen des Betriebssystems — „Kernal“ in Commodore–Sprech. Eigentlich hätte das bisherige X16–VolksForth, Version 3.9.3, mit dem alten wie dem neuen Board funktionieren sollen, und mit einer Pre–Release–Version des ersten „neuen“ Kernals, R39, tat es das im passenden Emulator auch.

Aber dann kamen die Kernal–Versionen R40 und R41 daher und zeigten: Adressen von Kernal–Variablen im RAM, die beim C64 und beim C16 noch völlig unveränderliche und verlässliche Größen waren — und von VolksForth genutzt wurden und werden — gehören eben doch nicht unbedingt zur API–Definition und können sich bei einem System wie dem X16, das sich noch in Entwicklung befindet, von Version zu Version verschieben.

Im Laufe des Juli und August habe ich VolksForth zunächst an den aktuellen R41–Kernal angepasst und anschließend die Verwendung von Kernal–Variablen soweit wie möglich durch API–Aufrufe ersetzt. Ein einziger Zugriff auf eine Variable ist noch geblieben, und die hat mit einem OS–Bug zu tun, von dem ich glaube, dass er schon seit mindestens 40 Jahren in diversen Commodore–Computern existiert. Für CBM–Geeks: Zu Beginn der Kernal–Routinen `LISTEN` und `TALK`, mit denen man den IEC–Bus direkt nutzen kann (VolksForth tut das), wird das I/O–Status–Byte (`$90` beim C64) nicht gelöscht, sodass man das vorher tun muss. Und dafür gibt es keinen Kernal–API. Mal sehen, ob ich dafür im X16–Kernal einen Fix bekommen kann. Die Diskussion ist angestoßen.

Inzwischen ist die aktualisierte VolksForth–Version 3.9.4 auf der X16–Website veröffentlicht und ins Haupt–GitHub–Repo integriert. Das Gleiche gilt für Version 0.11 meines Small–C–Compilers cc64, der ja auf VolksForth aufsetzt. Philip Zembrod

Links:

```
https://www.commanderx16.com/forum/index.php?
/topic/4467-volksforth-and-cc64-updated-to-
prototype-2-and-r41-kernal/
https://www.commanderx16.com/forum/index.php?
/files/file/84-volksforth-x16/
https://www.commanderx16.com/forum/index.php?
/files/file/121-cc64-x16/
https://github.com/commanderx16/x16-rom/issues/
333
https://github.com/forth-ev/VolksForth/tree/
master/6502/C64
https://github.com/pzembrod/cc64
```

Weitere Meldung auf S. 34

[10] Anton hat auch in den man–Pages von C–Funktionen nachgeschaut, die sowas machen, und fand zu `floor`: These functions return the largest *integral* value that is not greater than x. Und bei `rint`: The nearbyint(), nearbyintf() and nearbyintl() functions round their argument to an *integer* value in floating–point format, using the current rounding direction. Hm, da wurde mal so und mal so gesagt. Gforth folgt lieber dem Forth–Standard, und der macht diese Unterscheidung auch sauber.

RISC–V
Assembler, Disassembler und Simulator

Klaus Kohl–Schöpe

Letztes Jahr hat mich die Feuerstein–Gruppe — speziell Wolfgang Strauss *— durch die Zusendung des Longan Nano Boards dazu „gezwungen“, mich mit diesem RISC–V Prozessor und dem Mecrisp Forth darauf zu befassen. Da ich auch als Mikrocontroller–Spezialist immer mehr mit RISC–V, z. B. in FPGAs, zu tun habe, nutzte ich diesen Anreiz und habe in den folgenden Wochen meine schon angefangene Opcode–Liste fertiggestellt, Assembler und Disassembler realisiert und um einem Simulator ergänzt. Dieser erlaubt mir sogar, das Mecrisp für RISC–V auf einem PC laufen zu lassen.*

Der RISC–V Assembler

Wenn man sich mit Forth auf einem neuen Prozessor befasst, sollte man die Hardware, aber auch den Opcode des Prozessors, kennen, da dies die Grundlage für einen Assembler ist. Deshalb erstelle ich mir immer eine Liste der Befehle mit zugehöriger Adressierungsart und dem Opcode. Beim RISC–V sind es zwar nur wenige Befehlstypen, aber leider werden die Bits für Immediate bzw. Offset ziemlich gewürfelt, um angeblich optimiert für FPGAs zu sein (Tab. 1).

RISC–V ist Low–Endian und holt sich deshalb das niederwertigste Byte zuerst und kann anhand der ersten 2 bzw. 7 Bits erkennen, ob es sich um ein 16–, 32–, 64– oder 128–Bit–Befehl (geplant bis 864 Bit) handelt.

Mein Assembler in Forth soll z. B. folgendes Programm kompilieren können und dabei auch darauf achten, dass gültige Register und Parameter verwendet werden:

```
Code test
   x3 , x4 , x5 add,  \ x3 <= x4 + x5
   x3 , x6 , 1$ beqz, \ Jump wenn x3 == x6
   x3 ,( #4 x4 lb,    \ x3 <= Byte aus(x4+4)
  1$:                 \ Label
   x1 c.jr,           \ Return
 End-Code
```

Das bedeutet, die Register (32 mal 32 Bit), Adressierungsarten (angelehnt an Standard–Assembler) und Offsets müssen geprüft werden. Forth–like kommen zuerst die Parameter, bevor der Befehl mit Komma folgt, da er ja den Befehl speichert. Wie man hier sieht, gibt es das Komma auch für die Trennung der Parameter, angelehnt an den Standard–Assembler mit einigen Modifikationen wie `,(` für Register mit Offset. Es können auch innerhalb eines Befehls bis zu 16 Labels `1$:` bis `16$:` definiert und z. B. für Sprünge als `1$` bis `16$` verwendet werden.

Ich verwende in einem Assembler meist Create–Does–Konstruktionen zur Befehlsdefinition. Dies schaut bei Befehlen ohne Parameter wie folgt aus:

```
: NoPars:       ( opc -- ; -- ) \ no parameter
  Createp ,p
  Doesp>  >r  ?nopar?  r> @p  ,p  reset ;

  $0000100f     NoPars:         fence.i,
  $0ff0000f     NoPars:         fence,
...
```

Diese merken sich den 32–Bit–Opcode, prüfen bei Verwendung des Befehls, dass keine Parameter da sind und kompilieren dann den Opcode ins Programm.

Ähnlich, wenn auch mit deutlich mehr Tests, geht es dann bei den anderen Befehlen weiter:

```
: R-Type:  ( opcode -- ; -- ) \ rd , rs1 , rs2
  Createp ,p
  Doesp>  >r  $f00c0300 $10040100 ?reg?
          r> @p  r3@ +rd  r2@ +rs1
                 r1@  +rs2  ,p   reset ;

  $00000033     R-Type:         add,
  $40000033     R-Type:         sub,
...
```

Um schnell auf alle Parameter prüfen zu können, schieben die trennenden Wörter wie `,` bzw. `,(` und `Opcode,` Informationen, wie Adressierungsart, Registernummern bzw. Offset in Variablen, die mit einer einfachen AND– und Gleich–Maske mittels `?reg?` getestet werden können. Ist alles richtig, wird der gespeicherte Opcode um Registernummern oder Offset ergänzt, wobei man auf die Bitreihenfolge und die bei RISC–V meist vorzeichenbehafteten Werte achten muss. Wer sich den Source–Code ansieht, wird bemerken, dass genau dies einen Großteil ausmacht und auch gründlich getestet werden musste.

Der RISC–V Disassembler

Bei einem Disassembler (siehe Auszug aus RISC–V Disassembler) geht man den umgekehrten Weg und holt sich den nächsten (32–Bit) Opcode aus dem Speicher und kann mit Hilfe der niederwertigsten 2 Bits erkennen, ob ein 16–Bit–Befehl genutzt wird. Beim RISC–V habe ich eine Tabelle für 16–Bit und zwei für 32–Bit verwendet, bei denen auf eine Maske verglichen wird (AND dann =). Wurde der Befehl in den Listen gefunden, wird der Opcode–String und dann die Parameter durch Aufruf des Befehls, wie hier `rd,imm31_12_u` , oder `NOP` , wenn ohne Parameter, ausgegeben. Wie schon erwähnt, muss man vor allem auf diese gewürfelten Bits in den Immediate–Werten oder Offsets achten. Ein Sonderfall ist hier $FFFF

Mnemonics		Opcode																														Operation
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	imm is signed, uimm is unsigned
R-Type	**<rd>,<rs1>,<rs2>**	**funct7**							**rs2**					**rs1**					**funct3**			**rd**					**op**							**op = xxbbb11 with bbb<>111**
ADD	<rd>,<rs1>,<rs2>	0	0	0	0	0	0	0	rs2					rs1					0	0	0	rd					0	1	1	0	0	1	1	rd = rs1 + rs2
SUB	<rd>,<rs1>,<rs2>	0	**1**	0	0	0	0	0	rs2					rs1					0	0	0	rd					0	1	1	0	0	1	1	rd = rs1 - rs2
I-Type	**<rd>,<rs1>,imm**	**imm[11:0]**																	**rs1**			**funct3**			**rd**					**op**				**op = xxbbb11 with bbb<>111**
S-Type	**<rs2>,imm(<rs1>)**	**imm[11:5]**							**rs2**					**rs1**					**func3**			**imm[4:0]**					**op**						**op = xxbbb11 with bbb<>111**
SB-Type	**<rs1>,<rs2>,imm**	**imm[12,10:5]**							**rs2**					**rs1**					**funct3**			**imm[4:1,11]**					**op**						**op = xxbbb11 with bbb<>111**
BEQ	<rs1>,<rs2>,imm	imm[12,10:5]							rs2					rs1					0	0	0	imm[4:1,11]					1	1	0	0	1	1	PC = PC+imm<<1 if rs1=rs2
U-Type	**<rd>,<rs1>,<rs2>**	**uimm[31:12]**																				**rd**					**op**							**op = xxbbb11 with bbb<>111**
UJ-Type	**<rd>,imm**	**imm[20]:imm[10-1]:imm[11]:imm[19-12]**																				**rd**					**op**							**op = xxbbb11 with bbb<>111**
FENCE.I		0				0				0				0					0	0	0	0					0	0	0	1	1	1	Sync write before instruction read
FENCE		0				PI	PO	PR	PW	SI	SO	SR	SW	0					0	0	0	0					0	0	0	1	1	1	Sync write before instruction read
CR-Type	**<rd>,<rs2>**	---																	**func4**			**rd/rs**					**rs2**					**op**	
CI-Type	**<rd>,imm**	---																	**func3**		**i**	**rd**					**imm**					**op**	
C.LI	<rd>,imm	---																	0	1	0	i5	rd<>0				i4:0					0	1	Load immediate
CS-Type	**<rd>,imm**	---																	**func3**			**uimm**		**rd'**			**uimm**		**rs2'**			**op**	
CSS-Type	**<rd>,<rs2>**	---																	**func3**			**uimm**							rs2'			**op**	
CIW-Type	**<rs2'>,imm**	---																	**func3**			**uimm**							**rs2´**			**op**	
C.ADDI4SPN	<rs2'>,sp,uimm	---																	0	0	0	i5:4|9:6|2|3							rs2'			0	0	Add uimm (*4) to SP (uimm <> 0)
CL-Type	**<rs2'>,imm**	---																	**func3**			**uimm**		**rs'**			**uimm**		**rs2'**			**op**	
CB-Type	**<rd>,imm**	---																	**func3**			**offset**		**rd'/rs'**			**offset**					**op**	
C.BEQZ	<rs'>, addr	---																	1	1	0	o8|4:3		rs'			o7:6|2:1|5					0	1	Branch if rs' is 0 (offset*2)
CJ-Type	**offset**	---																	**func3**			**jump offset**										**op**	

Tabelle 1: Auszug aus RISC-V (RV32IMC) Opcodes

(für ein leeres Flash), das einfach als 16–Bit–Wert ausgegeben wird. Da der Opcode öfters gebraucht wird, hole ich ihn mit `opcode@` aus dem Speicher und lege diesen in der Variable `opcode` ab. `Createp` mit `End-Create` und Trennung in mehreren Listen ist wegen der Verwendung von *mcForth* notwendig, da in dieser Version nur maximal 512–Byte–Blöcke pro Befehl beim Flash erlaubt sind.

Der RISC–V Simulator

Zum besseren Test des geplanten mcForth für RISC–V habe ich auch noch den Simulator realisiert. Wer den Source–Code von Simulator (siehe Auszug aus RISC–V Simulator) und Disassembler ansieht, wird viele Ähnlichkeiten erkennen, da ein Simulator auch die einzelnen Registernummern und Parameter nutzen muss, um die Daten für die Verarbeitung aus Registern oder Speicher zu holen bzw. wieder zurückzuschreiben. Da aber normalerweise nichts ausgegeben wird (das macht der Disassembler), ist die verwendete Tabelle kompakter und nutzt auch Maske und Vergleichswert zur Erkennung des Befehls, der dann direkt aufgerufen wird. Dieser holt sich dann die entsprechenden Parameter, führt die Operation aus und schreibt die Daten wieder zurück. Der Auszug zeigt dies am Beispiel des Load–Byte bzw. Load–Halbwort (16 Bit). Für den Speicherzugriff gibt es die Wörter `c@s` , `cs@s` , `hs@s` ... `@s` bzw. `c!s` , `h!s` und `!s` jeweils für Byte, Halbwort und Wort, evtl. mit Sign–Extension. Diese Wörter müssen dann modifiziert werden, wenn das Programm andere Speicherbereiche nutzt oder falls man die Hardware für einen RISC–V Prozessor wie dem GD32F103CB vom Longan Nano simulieren will. Der Simulator für Mecrisp auf dem Longan Nano benötigte z. B. nur USART0–Status– und Datenregister.

```
$40013800 = Status
     (Bit 5 (RXNE): RX?;
      Bit 7 (TBE): TX?)
$40013804 = Data
```

Das Gute an einem Simulator ist, dass man alle Aktionen und Parameter ausgeben kann, und so kann man ansehen, wie und auf welche Adressen Mecrisp zugreift. Dabei ist mir aufgefallen, dass, obwohl im Users–Manual explizit nur 32–Bit–Zugriffe auf den USART erlaubt sind, Mecrisp einen *Byte*zugriff nutzt. Es war aber kein Problem, einfach alle Byte–, 16–Bit– und 32–Bit–Zugriffe um die Abfrage für diese Register zu ergänzen und eine Erweiterung für den Simulator zu realisieren, damit Tastatureingaben auf diese Register abgelegt, verfügbare Zeichen ausgegeben und dabei die `RX?` bzw. `TX?` Flags genutzt werden.

Für einen kompletten Target–Simulator des Chips GD32VF103CB fehlen noch ein paar Zutaten, wie externer Simulator–Speicher für 128 KByte Flash (ab $0000.0000) und 32 KByte RAM (ab $2000.0000). Zusätzlich noch Befehle zum Laden, Disassemblieren und Debuggen von Programmen. Hier die Befehlsliste, die ich im *RV32IMC_ TSIM* nutzen kann:

siminit (--) Reset RISC–V register

read (file ; --) Read file

dis (addr -- addr2) Disasm 1 line

ndis (addr n -- addr2) Disasm n lines

d (--) Show next command

s (--) One step

steps (n --) n steps

debug (--) Single step debug

- CR : Jump over (execute call)
- C : Input one character
- ESC: Stop debugger

run (--) Run till ESC or error

r_pc (pc --) Run till breakpoint

d_run (--) Trace

d_pc (pc --) Trace till breakpoint

x (--) Exit simulator

Damit brauche ich nur noch den Simulator mit `sim` zu starten, das Binary des Mecrisp mit `read mq.bin` zu laden und mit `run` zu starten. Zusätzlich gibt es noch die Variable `info` , die ich im Simulator nutze, um zusätzliche Informationen auszugeben. Dessen Bedeutung sollte man besser im Sourcecode anschauen. Mit der `ESC`–Taste kann ich dann den Simulator stoppen und mit Befehl `x` verlassen, wobei der reservierte Speicher wieder freigegeben wird.

Ich hoffe, dieser Artikel ist ein Anreiz, sich auch mal mit einem Simulator, Assembler bzw. Disassembler in Forth zu befassen. Die hier angegebenen Listings und Programme sind über `mcforth.net` und GitHub verfügbar.

Und nun viel Spaß damit. Probiert es aus und teilt mir gerne mit, wie euer Eindruck ist.

Anhang: FPGA Boards

Ich verwende hierzu meist ein Intel Max 10 FPGA bzw. deren Cyclone 10 LP FPGA–Reihe auf den Boards MAX1000 und CYC1000. Das sind günstige MKR–Format Boards, entwickelt von Arrow und Trenz. Und das Terasic DE10 Lite Kit mit einem 50 kLE MAX 10 darauf. Außerdem gibt es die RISC–V Hard–IP für Microchip's Polarfire (ICICLE–Kit). Neben Microchip haben auch Lattice und Intel (NIOS–V) ihre RISC–V IP. (Als Jäger und Sammler habe ich natürlich alle Kits hier im Büro. :)

Beim FORTH2020 ZOOM–Meeting #26 am 13. August 2022 habe ich davon berichtet. Ihr könnt euch das gerne bei YouTube ansehen, der Link ist angefügt. Oder holt euch das PDF der Präsentation.

Links

`http://mcforth.net` (siehe mcForth Beispiele)

`https://github.com/KlaKoSch` (siehe RISC–V)

`https://www.youtube.com/watch?v=3GM7S30VR50` (ab 1:10h)

`https://github.com/forth2020/zoom-presentations/raw/main/assets/2022-08-13/20220813_FPGA_Forth.pdf` (Präsentation)

Listings

Auszug aus RISC–V–Disassembler (RV32IMC)

```
Createp rv32i1_table
\ mask          opcode        mnemonics (12char)  operands
  $0000707f ,p  $00000003 ,p  s" lb        " $,p  ' rd,imm11_0_i(rs1)  ,p
  $0000707f ,p  $00001003 ,p  s" lh        " $,p  ' rd,imm11_0_i(rs1)  ,p
...
  $00000000 ,p
End-Create

Createp rv32i2_table
  $0000007f ,p  $00000037 ,p  s" lui       " $,p  ' rd,imm31_12_u      ,p
  $0000707f ,p  $00000063 ,p  s" beq       " $,p  ' rs1,rs2,imm12_1_sb ,p
...
  $00000000 ,p
End-Create

Createp rv32c_table
\ mask         opcode       mnemonics        operands
\ bit 16=1 if rd<>0; bit 17=1 if rs2<>0
  $0000ffff ,p  $00000000 ,p  s" unimp     " $,p  ' nop                 ,p
  $0000e003 ,p  $00000000 ,p  s" c.addi4spn" $,p  ' rs2',sp,uimm9_2_ciw ,p
...
  $00000000 ,p
End-Create

...
: findop    ( opcode table -- 0 | entry -1 ) \ Find opcode in table
  BEGIN  dup @p                              \ end of table ?
  WHILE  over  over @p  and   over 4 +  @p  = \ compare mask opcode with table
         IF  nip  -1  exit THEN              \ found
         #24 +                               \ next entry
  REPEAT drop  drop  0 ;                     \ not found
...
: dis32         ( addr -- addr+4 )
  cr  addr.  opcode32.
  opcode @  rv32i1_table  findop
  IF   8 +  dup countp typep space   #12 +   @p execute  4 +  exit THEN
  opcode @  rv32i2_table  findop
  IF   8 +  dup countp typep space   #12 +   @p execute  4 +  exit THEN
  ." unimp " 4 + ;
...
: dis           ( addr -- addr+x )
```

```
\ dup 1 and IF 1 + THEN \ !!! tests for aligned address !!!
  base @  >r  hex
  dup ( here + ) \ !!! expect program address 0 at here !!!
  opcode@   dup opcode !
  dup $ffff and  $ffff =             \ special case $ffff
  IF   drop  cr  addr.  opcode16.  ." 0xffff "  2 +
  ELSE 3 and  3 =   IF dis32 ELSE dis16 THEN
  THEN r>  base ! ;

: ndis          ( addr n -- addr+x )
  FOR dis NEXT ;
```

Auszug aus dem RISC-V Simulator (RV32IMC)

```
: imm11_0_i(rs1) ( -- addr ; imm = [31:20] => [11:0] )
  imm11_0_i  rs1@  + ;
...

\ execute 32-bit opcode ( pc on stack )
: i.lb                  imm11_0_i(rs1) cs@s   rd!  4 + ;
: i.lh                  imm11_0_i(rs1) hs@s   rd!  4 + ;
...

Createp rv32i1_table
\ mask          opcode        command
  $0000707f ,p  $00000003 ,p  ' i.lb         ,p
  $0000707f ,p  $00001003 ,p  ' i.lh         ,p
...
\ End
  $00000000 ,p
End-Create

: sim32         ( addr -- addr2 0 / addr -1 )
  opcode @  rv32i1_table  findop
  IF 8 +  @p execute  0 exit THEN
  opcode @  rv32i2_table  findop
  IF 8 +  @p execute  0 exit THEN
  -1 ;
...

: sim           ( addr -- addr+x f ) \ 0=ok; -1=unimpl
\ dup 1 and IF 1 + THEN \ !!! tests for aligned address !!!
  base @  >r  hex
  dup ( here + ) \ !!! expect program address 0 at here !!!
  @s   dup opcode !
  dup $ffff and  $ffff =             \ special case $ffff
  IF   -1
  ELSE 3 and  3 =   IF sim32 ELSE sim16 THEN
  THEN r>  base !  over regs ! ;

: nsim          ( addr n -- addr+x )
  FOR sim IF rdrop exit THEN NEXT ;
```

ASCII–Art fürs Analoge: Das Signallabor

Matthias Koch

Während es für kleine Versuche mit analogen Sensoren meist ausreichend ist, einfach die eingelesenen Werte im Terminal auszugeben und mal drüberzuschauen, ob es passt, stellt sich bei komplizierteren analogen Schaltungen und verrückten Signalen immer wieder die Frage: Was messe ich denn dort eigentlich, und wie kann ich es justieren? Amplitude, Offset, Clipping, Verteilung des Rauschens, auftretende Frequenzen … All das will gründlich bedacht sein. Viel zu oft treten außerdem Störungen auf, die nur mit dem Lötkolben behoben werden können, bei denen der Mensch in der Standardausführung mit nur zwei Händen also zwischen Löten, Aufnehmen von Probedaten und Auswerten hin und her wechseln muss.

Um den aufwändigen Feinschliff am Analogen zu unterstützen, habe ich ein kleines Signallabor entwickelt und mir Gedanken gemacht, welche Arten der Signaldarstellung im Terminal besonders hilfreich sein könnten. Insbesondere eine Herausforderung tauchte dabei immer wieder auf: Wie lassen sich große Datenmengen eigentlich kompakt und übersichtlich darstellen, damit der Mensch freihändig an der Elektronik basteln kann?

Das Signallabor

Insgesamt unterstützt das Signallabor, welches übrigens fertig zum Ausprobieren in *Mecrisp–Ice 2.6c* fürs *ULX3S* enthalten ist, momentan vier verschiedene Ausgaben:

- *Oszilloskop–Ansicht*, linear oder logarithmisch, wahlweise skaliert auf Minimum und Maximum des Datensatzes (zur detailierten Betrachtung des Signals selbst) oder mit dem gesamten vom Wandler abgedeckten Wertebereich (zur Betrachtung von Amplitude und Offsets und zur Justage des Signalpfades).
- *Binning*, als Tabelle ausgegeben, um zu sehen, welcher Messwert wie oft vorkommt. Dies ist besonders nützlich, um die statistische Verteilung des Rauschens um ein zeitlich konstantes Eingangssignal näher unter die Lupe zu nehmen.
- *Fourier–Ansicht*, sinnvollerweise logarithmisch, um die im Signal vorhandenen Frequenzanteile zu betrachten.
- Und schließlich das *Live–Binning*, welches fürs freihändige Justieren prima geeignet ist. Dafür werden schnell hintereinander immer wieder Aufnahmen gemacht und kompakt in einer einzigen Zeile ausgegeben, wobei Durchschnittswert, Minimum, Maximum und die Verteilung der Messwerte über den Gesamtbereich des Wandlers hinweg übersichtlich dargestellt werden.

Beispiele für die Vorstellung oder was uns erwartet

Während die ersten drei Möglichkeiten vermutlich bereits vertraut erscheinen, möchte ich ganz zu Beginn erst einmal die vierte, neue Variante vorstellen. Hier ein kleines Beispiel des Live–Binnings anhand einer schnellen Sinusschwingung aus einem Funktionsgenerator, deren Amplitude zunächst sinkt, bevor der Offset größer wird (Abb. 1).

Für jede Zeile wurden dabei 512 Messpunkte (250 kHz Abtastrate, 12 Bits, mögliche Werte zwischen 0 und 4095) aufgenommen, nach Größe sortiert, und kompakt ausgegeben. So ist schon auf den ersten Blick klar, wo sich das Signal innerhalb der möglichen Grenzen des Analog–Digital–Wandlers bewegt, ob beispielsweise die Verstärkung passt, oder ob das Signal besser noch ein wenig verschoben werden sollte, um einen Anschlag an die Grenzen (das sogenannte Clipping) zu vermeiden. Zu beachten ist, dass es sich hier nicht um eine kontinuierlich durchgehende Aufnahme handelt, sondern jede Zeile eine separate Aufnahme ist. In den Rechen– und Ausgabepausen dazwischen können uns somit einmalige Ausreißer entgehen — diese Ansicht ist also kein wissenschaftlicher „Langzeitdatenrekorder“, sondern als Werkzeug dafür gedacht, einen Überblick und eine vor allem während der Justage hilfreiche schnelle Rückmeldung zu erhalten.

Abb. 4 zeigt, wie ein amplitudenmodulierter Träger (5–Hz–Sinus–Modulation, 31–kHz–Trägerwelle, 100% Modulationstiefe) in der Live–Ansicht aussieht.

Doch nun zu den „klassischen“ Ansichten! Eine 12–kHz–Sinusschwingung mit 1–Vpp–Amplitude und 800–mV–Offset, welches sich in der Live–Ansicht (`0 adc-channel io! live`) so präsentiert, wie in Abb. 5 gezeigt, sieht im Detail (`capture show`) dann so aus, wie Abb. 6 es zeigt.

Diese Oszilloskop–Ansicht ist auf Minimum und Maximum der Messpunkte skaliert und linear aufgetragen. Bei so einer Sinus–Schwingung mit vergleichsweise großer Amplitude ist deren Binning–Tabelle nicht so sonderlich interessant, da die Messpunkte über einen weiten Bereich verstreut liegen (Abb. 2).

Die logarithmisch mit festen Grenzen aufgetragene Fourier–Ansicht zeigt — im Rahmen ihrer Auflösung mit nur 512 Messpunkten — deutlich eine Schwingung um 12 kHz an Abb. 7.

Da wir hier nicht so viel Speicherplatz zur Verfügung haben, überschreibt die Fouriertransformation die Messergebnisse, so dass `show` nur einmal für jedes `capture` (oder direkt nach `live`) aufgerufen werden kann, was aber mit mehr Speicherplatz ohne weiteres auch anders implementiert werden könnte.

Schließlich möchte ich als letztes Anwendungsbeispiel die Ausgabe für ein konstantes Eingangssignal — hier 800 mV — zeigen (Abb. 8), anhand dessen das Rauschen des Signalpfades, insbesondere mit Hilfe der hier ungekürzt abgedruckten Binning–Tabelle, charakterisiert werden kann (Abb. 3).

Die Abtastrate für alle Beispiele liegt bei 250 kHz, wobei ich für die Oszilloskop–Ansicht eine unkalibrierte Messpunkte–Zeitachse gewählt habe, und für die Fourier–Ansicht eine kalibrierte Frequenzachse in Kilohertz. Fast alles kann im Detail an den gewünschten Einsatzzweck angepasst werden — das Signallabor ist keinesfalls als statisches Werkzeug gedacht, sondern vielmehr als eine leicht anpassbare Sammlung von Ideen und Anregungen, wie eine Messwertedarstellung im Terminal gelingen kann. Wer noch viel ausgefeiltere Darstellungen benötigt, wird vermutlich die gepufferten Messpunkte als Rohdaten ausgeben und anschließend mit *Gnuplot* (`http://gnuplot.info/`) oder *GNU Octave* (`https://octave.org/`) im Detail untersuchen und auswerten.

Details zur Implementierung

Das Signallabor läuft mit Mecrisp–Ice, einem 16–Bit–Forth, auf der ULX3S–FPGA–Ausprobierplatine, die typisch für analoge Schaltungen mit FPGAs einen separaten Analog–Digital–Wandler (MAX11125) besitzt, welcher in diesem Falle über eine SPI–Schnittstelle angesprochen wird.

Die Ansteuerung des Wandlers und die Aufnahme habe ich in Logik implementiert, welche Messwerte mit einer festen Abtastrate von hier 250 kHz in einen Puffer legt, der über zwei IO–Register angesprochen werden kann: Das Register `adc-valid` zeigt an, ob mindestens ein Messwert bereitliegt, welcher anschließend über das Register `adc-fifo` aus dem Puffer entnommen werden kann. Um Werte mit präzisen Zeitabständen einzulesen, wird zunächst der Puffer geleert (bei einem FIFO würden sonst zuerst alte Werte geliefert werden), anschließend werden frische Messwerte in einer engen Schleife aus dem FIFO entnommen und in ein Array im RAM abgelegt. Außerdem gibt es noch das Register `adc-channel`, womit der analoge Eingangskanal durch Umschalten eines Multiplexers gewählt werden kann, und `adc-result`, falls man einfach so mal in den zuletzt gewandelten Messwert reinschauen möchte. All dies ist im Listing `capture.txt` zu finden.

Wer das Prinzip noch ein bisschen verfeinern möchte, kann dafür sorgen, dass der Puffer in Logik von selbst „nachschiebt“, so dass ein Leeren vor der Aufnahme entfallen kann. Außerdem ist ein Flag, welches einen Überlauf anzeigt und manuell gelöscht werden kann, nützlich, wenn wir es mit sehr schnellen Abtastraten zu tun bekommen: Ein solches Flag hilft während der Entwicklung der Software sehr, um mit einer einfachen Abfrage prüfen zu können, ob es an irgendeiner Stelle einmal zu einem Verlust von Messwerten gekommen ist — hier ist jedoch der speziell für Forth entwickelte Stack–Prozessor viel schneller als der Analog–Digital–Wandler, so dass auf diese ansonsten sehr empfehlenswerten „Sahnehäubchen“ verzichtet werden konnte.

Wer so etwas Ähnliches in einem Mikrocontroller implementieren möchte, wird vermutlich einen Interrupthandler zum Auslesen des Analog–Digital–Wandlers verwenden und die Werte direkt in der Tabelle oder — falls eine kontinuierliche Aufnahme gewünscht ist — in einem Ringpuffer hinterlegen.

Die Ganzzahl–Fouriertransformation, welche hier aus Platzgründen nicht abgedruckt ist, stellt außer den Definitionen `fix-fft ( 2^n inverse? -- scale )` und `fftswap ( 2^n -- )` folgende Datenstrukturen zur Verfügung, die ausgiebig verwendet werden:

```
 512 constant N_WAVE
   9 constant LOG2_N_WAVE

N_WAVE cells buffer: fft-real
N_WAVE cells buffer: fft-imag
```

Auch, falls keine Fouriertransformation gewünscht sein sollte, wird das Array `fft-real` aus Platzgründen direkt für die Aufnahme der Messwerte verwendet, wobei `fft-imag` für temporäre Berechnungen wie das Binning verwendet und anschließend wieder gelöscht wird.

Das Listing `asciisignal.txt` stellt im Wesentlichen hardwareunabhängige Werkzeuge für die Ausgabe eines in `fft-real` enthaltenen Signales bereit. Es beginnt mit einem Sortieralgorithmus, Heapsort, dessen Vergleich zwischen mit und ohne Vorzeichen umschaltbar implementiert ist. Dies ist wichtig, wenn zum Beispiel ein differentieller Analog–Digital–Wandler verwendet werden sollte. Außerdem gibt es die Möglichkeit zur Umwandlung von 16–Bit–Messwerten von vorzeichenlos zu vorzeichenbehaftet und umgekehrt. Weiter geht es mit kleinen ASCII–Art–Werkzeugen, bevor es spannend wird: Es folgen die Routinen für Mittelwert, Minimum und Maximum, ebenso jeweils in Varianten mit und ohne Vorzeichen, sowie das Binning — aus Platzgründen nur vorzeichenlos — wozu die Messwerte aus `fft-real` in `fft-imag` kopiert, dort sortiert und anschließend gezählt werden, um eine Tabelle für die Häufigkeit bestimmter Messwerte innerhalb der Aufnahme auszugeben. Zum Abschluss folgt die „Signaldruckerei“, welche schließlich die schon in der Vorstellung der Beispielsignale zu bestaunenden Ausgaben zeichnet.

Zu guter Letzt folgt im Listing `live.txt` die Implementierung des Live–Binnings, wo mit Hilfe von Schiebebefehlen eine sehr schnelle Übersicht der Messwerte mit einer festen Breite von 64 Zeichen erstellt wird. Diese Implementierung ist natürlich nicht so flexibel wie die Routinen in Listing `asciisignal.txt`, dafür jedoch ist die benötigte Rechenzeit zwischen zwei Aufnahmen so kurz wie möglich, was für eine Live–Ansicht entscheidend ist.

Anhang: Abbildungen

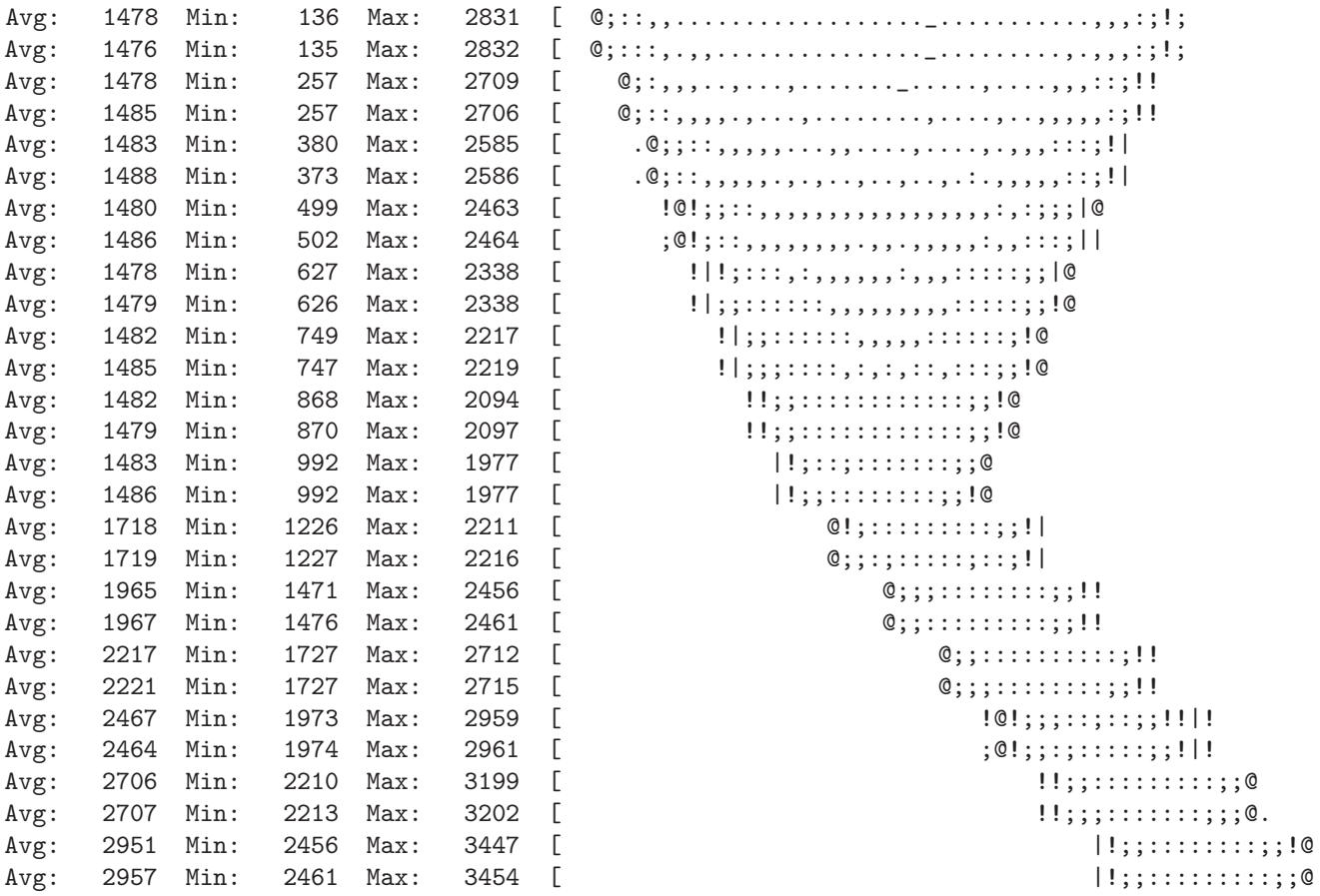


Abbildung 1: Live–Binning. Funktionsgenerator einstellen: Sinus–Amplitude verringern, dann Offset erhöhen.

```
Avg:   1991
Min:    749
Max:   3206

-------------
   ADC     #
-------------
   749     1
   750     4
   751     4
   753     2
   754     2
   755     2
   756     1
   757     3
   ...
  3197     1
  3198     3
  3199     4
  3200     1
  3201     2
  3202     2
  3205     2
  3206     1
```

Abbildung 2: Binning–Tabelle, 12–kHz–Sinusschwingung.

```
Avg:   1977
Min:   1968
Max:   1989

-------------
   ADC     #
-------------
  1968     1
  1969     1
  1971     2
  1972     3
  1973     8
  1974    11
  1975    20
  1976    88
  1977   118
  1978   119
  1979    79
  1980    31
  1981    12
  1982     9
  1983     4
  1984     3
  1986     2
  1989     1
```

Abbildung 3: Binning–Tabelle, konstante Spannung mit Rauschen.

```
Avg:   1976  Min:   1969  Max:   1986  [                    @                    ]
Avg:   1976  Min:   1952  Max:   2002  [                    @!                   ]
Avg:   1976  Min:   1875  Max:   2079  [                   @|||                  ]
Avg:   1977  Min:   1745  Max:   2209  [                 @|!!!!!|               ]
Avg:   1974  Min:   1604  Max:   2349  [               @!;;;;;;;;;!|             ]
Avg:   1977  Min:   1478  Max:   2482  [             @!;;;;;;;;;;;;!|           ]
Avg:   1978  Min:   1387  Max:   2565  [           !|!;;;;;;;;;;;;;!@.         ]
Avg:   1973  Min:   1365  Max:   2589  [          @!;;;;;;;;;;;;;;;;||         ]
Avg:   1977  Min:   1405  Max:   2549  [          _@;;;;;;;;;;;;;;;!@          ]
Avg:   1976  Min:   1503  Max:   2455  [            ||!;;;;;;;;;;;!@!          ]
Avg:   1975  Min:   1640  Max:   2321  [              !|!;;;;;;!@:             ]
Avg:   1977  Min:   1781  Max:   2171  [                :|!!!!@                ]
Avg:   1976  Min:   1898  Max:   2055  [                  !|@,                 ]
Avg:   1976  Min:   1965  Max:   1989  [                   @:                  ]
Avg:   1976  Min:   1960  Max:   1992  [                   @!                  ]
Avg:   1976  Min:   1890  Max:   2062  [                   ||@!                ]
Avg:   1977  Min:   1768  Max:   2187  [                 !|!!!!@,              ]
Avg:   1975  Min:   1625  Max:   2327  [               @|!!;;;!!!|!            ]
Avg:   1975  Min:   1491  Max:   2464  [             @!!;;;;;;;;;;!||          ]
Avg:   1979  Min:   1400  Max:   2556  [           :|;;;;;;;;;;;;;;;!@         ]
Avg:   1973  Min:   1366  Max:   2590  [          @!;;;;;;;;;;;;;;;;!|         ]
Avg:   1977  Min:   1392  Max:   2564  [          ;|;;;;;;;;;;;;;;;;@          ]
Avg:   1978  Min:   1486  Max:   2468  [            @!;;;;;;;;;;;;;!|          ]
Avg:   1974  Min:   1613  Max:   2338  [              @!;;;;;;;!||              ]
Avg:   1976  Min:   1762  Max:   2191  [                ||!!!!!@!               ]
Avg:   1976  Min:   1884  Max:   2074  [                   ||@|                 ]
Avg:   1976  Min:   1958  Max:   1996  [                    @!                  ]
Avg:   1976  Min:   1965  Max:   1988  [                    @.                  ]
```

Abbildung 4: Live–Binning, amplitudenmodulierter Träger.

```
Avg:   1973  Min:    746  Max:   3209  [       !|;:,,,,.,.,.....,......,....,,.:,::!@,         ]
```

Abbildung 5: Live–Ansicht einer 12–kHz–Sinusschwingung mit 1–Vpp–Amplitude und 800–mV–Offset

```
            |    749   Lin                                                3206 |
             ================================================================
     0  1820                              |
     1  2192                                        |
     2  2543                                              |
     3  2838                                                    |
     4  3056                                                        |
     5  3183                                                           |
     6  3193                                                           |
     7  3099                                                          |
     8  2901                                                     |
     9  2621                                                  |
    10  2280                                             |
    11  1915                                       |
    12  1552                           |
    13  1232                    |
    14   974             |
    15   810          |
    16   750        |
    17   803          |
    18   960             |
    19  1206                    |
    20  1526                           |
    21  1884                                 |
    22  2250                                      |
   ...
   503  2842                                                    |
   504  3061                                                         |
   505  3178                                                            |
   506  3197                                                            |
   507  3099                                                          |
   508  2900                                                     |
   509  2618                                                  |
   510  2278                                             |
   511  1911                                       |
             ================================================================

Avg:   1991
Min:    749
Max:   3206
```

Abbildung 6: Oszilloskop-Ansicht einer 12–kHz–Sinusschwingung

```
          |      0   Log                                                32767 |
          ====================================================================
        0,000 <------------------------------------------>
        0,488 <------>
        0,976 <------>
        1,464 <------>
        1,953 <------>
        2,441 <------>
        2,929 <------>
        3,417 <------->
        3,906 <------->
        4,394 <------->
        4,882 <-------->
        5,371 <-------->
        5,859 <--------->
        6,347 <--------->
        6,835 <--------->
        7,324  <--------->
        7,812 <---------->
        8,300 <---------->
        8,789  <---------->
        9,277  <----------->
        9,765   <----------->
       10,253   <------------>
       10,742     <------------>
       11,230         <------------>
       11,718                <------------>
       12,207                   <------------>
       12,695          <------------>
       13,183      <------------>
       13,671    <----------->
       14,160  <----------->
       14,648  <---------->
       15,136  <--------->
       15,625  <--------->
       16,113  <------->
       16,601  <------>
       17,089 <------>
       17,578 <----->
       18,066  <--->
       18,554 <---->
       19,042 <--->
       19,531  <--->
       20,019 <-->
       20,507 <-->
       20,996 <-->
       21,484 <->
       21,972 <->
       22,460 <>
       22,949 <>
       23,437 <>
       23,925 |
       24,414 |
       24,902 |
         ...
      123,535 |
      124,023 |
      124,511 |
          ====================================================================
ok.
```

Abbildung 7: Fourier–Ansicht der 12–kHz–Sinusschwingung aus Abb. 6.

```
live
Avg:   1977  Min:   1962  Max:   1989  [                          @                          ]
 ok.


capture show
             |   1968   Lin                                                    1989 |
             ======================================================================
     0  1977                                |
     1  1979                                      |
     2  1975                           |
     3  1975                           |
     4  1977                                |
     5  1977                                |
     6  1974                      |
     7  1979                                      |
     8  1978                                   |
     9  1978                                   |
    10  1979                                      |
    11  1977                                |
    12  1978                                   |
   ...
   506  1978                                   |
   507  1979                                      |
   508  1979                                      |
   509  1975                           |
   510  1977                                |
   511  1982                                              |
             ======================================================================

Avg: 1977
Min: 1968
Max: 1989


             |      0   Log                                                 32767 |
             ======================================================================
       0,000 <------------------------------------------>
       0,488 |
       0,976 |
       1,464 |
       1,953 |
       2,441 |
       ...
     123,535 |
     124,023 |
     124,511 |
             ======================================================================
```

Abbildung 8: Konstante Spannung mit Rauschen in drei Ansichten (von oben nach unten): Live, Oszilloskop– und Fourier–Ansicht.

Listings

asciisignal.txt

```
1
2   \ Requires double, fixpoint, bitlog and fourier.
3
4   \ ------------------------------------------------
5   \  Heapsort
6   \  https://rosettacode.org/wiki
7   \      /Sorting_algorithms/Heapsort#Forth
8   \ ------------------------------------------------
9
10  : r'@
11    r> r> r@ swap >r swap >r ;
12
13  ' < variable heapsort-comparison
14
15  : precedes ( n1 n2 a -- f)
16    >r cells r@ + @ swap cells
17    r> + @ swap heapsort-comparison @ execute ;
18  : exchange ( n1 n2 a --  )
19    >r cells r@ + swap cells
20    r> + over @ over @ swap rot ! swap ! ;
21
22  : siftDown                        ( a e s -- a e s)
23    swap >r swap >r dup             ( s r)
24    begin                           ( s r)
25      dup 2* 1+ dup r'@ <           ( s r c f)
26    while                           ( s r c)
27      dup 1+ dup r'@ <              ( s r c c+1 f)
28      if                            ( s r c c+1)
29        over over r@ precedes if swap then
30      then drop                     ( s r c)
31      over over r@ precedes         ( s r c f)
32    while                           ( s r c)
33      tuck r@ exchange              ( s r)
34    repeat then                     ( s r)
35    drop drop r> swap r> swap       ( a e s)
36  ;
37
38  : heapsort-core                           ( a n --)
39    over >r                                 ( a n)
40    dup 1- 1- 2/                            ( a c s)
41    begin                                   ( a c s)
42      dup 0< 0=                             ( a c s f)
43    while                                   ( a c s)
44      siftDown 1-                           ( a c s)
45    repeat drop                             ( a c)
46
47    1- 0                                    ( a e 0)
48    begin                                   ( a e 0)
49      over 0>                               ( a e 0 f)
50    while                                   ( a e 0)
51      over over r@ exchange                 ( a e 0)
52      siftDown swap 1- swap                 ( a e 0)
53    repeat                                  ( a e 0)
54    drop drop drop r> drop
55  ;
56
57
58  : heapsort-signed ( addr len -- )
59    ['] < heapsort-comparison !
60    heapsort-core
61  ;
62
63  : heapsort-unsigned ( addr len -- )
64    ['] u< heapsort-comparison !
65    heapsort-core
66  ;
67
68  \ ------------------------------------------------
69  \  Switch between unsigned and signed,
70  \  necessary for 16 Bit ADCs
71  \ ------------------------------------------------
72
73  \ 0 .. 65535 <--> -32768 .. 32767
74  : un<->signed ( -- )
75    N_WAVE 0 do
76      i dup fft-real@ $8000 xor swap fft-real!
77      \ i fft-real@ $8000 xor i fft-real!
78    loop
79  ;
80
81  \ 0 .. 65535 <--> -32768 .. 32767
82  : un<->signed-imag ( -- )
83    N_WAVE 0 do
84      i dup fft-imag@ $8000 xor swap fft-imag!
85      \ i fft-imag@ $8000 xor i fft-imag!
86    loop
87  ;
88
89  \ ------------------------------------------------
90  \  ASCII art tools
91  \ ------------------------------------------------
92
93
94  \ Prints a s15.16 number
95  : f.nr7 ( f -- ) ( f-Low f-High n -- )
96
97    >r ( Low High R: n )
98
99    dup 0< >r
100   dabs
101   ( uLow uHigh )
102   0 <# #s   ( uLow 0 0 )
103
104   r> if [char] - hold then
105
106   drop swap ( 0 uLow )
107
108   [char] , hold<
109   r> 0 ?do f# loop
110
111   #> dup 12 swap - spaces
112
113   type space
114 ;
115
116 : dashes 0 ?do [char] - emit loop ;
117
118 : airbar ( min max -- )
119     2dup =
120     if
121       drop
122       spaces [char] | emit
123     else
124       over      spaces [char] < emit
125       swap - 1- spaces [char] > emit
126     then
127 ;
128
129 : fillbar ( min max -- )
130     2dup =
131     if
132       drop
133       spaces [char] | emit
134     else
135       over      spaces [char] < emit
136       swap - 1- dashes [char] > emit
137     then
138 ;
139
140 \ ------------------------------------------------
141 \  Signal statistics
142 \ ------------------------------------------------
143
144 : stats ( -- avg min max )
145   0.
146   N_WAVE 0 do
147     i fft-real@  0 d+
148   loop
```

```
    \ N_WAVE um/mod nip
    LOG2_N_WAVE 2rshift drop

    -1
    N_WAVE 0 do
      i fft-real@ umin
    loop

    0
    N_WAVE 0 do
      i fft-real@ umax
    loop
  ;

  : stats-signed ( -- avg min max )
    0.
    N_WAVE 0 do
      i fft-real@  m+
    loop
    \ N_WAVE sm/rem nip
    LOG2_N_WAVE 2arshift drop

    32767
    N_WAVE 0 do
      i fft-real@ min
    loop

    -32768
    N_WAVE 0 do
      i fft-real@ max
    loop
  ;

  : printstats ( -- )
    cr
    stats
    rot  ." Avg: " 6 u.r cr
    swap ." Min: " 6 u.r cr
         ." Max: " 6 u.r cr
  ;

  : printstats-signed ( -- )
    cr
    stats-signed
    rot  ." Avg: " 6 .r cr
    swap ." Min: " 6 .r cr
         ." Max: " 6 .r cr
  ;


  : printbin-signed ( bin count -- )
    dup if swap 6 .r 6 .r cr else 2drop then
  ;

  : signal-binning-signed ( -- )

    cr
    13 dashes cr
    ."    ADC     #" cr
    13 dashes

    fft-real fft-imag N_WAVE cells move
    fft-imag N_WAVE heapsort-signed

    cr
    -32768 0
    N_WAVE 0 do
      over i fft-imag@ =
      if 1+
      else
        printbin-signed
        i fft-imag@ 1
      then
    loop
    printbin-signed

    fft-imag N_WAVE cells 0 fill
  ;

  : printbin ( bin count -- )
    dup if swap 6 u.r 6 u.r cr else 2drop then
  ;

  : signal-binning ( -- )

    cr
    13 dashes cr
    ."    ADC     #" cr
    13 dashes

    fft-real fft-imag N_WAVE cells move
    fft-imag N_WAVE heapsort-unsigned

    cr
    0 0
    N_WAVE 0 do
      over i fft-imag@ =
      if 1+
      else
        printbin
        i fft-imag@ 1
      then
    loop
    printbin

    fft-imag N_WAVE cells 0 fill
  ;

  \ ------------------------------------------------
  \  Signal plotters
  \ ------------------------------------------------

  : u*/ ( a b c -- d ) >r um* r> um/mod nip ;

  80 13 - constant WIDTH

  : signalplot-lin-bounds ( umin umax -- )

    cr
    13 spaces
     ." | " over 6 u.r
     ."    Lin" WIDTH 22 - spaces dup 6 u.r
     ."  |"
    cr
    13 spaces
    WIDTH 0 do [char] = emit loop cr

    2dup = if 1+ then \ Prevent division by zero!
    over - ( min range )

    N_WAVE 0 do
      i 6 u.r
      over i fft-real@ dup 6 u.r space
      swap - over WIDTH 1- swap u*/
      spaces [char] | emit cr
    loop

    13 spaces
    WIDTH 0 do [char] = emit loop cr

    2drop
  ;

  : signalplot-lin-bounds-signed ( min max -- )

    cr
    13 spaces
     ." | " over 6 .r
     ."    Lin" WIDTH 22 - spaces dup 6 .r
     ."  |"
    cr
```

```
    13 spaces
    WIDTH 0 do [char] = emit loop cr

    2dup = if 1+ then \ Prevent division by zero!
    over - ( min range )

    N_WAVE 0 do
      i 5 u.r
      over i fft-real@ dup 7 .r space
      swap - over WIDTH 1- swap u*/
      spaces [char] | emit cr
    loop

    13 spaces
    WIDTH 0 do [char] = emit loop cr

    2drop
  ;

  : signalplot-log-bounds ( umin umax -- )

    cr
    13 spaces
     ." | " over 6 u.r
     ."    Log" WIDTH 22 - spaces dup 6 u.r
     ."  |"
    cr
    13 spaces
    WIDTH 0 do [char] = emit loop cr

    bitlog swap bitlog swap

    2dup = if 1+ then \ Prevent division by zero!
    over - ( min range )

    N_WAVE 0 do
      i 6 u.r
      over i fft-real@ dup 6 u.r space bitlog
      swap - over WIDTH 1- swap u*/
      spaces [char] | emit cr
    loop

    13 spaces
    WIDTH 0 do [char] = emit loop cr

    2drop
  ;

  100,0 2variable samplerate

  : fft-plot ( -- ) \ FFT output is always signed
    cr
    13 spaces
     ." | " 0 6 u.r
     ."    Log" WIDTH 22 - spaces $7FFF 6 u.r
     ."  |"
    cr
    13 spaces
    WIDTH 0 do [char] = emit loop cr

    N_WAVE
  \        0 do  \ Symmetric printout
    N_WAVE 2/ do  \ Start with DC component


      i N_WAVE 2/ -
        s>f N_WAVE
        s>f f/ samplerate 2@ f* 3 f.nr7

      i fft-real@ abs bitlog
        WIDTH 1- $7FFF bitlog u*/
      i fft-imag@ abs bitlog
        WIDTH 1- $7FFF bitlog u*/

      2dup umax >r
            umin r> ( min max )

      fillbar
      cr

    loop

    13 spaces
    WIDTH 0 do [char] = emit loop cr
  ;


  : signalplot-lin ( -- )
    0 65535
    signalplot-lin-bounds
  ;

  : signalplot-lin-min-max ( -- )
    stats rot drop
    signalplot-lin-bounds
  ;


  : signalplot-lin-signed ( -- )
    -32768 32767
    signalplot-lin-bounds-signed
  ;

  : signalplot-lin-min-max-signed ( -- )
    stats-signed rot drop
    signalplot-lin-bounds-signed
  ;


  : signalplot-log ( -- )
    0 65535
    signalplot-log-bounds
  ;

  : signalplot-log-max ( -- )
    0 stats rot drop nip
    signalplot-log-bounds
  ;
```

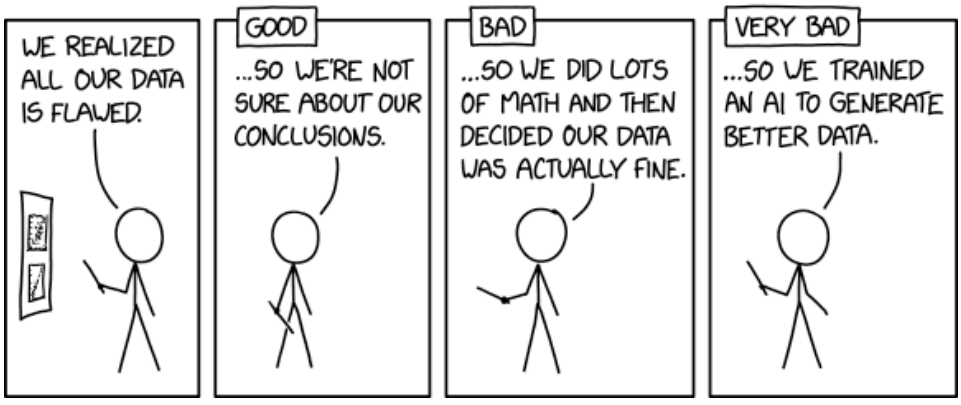
capture.txt

```

  \ ------------------------------------------------
  \   Registers
  \ ------------------------------------------------

  $2030 constant adc-result
  $2040 constant adc-channel
  $20B0 constant adc-valid
  $20C0 constant adc-fifo

  \ ------------------------------------------------
  \   Capture using FIFOs in logic
  \ ------------------------------------------------

  : capture ( -- )

    \ Clear FIFO
    begin
      adc-valid io@
    while
      adc-fifo io@ drop
    repeat

    \ Capture measurements
    fft-real
    begin
      begin adc-valid io@ until adc-fifo io@

```

```
      over !
      1 cells +
      dup fft-real N_WAVE cells + =
    until
    drop

    fft-imag N_WAVE cells 0 fill
  ;

  \ ------------------------------------------------
  \   Analyse the signal
  \ ------------------------------------------------

  : show ( -- )

    signalplot-lin-min-max
    printstats
    signal-binning

    \ un<->signed

    LOG2_N_WAVE false fix-fft drop
    LOG2_N_WAVE fftswap

    250,000 samplerate 2!
    fft-plot
  ;

  \ How to use:
  \
  \   0 adc-channel io!
  \   capture show
```

live.txt

```
: live-bins ( -- )

  \ Clear bins
  64 0 do 0 i fft-imag! loop

  \ Collect all data points into bins
  N_WAVE 0 do

    \ 10 rshift for 16 bit ADC,
    \  6 rshift for 12 bit ADC
    1  i fft-real@ 6 rshift cells fft-imag +  +!
  loop

  \ Find maximum value for scaling
  0
  64 0 do i fft-imag@ umax loop
  bitlog ( maxcount )

  \ Print bins as scaled ASCII art
  64 0 do
    i fft-imag@ bitlog ( maxcount count )

    \ Scale maximum to available chars
    over 8 swap u*/

    \  0123456789012345678901234567890123456
    s"  _.,:;!|@" drop

    + c@ emit
  loop
  drop
;

: live ( -- )
  cr
  begin
    capture
    stats

    rot  ." Avg: " 6 u.r
    swap ."   Min: " 6 u.r
         ."   Max: " 6 u.r

    ."   [" live-bins ." ]" cr

  esc? until
;
```



„Flawed Data“ — Quelle: `https://xkcd.com/2494`

Pictured Numeric Output

Michael Kalus

Wie der geneigte Leser ja bereits weiß, beschäftigt mich die Frage, wie so ein „Haufen Draht“ (Computer) es schafft, mit mir zu sprechen. Bei der Aufklärung hat mir Forth immer geholfen, denn man kann es bis hinunter zu den Bits und Bytes in der Maschine auseinandernehmen und verstehen.

Man kennt das „hello world“ der MCUs: Eine blinkende LED — erste Lebenszeichen einer solch kleinen Maschine und der Beweis, dass man es geschafft hat, sie zum Leben zu erwecken. Diese binären „blinken lights“ wurden später von Schriftzeichen auf Bildschirmen abgelöst. So können Leute heute tatsächlich lesen, was die Maschine sagt!

Es war ein langer Weg bis zur internationalen Verständigung auf Zeichensätze für alle Sprachen dieser Erde. Das ist gelungen und heute kann man tatsächlich mehr auf dem Bildschirm anzeigen als das Alphabet. Die Buchstaben waren jedoch der Anfang. Graphisch vereinfacht für die Maschinen haben sie als ASCII einen Siegeszug um die Welt gemacht. Ohne das gäbe es Forth nicht, es lebt ja von seinen `words`, den Zeichenketten aus solchen Buchstabencodes. Das `type` ist zentral im Forth, es schickt die Zeichenkette an das Terminal wo es dann für das menschliche Auge lesbar erscheint. Vom ASCII zum UTF[1] war es aber noch ein langer Weg, doch das ist eine andere Geschichte. Die Anfänge zeigt Abb. 1. Wie das heute ist, seht ihr ja an unserem Forth–Magazin. Schließlich ist es komplett gesetzt am Laptop, zu Hause, in LaTeX.

Abbildung 1: Aus den Anfängen: Die 14–Segment–Anzeige

Wie die Maschine solche Codes handhabt, kann man in Forth ja einfach verfolgen, oder? Eine Zeichenkette, im Speicher als ASCII–Sequenz aus Bytes abgelegt, wird von `type` an das Terminal geschickt, an die Stelle des Schreibzeigers (Cursor).

```
locate type
/usr/share/gforth/0.7.9_20220713/kernel/
   io.fs:72
User out ( -- addr ) \ gforth
\G counts number of characters TYPEd
\G or EMITed; CR resets it

: (type) ( c-addr u -- ) \ gforth
  dup out +!
  outfile-id write-file drop ;
```

Es ist Sache des Betriebssystems deiner Maschine, wie das bewerkstelligt wird, und Sache der Implementation des Forth, die Zeichenkette (String) an das Betriebssystem zu übergeben. Von Interesse ist hier lediglich, dass dazu die Adresse `c-addr` des Strings und seine Länge `u` gebraucht werden.

Doch anders als die ASCII–Zeichen von Texten, liegen die Zahlen *binär* im Speicher des Computers. In einem 64–Bit–System bilden 8 Bytes eine *CELL* für die Zahl. Doch für die Darstellung im Text müssen wir diesen Zahlenwert in die *Ziffernfolge eines Stellenwertsystems* umschreiben.

Im Dezimalsystem haben wir dafür den bekannten Ziffernvorrat 0 1 2 3 4 5 6 7 8 9. Im Hexadezimalen gibt es darüber hinaus noch A B C D E F, da wird schon eine Anleihe bei den Buchstaben gemacht, denn es gibt keine eigenen Ziffern dafür. Das Binärsystem hingegen nutzt nur die Ziffern 0 und 1 zur Abbildung.[2]

Vom n zur numerischen Zeichenkette

Wie wird eine digital abgelegte ganze Zahl (Integer Number) `n` in das Stellenwertsystem übertragen?

„Tja, Leute, das ist Mathematik!“ (Martin Bitter)

Indem `n` durch die Basis geteilt wird, verschieben sich die Stellen um eine nach rechts. Sei die Basis 10, der Wert 12345, dann ergibt die Division durch 10 den Wert 1234 mit dem Rest 5.

```
decimal
12345 10 /mod . . 1234 5 ok
```

Die Ziffer `5` (Least Significant, „Einer“) wird bei `/mod` als Rest der Teilung auf dem Stack hinterlassen. Das ist die erste Ziffer der Zahl, von rechts gesehen. Dieser Wert muss dann in ASCII verwandelt werden.

```
see #
: # ( d1 -- d2 )
  base @ ud/mod
  rot dup #9 u> #7 and +
  #48 + hold ;
```

Das Forthwort `#` kalkuliert den Code der Einer–Stelle, welcher von `hold` in den Puffer geschrieben wird. Doch

[1] American Standard Code for Information Interchange; Universal Coded Character Set Transformation Format.

[2] Das ist so, weil wir das Stellensystem aus dem Arabischen übernommen haben, die es wiederum vom Sanskrit haben; die römische Zahlschrift hat sich dagegen nicht halten können in Wissenschaft, Technik, Handel und Wandel.

wie kommt man darauf, dass es so ist? Nun, man hangelt sich durch die Zahlenausgabe. Bekanntlich schreibt man in Forth den „Punkt“ `.` dafür. Einfacher gehts nicht.

```
decimal ok
123 456 + . 579 ok
```

Bevor wir dazu kommen, den „Punkt“ zu analysieren, zunächst zum eigentlichen Problem.

Das Stellenwertsystem ist ja arabisch, da wird im Schreibfluss von rechts nach links natürlicherweise von der niedrigsten zur höchsten Stelle die Zahl aufgeschrieben. Aber wir schreiben verkehrt herum, von links nach rechts, was zur Folge hat, dass wir die Ziffern nicht wie sie kommen sogleich ausgeben können, sondern erst einmal aufbauen und vorrätig halten müssen — `hold` — um sie dann von dort als String per `type` ausgeben zu können.

Und wollen wir die Ziffern gar rechtsbündig haben oder mit führenden Nullen oder mit Komma darin und Punkten zwischen Dreiergruppen, Vorzeichen und Währungssymbolen auch noch, dann können die auch nicht einfach so Stelle für Stelle ans Terminal gehen. Auch die eingefügten Satzzeichen müssen mit in den hold–Buffer eingebaut werden, damit das komplette Zahlenbild entsteht.

Dieses Zahlenbild wurde im englischsprachigen Forth „pictured numeric output“ getauft. Gforth spricht auch vom „formatted output“ für seine „numeric–to–ASCII conversion“. Andere Systeme wie Python, Exel, Word, Java sprechen da von „number formatting“. So oder so, das Problem ist überall dasselbe.

Im Libre Office Calc z. B. wird den „Numbers“ ein Format–Code zugeordnet, um eine bestimmte Darstellung zu erreichen. Es sind etliche vorgefertigte da, derer man sich bedienen kann. Es gehen aber auch eigene Formate. Der Format–Code wird als „Bild“ vorgegeben.

Die Zahl 1234.56 ergibt z. B.:

Format–Code	Darstellung
000,000,000.00	000,001,234.56
###,#00,000.00	01,234.56

Tabelle 1: Formatierungsbilder und ihre Darstellung in anderen Sprachen

Alle `#` bleiben leer und alle `0` werden mit `0` aufgefüllt, falls an ihrem Platz keine Ziffern übrig sind, ansonsten kommt die Ziffer dahin. Auch seltsam, oder?

Der Forth–Weg

Problem erfasst. Und wie hat Forth das praktisch gelöst?

Schauen wir nach. In Gforth hat es die Kommandos `see` und ab Version 0.7.9. auch `locate` , um die Konstruktion eines Forthwortes anzusehen. Wir bekommen:

```
see .
: .
  s>d d. ;  ok
```

oder

```
locate
/usr/share/gforth/0.7.9_20220217/kernel/
   nio.fs:131

: . ( n -- ) \ core dot
    s>d d. ;
```

Ihr seht, `locate` liefert sowohl den Pfad zur Definition im Quelltext — `nio.fs` — als auch das Zitat.[3] [4]

Hm, ganz so schlicht ist der Punkt innen drin also nicht, da passiert doch mehr, als man vielleicht vermutet. `s>d` wandelt von einfachgenauer Ganzzahl in eine doppeltgenaue. Und diese wird anschließend mittels `d.` ausgegeben. Zitat aus der Doku des Gforth:

> „Three important things to remember about pictured numeric output:
>
> 1. It always operates on double–precision numbers; to display a single–precision number, convert it first (for ways of doing this see Double precision).
> 2. It always treats the double–precision number as though it were unsigned . . .
> 3. The string is built up from right to left; least significant digit first.“

Wozu der ganze Umstand? Liegt das Geheimnis der Darstellung im `d.` vergraben?[5]

```
: d. ( d -- ) \ double d-dot
  0 d.r space ;
```

Hm, da ist noch immer nicht zu erkennen, wie die Zifferndarstellung gemacht wird. Es wird `d.r` verwendet, was die rechtsbündige Ausgabe von `d` ist. Der Versatz nach rechts ist aber auf Null gesetzt im „Punkt“. Und nach den Ziffern wird noch ein Leerzeichen ausgegeben.

Also tauchen wir noch tiefer ein.

```
: d.r ( d n -- )
  >r tuck dabs <<# #s rot sign #> r> over -
  spaces type #>> ;
```

Ja, da passiert es: `type` ist die String–Ausgabe! Sieht so aus, als hätten wir hier nun tatsächlich die *numeric–to–ascii conversion* der Zahl vor uns. Und die `spaces` machen die passenden Leerzeichen *vor* die Ziffernkette, grade so viele, dass diese rechtsbündig aussieht. Das `r> over -` kalkuliert, wie viele `spaces` gebraucht werden. Im folgenden Beispiel wird die Zahl `12345` um `10` Zeichen nach

[3] Hier auszugsweise wiedergegeben. Im Listing ist der komplette Quellcode.

[4] \G ist ein Alias von \ und wird gebraucht, um das Handbuch zu erstellen. Von der Funktion her kennzeichnet es auch einfach nur eine Zeile als Kommentar.

[5] Die folgenden Codeschnipsel wurden der Quelle entnommen mit `see` oder `locate` und für das Layout des Magazins aufbereitet.

rechts versetzt und dort von rechts nach links geschrieben, also arabisch in unsere Links–Rechts–Schreibung eingepasst.

```
12345 s>d 10 d.r      12345 ok
```

Die Hürde ist somit genommen, es funktioniert.

Wir haben erkannt: Der Punkt wird aus dem allgemeiner verwendbaren Faktor `d.r` erzeugt, er ist ein Spezialfall. Das ist besonders! Denn offensichtlich steht dann da nicht nur die `5`. Es erscheint die komplette Ziffernfolge! Forth erweitert wohl automatisch das Feld, falls mehr Ziffern da hinein sollen, als Platz angegeben worden ist. Wo versteckt sich das?

So wie es aussieht, stellen `<<# ... #>>` eine Klammer dar, innerhalb derer die eigentliche Konversion der Zahl zu ASCII–Ziffern stattfindet — wir kommen nachher noch dazu. Was zwischen `#>` und `#>>` passiert, haben wir schon geklärt, das ist die Ausgabe der Zeichenkette. Aber vor `<<#` steht noch `tuck dabs` und macht stutzig.

```
: tuck ( w1 w2 -- w2 w1 w2 )  \ core-ext
  swap over ;
```

Da wird also die höhere Zelle der doppeltgenauen Zahl zu unterst in den Stack manövriert, um das Vorzeichen zu retten, welches in `w2` enthalten ist. Um es nach der Konversion der Ziffernfolge noch voranstellen zu können. `rot sign` erledigt das.

Damit ist die eigentliche Konversion gefunden: `#s` macht das!

```
: #s ( ud -- 0 0 ) \ core number-sign-s
  BEGIN
  #
  2dup or 0= UNTIL ;
```

Ja, da ist die `begin ... until` Schleife, welche die Zahl Stelle für Stelle abarbeitet und in eine Ziffernfolge wandelt. Unser `#` ist das Arbeitstier darin, und das `2dup or 0=` stellt lediglich fest, ob die Zahl schon ganz aufgebraucht worden ist, denn dann sind wir fertig mit der Konversion. So, nun muss man nur noch herausfinden, wo denn die Zeichenkette ist, die soeben gebaut wurde: `#>` liefert uns die Adresse davon und deren Länge, fertig für `type`.

```
: #> ( xd -- addr u ) \ core number-sign-greater
  2drop holdptr @ holdend @ over - ;
```

hold

Die Funktion `hold` ist noch nicht erklärt.

```
: +hold ( n -- addr )
  \G Reserve space for n chars in the
  \G pictured numeric buffer.
  \G -17 THROW if no space
  negate holdptr +!
  holdptr @ dup holdbuf u< -&17 and throw ;
```

[6] engl.: hold pointer

```
: hold ( char -- ) \ core
  \G Used within <## and #> .
  \G Append the character char to the
  \G pictured numeric output string.
  1 +hold c! ;
```

`hold` setzt den Schreibzeiger um eins weiter und speichert danach den Zeichencode `char` dorthin. Zwischendrin wird noch auf Pufferüberlauf getestet, für alle Fälle. Das bedeutet auch, dass der Puffer `holdbuf` und sein Schreibzeiger[6] `holdptr` an der Stelle dem Gforth schon bekannt sind.

Wie wurde dieser Puffer angelegt, woher weiß Gforth, wo der ist, und wie wird dessen Zeiger verwaltet? Wir kommen zurück auf die oben im Text schon erwähnten mysteriösen Klammern `<<# ... #>>`

```
: <<# ( -- ) \ gforth less-less-number-sign
  \G Start a hold area that ends with #>>.
  \G Can be nested in each other and in <#}.
  \G Note: if you do not match up the <<# with
  \G #>> , you will eventually run out of
  \G hold area;
  \G you can reset the hold area to empty
  \G with <# .
  holdend @ holdptr @ - hold
  holdptr @ holdend ! ;

: #>> ( -- ) \ gforth number-sign-greater-greater
  \G Release the hold area started with <<#.
  holdend @ dup holdbuf-end u>= -&11 and throw
  count chars bounds holdptr ! holdend ! ;
```

Offensichtlich wird der hold–Puffer damit verwaltet, aber nicht ur–initialisiert. Denn `<#` stellt den frisch bereit, ist aber bei der Nachverfolgung unseres „Punktes“ nirgends erschienen. Aber man findet es im `nio.fs` gleich beim `#>` und ist da nicht zu übersehen.

```
: <# ( -- )
holdbuf-end dup holdptr ! holdend ! ;
```

Diese Initialisierung macht Gforth beim Start und ab da verwalten `<<#` und `#>>` den Puffer. Dass der Puffer vom Ende her gefüllt wird, ist, wie gesagt, der arabischen Abfolge geschuldet. Was hier rückwärts eingeschrieben wird, erscheint im `type` dann richtig herum, von links nach rechts. Wenn der String von seinem Ende her aufgebaut wird, muss man ihn fürs `type` nicht nochmal umdrehen.

Der Sinn dieser Holdarea–Klammer `<<#` und `#>>` ist dem Code selbst nicht so ohne weiteres zu entnehmen. Aber wer ähnliche Konstruktionen schon mal gesehen hat, ahnt, was das soll: Es dient dazu, die Ausgaben auch verschachteln zu können, *nesting* von Tasks wird möglich. Interrupt–fest ist das jedoch nicht.

Eigene Zahlenformate

Mit dem nun Entdeckten sollte es gelingen, auch eigene Zahlenformate zu erzeugen. Wie wäre es mit der Ausgabe eines Dezimalpunktes, zwei Nachkommastellen und einem Dollar–Zeichen? Dazu bohren wir das `d.r` zu einem eigenen `print$` auf — siehe Listing 2. Es verhält sich dann wie der „Punkt“, jedoch mit Nachkommastellen und dem $–Zeichen hinten dran.

```
123.45 567.90 d+ print 802.35$
```

Die Stelle des Dezimalpunktes wird in `DPL` gespeichert.[7] Das macht Gforth immer, benutzt es standardmäßig jedoch nicht. Aber man kann sich ja eigene Ausgaben machen, die das nutzen. Und auch andere Zeichen lassen sich einbauen, wie man sieht. Das €–Zeichen funktioniert damit aber nicht, es ist im ASCII nicht enthalten. Dafür braucht man eine neuere Ausgabetechnik, doch das ist ja eine andere Geschichte.

Links

`https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Formatted-numeric-output.html#Formatted-numeric-output`

`https://gforth.org/manual/Number-Conversion.html`

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

Z. B.: `https://www.torsten-horn.de/techdocs/ascii.htm`

Listing 1

```
1   \ Number IO
2
3   \ Authors: Anton Ertl, Bernd Paysan, Neal Crook, Jens Wilke
4   \ Copyright (C) 1995,1996,1997,1998,2000,2003,2006,2007,2010,2015,2016,2019 Free Software Foundation, Inc.
5
6   \ This file is part of Gforth.
7
8   \ Gforth is free software; you can redistribute it and/or
9   \ modify it under the terms of the GNU General Public License
10  \ as published by the Free Software Foundation, either version 3
11  \ of the License, or (at your option) any later version.
12
13  \ This program is distributed in the hope that it will be useful,
14  \ but WITHOUT ANY WARRANTY; without even the implied warranty of
15  \ MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  \ GNU General Public License for more details.
17
18  \ You should have received a copy of the GNU General Public License
19  \ along with this program. If not, see http://www.gnu.org/licenses/.
20
21  require ./io.fs
22
23  : pad    ( -- c-addr ) \ core-ext
24      \G @var{c-addr} is the address of a transient region that can be
25      \G used as temporary data storage. At least 84 characters of space
26      \G is available.
27      here word-pno-size + aligned ;
28
29  \ hold <# #> sign # #s                                  25jan92py
30
31  : +hold ( n -- addr )
32      \G Reserve space for n chars in the pictured numeric buffer.
33      \G -17 THROW if no space
34      negate holdptr +!
35      holdptr @ dup holdbuf u< -&17 and throw ;
36
37  : hold    ( char -- ) \ core
38      \G Used within @code{<#} and @code{#>}. Append the character
39      \G @var{char} to the pictured numeric output string.
40      1 +hold c! ;
41
42  : <# ( -- ) \ core      less-number-sign
43      \G Initialise/clear the pictured numeric output string.
44      holdbuf-end dup holdptr ! holdend ! ;
45
46  : #>      ( xd -- addr u ) \ core       number-sign-greater
47      \G Complete the pictured numeric output string by discarding
48      \G @var{xd} and returning @var{addr u}; the address and length of
49      \G the formatted string. A Standard program may modify characters
```

[7] Leo Brodie sagt (in *Starting Forth*, Kapitel 10): Decimal PLace. [Dank an Bernd Paysan für den Hinweis auf die Herkunft der Abkürzung `DPL`.]

```
50      \G within the string.  Does not release the hold area; use
51      \G @code{#>>} to release a hold area started with @code{<<#}, or
52      \G @code{<#} to release all hold areas.
53      2drop holdptr @ holdend @ over - ;
54
55  : <<# ( -- ) \ gforth   less-less-number-sign
56      \G Start a hold area that ends with @code{#>>}. Can be nested in
57      \G each other and in @code{<#}.  Note: if you do not match up the
58      \G @code{<<#}s with @code{#>>}s, you will eventually run out of
59      \G hold area; you can reset the hold area to empty with @code{<#}.
60      holdend @ holdptr @ - hold
61      holdptr @ holdend ! ;
62
63  : #>> ( -- ) \ gforth   number-sign-greater-greater
64      \G Release the hold area started with @code{<<#}.
65      holdend @ dup holdbuf-end u>= -&11 and throw
66      count chars bounds holdptr ! holdend ! ;
67
68  : sign    ( n -- ) \ core
69      \G Used between @code{<<#} and @code{#>}. If @var{n}
70      \G (a @var{single} number) is negative, append the display code
71      \G for a minus sign to the pictured numeric output string. Since
72      \G the string is built up ``backwards'' this is usually used
73      \G immediately prior to @code{#>}, as shown in the examples below.
74      0< IF '-' hold THEN ;
75
76  : # ( ud1 -- ud2 ) \ core               number-sign
77      \G Used between @code{<<#} and @code{#>}. Add the next
78      \G least-significant digit to the pictured numeric output
79      \G string. This is achieved by dividing @var{ud1} by the number in
80      \G @code{base} to leave quotient @var{ud2} and remainder @var{n};
81      \G @var{n} is converted to the appropriate display code (eg ASCII
82      \G code) and appended to the string. If the number has been fully
83      \G converted, @var{ud1} will be 0 and @code{#} will append a ``0''
84      \G to the string.
85      \ special-casing base=#10 does not pay off:
86      \ <2022Mar11.130937@mips.complang.tuwien.ac.at>
87      base @ ud/mod rot dup 9 u>
88      [ char A char 9 1+ - ] Literal and +
89      '0' + hold ;
90
91  : #s      ( ud -- 0 0 ) \ core  number-sign-s
92      \G Used between @code{<<#} and @code{#>}. Convert all remaining digits
93      \G using the same algorithm as for @code{#}. @code{#s} will convert
94      \G at least one digit. Therefore, if @var{ud} is 0, @code{#s} will append
95      \G a ``0'' to the pictured numeric output string.
96      BEGIN
97          # 2dup or 0=
98      UNTIL ;
99
100 : holds ( addr u -- )
101     \G Used between @code{<<#} and @code{#>}. Append the string @code{addr u}
102     \G to the pictured numeric output string.
103     dup +hold swap move ;
104
105 \ print numbers                                     07jun92py
106
107 : d.r ( d n -- ) \ double       d-dot-r
108     \G Display @var{d} right-aligned in a field @var{n} characters wide. If more than
109     \G @var{n} characters are needed to display the number, all digits are displayed.
110     \G If appropriate, @var{n} must include a character for a leading ``-''.
111     >r tuck  dabs  <<# #s  rot sign #>
112     r> over - spaces  type #>> ;
113
114 : ud.r ( ud n -- ) \ gforth     u-d-dot-r
115     \G Display @var{ud} right-aligned in a field @var{n} characters wide. If more than
116     \G @var{n} characters are needed to display the number, all digits are displayed.
117     >r <<# #s #> r> over - spaces type #>> ;
118
119 : .r ( n1 n2 -- ) \ core-ext    dot-r
120     \G Display @var{n1} right-aligned in a field @var{n2} characters wide. If more than
121     \G @var{n2} characters are needed to display the number, all digits are displayed.
122     \G If appropriate, @var{n2} must include a character for a leading ``-''.
123     >r s>d r> d.r ;
124
125 : u.r ( u n -- )  \ core-ext    u-dot-r
```

```
126       \G Display @var{u} right-aligned in a field @var{n} characters wide. If more than
127       \G @var{n} characters are needed to display the number, all digits are displayed.
128       0 swap ud.r ;
129
130   : d. ( d -- ) \ double  d-dot
131       \G Display (the signed double number) @var{d} in free-format. followed by a space.
132       0 d.r space ;
133
134   : ud. ( ud -- ) \ gforth        u-d-dot
135       \G Display (the signed double number) @var{ud} in free-format, followed by a space.
136       0 ud.r space ;
137
138   : . ( n -- ) \ core     dot
139       \G Display (the signed single number) @var{n} in free-format, followed by a space.
140       s>d d. ;
141
142   : u. ( u -- ) \ core    u-dot
143       \G Display (the unsigned single number) @var{u} in free-format, followed by a space.
144       0 ud. ;
145
```

Listing 2

```
1   : print$ ( n -- ) \ print n with decimal point if any, append $
2                     \ needs variable DPL to work (Gforth specific)
3                     \ handling negatives, behaves like standard "dot"
4     s>d               \ make it a double
5     tuck dabs         \ save sign byte followed by unsigned double
6     <<#               \ start conversion
7     [char] $ hold     \ append "$"
8     dpl @ 0 ?do # loop \ convert decimal places
9     [char] . hold     \ insert "."
10    #s                \ convert remaining digits
11    rot sign          \ get at sign byte, append "-" if needed
12    #>                \ complete conversion
13    TYPE SPACE        \ display, with trailing space
14    #>> ;             \ release hold area
```

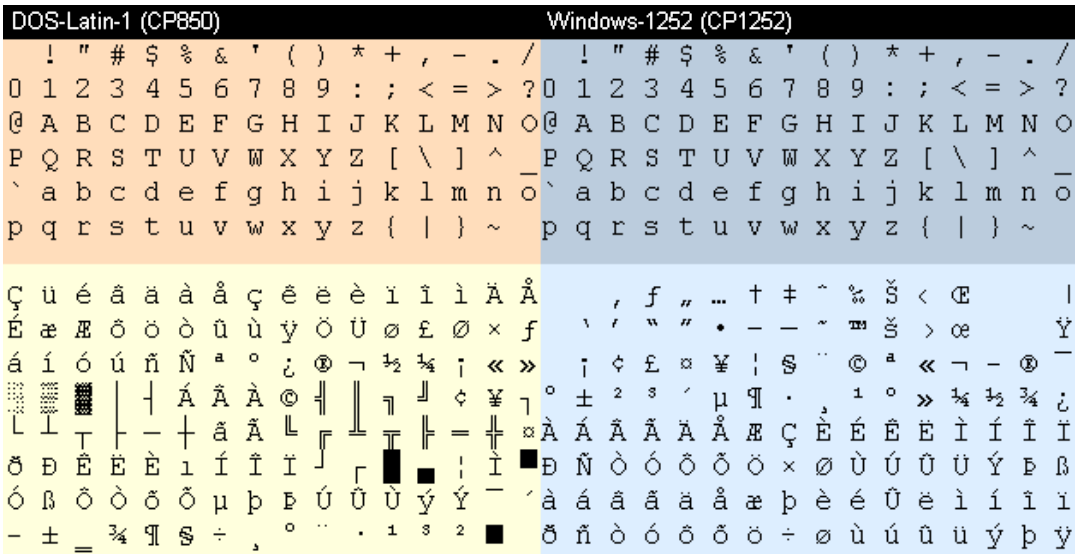


Abbildung 2: Umsetzung der ASCII–Zeichensätze

Print Hex

Colaboration in Forth Works, Albert Nijhof and Ulrich Hoffmann

In this programming pearl Albert Nijhof *shows how to print hexadecimal numbers with a given number of digits, even if your Forth system does not provide pictured numeric output by means of* `<# # #>` *etc.*

Idea

In order to print numbers you have to extract the digits one by one and print each digit. A typical written number representation starts with the most signficant digit, but extracting is easier starting with the least significant digit, the order of digits needs to be reversed. This could be done by storing the digits either on the data stack or on the return stack and make use of their last in first out property.

Display a single digit: `.1HX`

The word `.1HX` (print 1 digit in hex, listing line 5–8) displays a single digit, the last digit of the unsigned number `x`.

Line 6 extracts the least significant nibble of `x` and ignores its more significant part.

To display that nibble as a character the word uses arithmetic with comparison results: The phrase `9 over <` in line 7 compares the nibble with `9`. If it is greater than `9` then the phrase results in `-1` (all bits set). If the nibble is not greater than `9` (i.e. `0` to `9`) the phrase results in `0` (all bits `0`).

`7 and` extracts the least 3 bits. Now we have either `7` or `0`, that we add to the nibble itself. This leads to the `nibble+7`, if nibble is greater `9` or to the original nibble value, if it is not greater than `9`. This is to bridge the gap in the ASCII character sequence between `9` and `A` as we see now.

If we look at the ASCII character sequence we see:

```
48  49  50  51  52  53  54  55  56  57
'0' '1' '2' '3' '4' '5' '6' '7' '8' '9'

58  59  60  61  62  63  64
':' ';' '<' '=' '>' '?' '@'

65  66
'A' 'B' ...
```

There is a gap between the character `9` and the character `A` which is 7 characters wide.

Line 8 adds the character value of `'0'` to the intermediate value. Nibbles `0` to `9` will be mapped to character `'0'` to `'9'`. Nibbles `10` to `15` will be mapped (bridging the gap) to character `'A'` to `'F'` as is required for hexadecimal display.

This character is eventually displayed by the `emit` at the end of `.1HX` (line 8).

Display `n` digits: `.NHX`

In order to display complete numbers the word `.NHX` (print n digits in hex) is defined (line 10–13). `x` is the number to display and `n` ist the number of digits to show.

Line 10 does some clipping so that `n` is always in the range 1 to 16. This avoids unpleasant surprises of seemingly endlesss output if the passed `n` happens to be outside that range.

To display complete numbers `.NHX` first iterates n–1 times (`DO ... LOOP` on line 12) and puts shifted numbers on the stack so that each numbers least significant nibble is one of the nibbles of `x`. The number with the least significant nibble of `x` first, the number with the most significant nibble of `x` on top of stack.

After that the nibbles on the stack are processed in reversed order. They are displayed via `.1HX` (`DO ... LOOP`, line 13) that runs `n` times. `.NHX` ends by printing a space so that consecutive calls to `.NHX` will print numbers space separated.

No `DO LOOP`

If your system does not provide `DO LOOP` then you can rewrite the loops using `BEGIN ... WHILE ... REPEAT` or `BEGIN ... UNTIL` as you can see in line 27–29.

Line 26 does parameter clipping of `n` as before.

To avoid stack juggling the first loop (line 28) collects nibbles on the return stack. (That wouldn't have been possible with `DO LOOP` above as the loop parameters block the return stack.)

The second loop (line 29) retrieves the nibbles from the return stack and prints them with `.1HX` as above. When finished it drops the loop parameter. The obligatory space ends the number output (line 29).

Example output

Lines 18–22 show how to use `.NHX` and what output you can expect. If the number of digits `n` given is less then the actual numbers of digits to completely display `x` then `.NHX` truncates the display showing only the least significant digits. If `n` is larger, then `.NHX` adds leading `'0'`s. (Text: Ulrich Hoffmann)

Listing

```
1   \ Tools -- Formatted unsigned single hex number output, not using BASE
2   \ an-17jan2022
3
4   decimal
5   : .1HX ( x -- )     \ Print last digit of x in hex
6       15 and                  \ lowest nibble
7       9 over < 7 and +        \ for A..F
8       [char] 0 +   emit ;
9
10  : .NHX ( x n -- )   \ Print last n digits of x in hex
11      1 max 16 min >r                 \ x r: n
12      r@ 1 ?do dup 4 rshift loop      \ collect on data stack
13      r> 0 do .1hx loop space ;
14  \ ----- end of code -----
15
16  (*
17  Examples
18  decimal
19  19150 2 .nhx      \ CE
20  19150 3 .nhx      \ ACE
21  19150 4 .nhx      \ 4ACE
22  19150 8 .nhx      \ 00004ACE
23
24  \ .NHX version without DO-LOOP
25  : .NHX ( x n -- )   \ Print last n digits of x in hex
26      swap >r   1 max 16 min dup          \ n n r: x
27      begin 1- dup while r@ 4 rshift >r   \ collect on return stack
28      repeat drop
29      begin r> .1hx   1- dup 0= until drop space ;
30  *)
31  \ <><>
32
```

You can find these and more Forth ideas at `project-forth-works.github.io`. Do you happen to have any Forth ideas? Then please take part in *Project Forth Works*.

Projekt Forth–Buch

In der Forth–Gesellschaft wurde ein Projekt gestartet mit dem Ziel, ein neues Forth–Buch zu erstellen.

Dieses Buch wird von der Forth–Gesellschaft unter einer freien Lizenz (z. B. Creative Commons) veröffentlicht werden und in der ersten Ausgabe Neueinsteiger bei den ersten Schritten mit Forth begleiten.

Das Buch wird sowohl Forth auf Mikrocontrollern (z. B. Mecrisp) als auch Forth auf PC–Systemen (GNU/Forth — Gforth) behandeln. Eine Book–on–Demand Papierausgabe ist geplant.

Die Diskussion rund um die Inhalte des Buches findet im Mattermost Chat[1] statt.

In regelmäßigen Abständen trifft sich das Redaktions–Team Online, um die weiteren Arbeiten an dem Buch zu koordinieren.

Die Quelldateien des Buches finden sich im GitHub–Repository unter `https://github.com/forth-ev/book`

Interessenten an diesem Projekt (Autoren, Grafiker, Leser) nehmen bitte Kontakt auf per E–Mail an das Büro der Forth–Gesellschaft: `secretary@forth-ev.de`

Beste Grüße, Carsten Strotmann

[1] `https://chat.forth-standard.org/forth-standard/channels/forth-buch`

Forth–Gruppen regional

Bitte erkundigt euch bei den Veranstaltern, ob die Treffen stattfinden. Das kann je nach Pandemie–Lage variieren.

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

München **Bernd Paysan**
Tel.: (0 89) – 41 15 46 53
`bernd@net2o.de`
Treffen: Jeden 4. Donnerstag im Monat um 19:00 auf `http://public.senfcall.de/forth-muenchen`, Passwort `over+swap`.

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

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

Dienste der Forth–Gesellschaft

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

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

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

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

Spezielle Fachgebiete

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

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

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

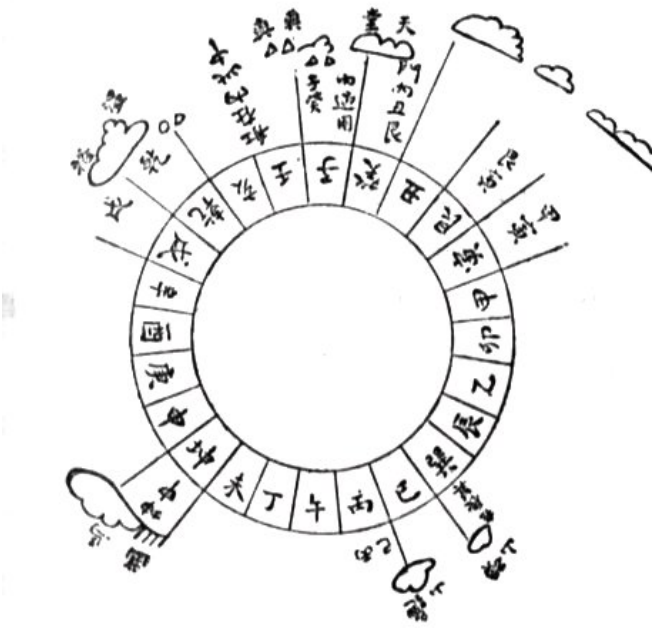
Termine

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

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

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

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



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

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

Nachlese zum Forth–Sommertreffen

M. Kalus

Unser Forth–Sommertreffen, das neue Format leibhaftiger Begegnung unerschrockener Forther, hatte dann im Juni tatsächlich stattgefunden. Wir waren im Linux–Hotel in Essen untergebracht (Abb. 1). Zu einem zwanglosen Gedankenaustausch, ohne besondere Ansprüche an Vorträge. So ganz ohne Präsentationen verlief es dann aber doch nicht. Es stellte sich nämlich heraus, dass der eine oder andere was mitgebracht hatte und vorführen wollte. Und so versammelten wir uns abends im kleinen Hörsaal des Linux–Hotels. Tagsüber wurde gefachsimpelt im Linux–Park. Am Samstag gab es eine Exkursion zum Museum Zeche Zollverein.

Abbildung 1: Auf der Treppe des Linuxhotels.

Abbildung 2: Im Restauarant „The Mine“ auf Zollverein.

Abbildung 3: Auf der Führung durch Zollverein.

Aufzeichnungen der Gespräche und der spontanen Vorführungen gab's keine. Das schlummert nun als kostbarer Schatz in der Erinnerung der Teilnehmer. Eine knappe Liste haben wir aber rekonstruieren können.

Jörg Völker hat eines seiner Hobbys, die Ton– und Musik–Synthesizer, präsentiert in Bild und Ton. Und hatte eine stolze Sammlung einschlägiger Synthesizer vorgeführt. Alte wie neue, eindrucksvoll. Klar, wer sich da *so* mit auskennt, geht irgendwann dazu über, Soundmodule selbst herzustellen und zu vertreiben.

Wolfgang Strauss organisierte einen Workshop zum Thema „Uxn Ecosystem“. Die Teilnehmer konnten auf dem eigenen Laptop „nachspielen“, wie es sich anfühlt, Tools zu verwenden, die von zwei Künstlern entwickelt wurden, welche auf einem Segelboot lebend mit geringen Resourcen auskommen müssen.

Die Exkursion ging diesmal in die Zeche Zollverein. Dieses riesige Gelände ist frei zugänglich, ohne Termine, hat ein großes Museum und Restaurants — Deftiges aus dem Ruhrgebiet, u. a. die klassische Currywurst / Pommes (Abb. 2). Es stellte sich heraus, dass eine Führung durch die Anlage Zollverein und den Förderturm doch nicht ausgebucht war, und so haben wir uns angeschlossen. Ein eindrucksvolles, riesiges Gelände! Man bekam einen lebhaften Eindruck von dem Betrieb, der das mal war (Abb. 3).

Erstaunlich, was für ein Durchsatz da mal Tag und Nacht erreicht worden war. Die modernste und förderstärkste Zeche der Welt war Zollverein. Kommt ihr dort mal in die Gegend: ein Muss!

Was gab's so beim Fachsimpeln? Für mich: ein gutes Gefühl zu den Forth–Leuten, die da waren. Eine illustere Gesellschaft mit enorm viel technischem und auch sonstigem Know–how. Mein Eindruck: Hier ein Grüppchen, dort eines. Jeder schien was mitgenommen zu haben, gute Erfahrungen. Die einzelnen Themen, die da angeschnitten wurden, kenne ich natürlich nicht alle. Lieferschwierigkeiten elektronischer Bauteile und die Sorge um unsere Stromversorung kam aber mehrmals auf.

Schade, dass man nicht so in der Nähe wohnt und öfters solchen Austausch haben kann. Doch man kann sich ja schreiben und sogar in Videokonferenzen sehen, wenn man möchte. Solche Treffen verbinden.

So hoffe ich auf weitere Treffen dieser Art. Aber auch die gute alte Kongressform soll wiederkommen im nächsten Jahr. Drücken wir die Daumen, dass es gelingt.