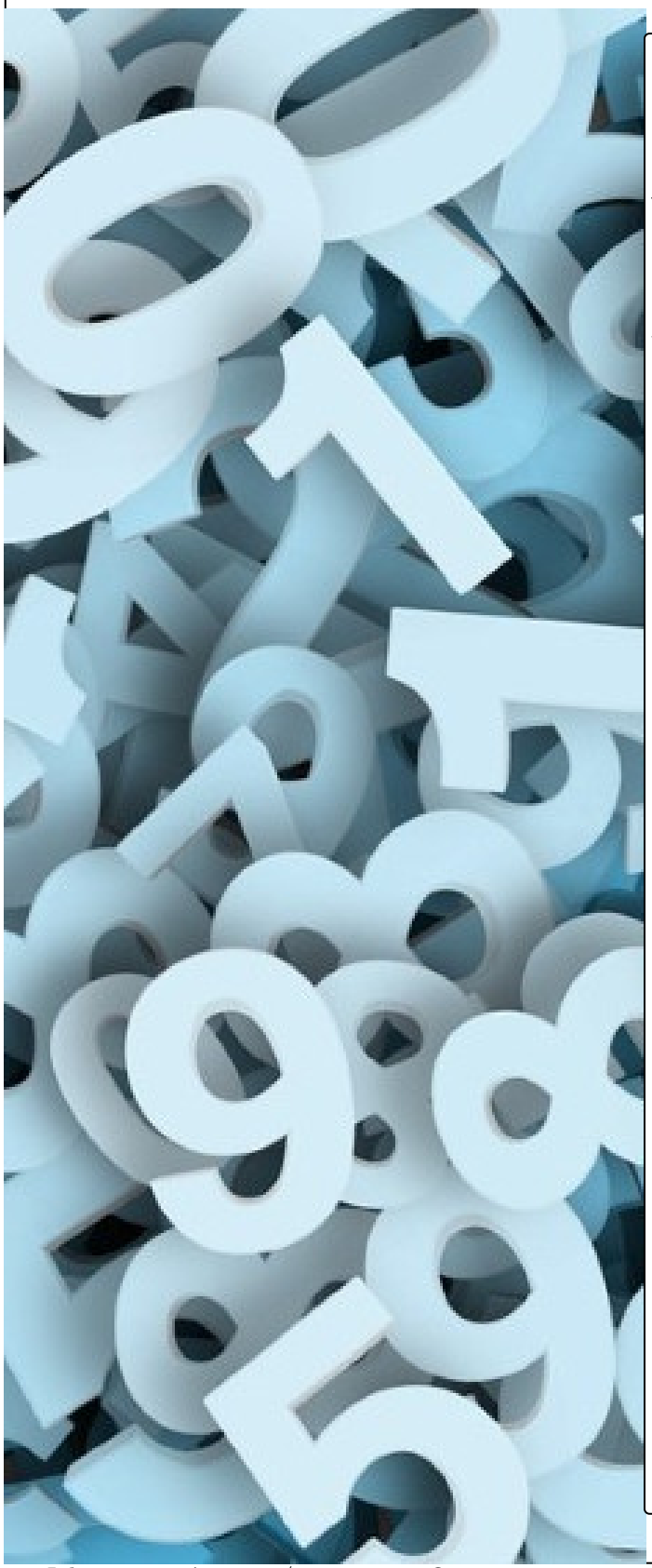
Vierte Dimension

1/2019

Das Forth–Magazin

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



In dieser Ausgabe:

Dienstleistungen und Produkte fördernder Mitglieder des Vereins

Titelbild MK Selbst erstellter Bildausschnitt ... irgendwo aus den Tiefen des Internets
Comic : https://www.xkcd.com

Impressum

Name der Zeitschrift

Vierte Dimension

Herausgeberin

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

Redaktion & Layout

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

Anzeigenverwaltung

Büro der Herausgeberin

Redaktionsschluss

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

Erscheinungsweise

1 Ausgabe/Quartal

Einzelpreis

4,00 € + Porto u. Verpackung

Liebe Leser,

ihr seht, es hat geklappt, das Heft ist da, noch vor der Forth–Tagung in Worms. Angemeldet seid ihr schon alle? Falls noch nicht, auf der letzten Seite findet ihr alles dazu.
Und bei `tagung.forth-ev.de` könnt ihr euch rasch noch anmelden zur Forth–Tagung 2019 im Weingut im Frühling.

Diesmal scheint es in der Luft gelegen zu haben, sich der Verarbeitung und Darstellung von Zahlen zu widmen. Auf einmal kamen Beiträge dazu herein und schwups war das Heft damit gefüllt. Und es sind sogar weitere angekündigt — das wäre ja toll, gäbe es dazu noch ein Heft.

Und als sich das so abzeichnete, hat Jens Storjohann zusammengefasst, was es mit den Zahlen so auf sich hat und wie man von der Idee der Zahl zur technischen Umsetzung kommt. Denn in Forth ist man immer nahe dran an der zugrunde liegenden Maschine, da geht man der Technik auf den Grund. Das gilt auch für die arithmetischen Operationen.

Albert Nijhof hat das auch getan und gleich eine Bruchrechnung mit Forth daraus gemacht. Keine leichte Übung; aber ihr seht, es geht. Da staune ich nur, wie überhaupt bei allem Jonglieren mit Zahlen.

So auch beim Zen Floating Point von Martin Tracy, an das bei dieser Gelegenheit erinnert werden soll — ein erstaunliches Kunststück der frühen Forth–Zeit.

Und in dem Zusammenhang erinnert Matthias Koch an das Fixkommazahlenformat und beschreibt es einmal gründlich, nachdem es schon so lange im Einsatz ist. Fast im Verborgenen, wie mir scheint.

Die Fixpoint Math Library von Andrew Palm fügt sich da prima ein und es freut mich, dass er sein Einverständnis gegeben hat, diese abdrucken zu dürfen. Danke, Matthias, für die Vermittlung.

Schließlich zeigt Carsten Strotmann auf, wie nahe man auch heute noch an seine Maschine herankommen kann; wenn man will auch ganz ohne Betriebssystem zwischen sich, Forth und der Hardware. Treiber schreiben ist da dann natürlich gefragt. Man muss es also wollen. Aber dann geht es eben doch.

So, dann sehen wir uns hoffentlich in Kürze in Worms.

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-2019-01`

EuroForth conference 2019 in Hamburg

Dear Fellow Forthers, the schedule for this years EuroForth conference has been settled. It will take place on *September 13–15, 2019*, in Hamburg. An optional 4th day on September 16th will be offered.

As usual the conference will be preceeded by the Forth 200x standards meeting which starts Wednesday, September 11th.

The exact location has yet to be found. If you have any questions please do not hesitate to ask. Ulrich Hoffmann

`uh@fh-wedel.de`

Projekt virenfreies Win32Forth, update

Lieber Leser,

seit geraumer Zeit wird unsere Win32Forth–Programmierumgebung von einigen Antiviren–Programmen beim Download oder beim Installieren als „Falsch Positiv“ angekündigt.

Was heißt „Falsch Positiv“? Das ist ein Zustand, in dem die Antivirussoftware nicht genau weiß, ob das Archiv (exe oder nicht) eine Bedrohung für das System bedeutet. Als Schutz wird das Programm dann gesperrt oder in Quarantäne gebracht.

Dies verursacht mehrere Probleme. Neu gewonnene Programmierer, die Win32Forth testen wollen, können es nicht installieren und gestandene Win32Forth–Liebhaber können ihre Win32Forth–Programme nicht mehr zum Laufen bringen. Etliche Programme, die mit Win32Forth erstellt worden sind, laufen in Forschungslaboren auf der ganzen Welt. Was passiert mit dieser Arbeit? Muss man sie vergessen oder wieder neu schreiben?

Viele dieser Win32Forth–Benutzer haben sich bei uns auf Facebook[1] gemeldet, sie brauchen eine Lösung.

Ich selbst habe FKERNEL.EXE (55 KB) auf der Suche nach Virus–Code oder irgendwelchen anderen anormalen Dingen disassembliert. Ich habe den Disassembler–Output mit dem Quellcode von Win32Forth FKERNEL verglichen, und alles passt zu 100 %. Das beudeutet, dass wir nichts Falsches im EXE–Code haben. Trotzdem wird Win32F als „Falsch Positiv“ identifiziert und blockiert.

Bei der weiteren Suche fand ich einige Probleme im HEAD von PE-EXE. Einige Eingaben erwiesen sich als problematisch, und mit einem anderen HEAD, geborgt aus einer älteren Version von Tom Zimmer, konnte ich die Rate der Falschmeldungen ganz beeindruckend vermindern, aber sie ist noch nicht Null.

Unser Team ist auf der Suche nach einer Lösung für Win32Forth. Wir haben mehrere Fronten, an denen wir simultan arbeiten und wir brauchen Hilfe! Viele Köpfe denken besser als wenige und in der Zusammenarbeit werden ganz sicher weitere Ideen entstehen.

Wir sind bis jetzt: Jos van de Ven als Chief Maintainer des Win32Forth–Projekts, Wesley Zhang 張燕南 als Head des Win32F–Heilungs–Teams der FIG–Taiwan und als Leiter der Win32Forth–Facebook–Gruppe und Peter Minuth als Tester, Programmierer und Ansprechpartner für die VD– und für die FB–Gruppe.

Bitte helft uns, Win32Forth zu heilen! Alle Forth–Programmierer sind herzlichst in der Gruppe willkommen!

Peter Minuth

Wer bin ich?

Aus der Korrespondenz zwischen Jens Storjohann und Michael Kalus, im Februar.

JS: Lieber Michael, beim Lesen der letzten VD ist mir aufgefallen, dass bei Namensnennungen im Editorial und in den Beiträgen bei Personennennungen oft nur der Vorname verwendet wird. Nun wissen die Eingeweihten, wer jeweils gemeint ist, aber eben nur die.

Die Zeitschrift führt das Wort „Wissenschaft“ auf dem Titelblatt. Da möchte man gerne mehr Nachvollziehbarkeit, auch zu einem späteren Datum, und die Möglichkeit einer Suche im Internet.

Sonst fand ich die VD gut. Viele Grüße, Jens

MK: Hi Jens. Im Editorial verwende ich nur die Vornamen, das ist richtig. Da wird aber auch lediglich auf den Beitrag des Autors im Heft Bezug genommen. Und die Beiträge haben immer den vollen Namen. Und Titel. (hoffentlich hab ich keine Titel vergessen) :)

Bei den Meldungen handelt es sich ja meist nicht um einen Autor, sondern um einen aufmerksamen Leser, der etwas aus dem Netz gefischt hat. Da werden auch nur die Vornamen verwendet oder gar nur ein Kürzel. Da soll nicht im Vordergrund stehen, *wer* das geschickt hat, sondern die Meldung an sich. Ein echter Leserbrief wiederum soll den vollen Namen haben.

Wo hast du keinen Bezug zum vollen Namen im gleichen Heft gefunden, obwohl es dir wichtig gewesen wäre?

JS: . . . wir sind uns darüber einig, dass wir nur wenig verschiedene Ansichten haben. Über die „first name basis“ im Editorial ließe ich mit mir reden.

Es stört mich auch nicht, wenn in Kurznachrichten der Autor, z. B. Carsten Strotmann als *Cas* erscheint, obwohl für Rückfragen eine Entschlüsselung, z. B. im Impressum hilfreich wäre. Gegen das Kürzel *mk* habe

[1] `https://www.facebook.com/groups/714452415259600`

ich auch nichts, weil unschwer zu erraten ist, wer sich bescheiden hinter sein Kürzel stellt.

Etwas störender für mich war der Bericht auf Seite 5 über die Open–Rhein–Ruhr. „Ich“ stellte den *NumWorks* aus und dieser „ich“ heißt Wolfgang. Wer ist Wolfgang?

Ich kann raten, dass die auch im Text erwähnten „Carsten“ und „Martin“ die altgedienten Mitglieder Carsten Strotmann und Martin Bitter bedeuten. Aber warum nicht ein bisschen Redundanz?

Der Artikel „Amforth wird groß“ enthält ein paar einsame Vornamen.

Mich stört an der Beschränkung auf Vornamen, dass der Informationsgehalt der Artikel damit im Laufe der Zeit teilweise verloren geht. Insbesondere wenn man eine einzelne Meldung oder einen einzelnen Artikel am Computer liest, möchte man, dass diese aus sich selbst heraus verständlich sind.

MK: Dann werde ich darauf achten und es so machen, wie du es vorgeschlagen hast. Danke für's aufmerksame Lesen. mk

Who is who?

Nachtrag zum Heft 2018–04 unseres Forth–Magazins, der „Vierten Dimension“.

Hier die vermissten Namen und ihre Kürzel in alphabetischer Reihenfolge:

cas	Carsten Strotmann
ew	Erich Wälde
mb	Martin Bitter
mk	Michael Kalus
wost	Wolfgang Strauß

Nun bleibt noch die Frage zu klären, ob wir künftig alle Autoren mit oder ohne ihre Titel nennen sollen. Bisher waren wir im Verein, unserer Forth–Gesellschaft, so verfahren, dass wir per „du“ sind und auf die Titelei verzicht haben; bei den Treffen und auch beim Abdruck des Namens eines Autors. Wohl, weil wir mit dem Heft vor vielen Jahren angefangen haben, als noch viele Schüler oder Studenten waren und ein Doktor– oder Professor–Titel in weiter Ferne lag. Das hat sich ja inzwischen erfreulich geändert. Den einen oder anderen ziert nun solch ein akademischer Titel. Also warum nicht hinzunehmen zum Beitrag? mk

Dr. Forth h. c.

Jüngst hat Matthias Koch die Doktorwürde an der Universität Hannover im Fach Biophysik erhalten, ein *Doktor rerum naturalis*.

Bei der Maker–Fair in Hannover wurde ihm darüber hinaus von Klaus Zobawa der Titel *Doktor Forth h. c.* verliehen. Hier der Schnappschuss davon. Auf dem Hut ein Swap, eine Mikroprozessorplatine und ein Reagenzglas, symbolisch für die Algenexperimente seiner Doktorarbeit. :-) mk

Neuigkeiten aus der Mecrisp–Compilerschmiede: Acrobatics

Nachdem mein erster Registerallokator schon lange stabil auf der ARM–Architektur im Einsatz ist, wurde es nun höchste Zeit, auch Mecrisp–Quintus mit einem stark optimierenden Compiler zu versehen! Hiermit möchte ich *Acrobatics* vorstellen, meinen neuen Forth–Compiler, der flotten Maschinencode für RISC–V (RV32IM) und MIPS (M4K) erzeugt. Der Name ist scherzhaft an „Stackrobatics“ angelehnt, entspricht aber ebenso der Wirklichkeit: Innendrin werden tatsächlich Stackelemente herumgewirbelt!

Während Mecrisp–Stellaris RA ein ausgefeiltes Kunstwerk in Assembler ist und für die meisten vermutlich nur mit Kopfverrenkungen zu verstehen sein wird, ist Acrobatics in Forth geschrieben. RISC–V und MIPS sind sich sehr ähnlich, weswegen der gleiche Compiler mit ein paar Schaltern für beide Architekturen verwendbar ist. Auf diesen beiden stehen — ganz im Gegensatz zu den ARM–Cortex–Kernen — viele Register zur Verfügung, sodass Acrobatics nicht nur den Datenstack, sondern auch den Returnstack in Registern halten kann. Opcodes mit durchgehend drei Registerfeldern vereinfachen die Compilerlogik sehr.

Mir hat die Entwicklung Spaß gemacht, und ich hoffe, dass der dem Mecrisp–Quintus beiliegende Forth–Quelltext (common/acrobatics.txt) verständlich genug ist, um Neugierigen einen Einblick zu gewähren.

Matthias Koch

`https://sourceforge.net/projects/mecrisp/`

Fortsetzung auf Seite 17

Warum gibt es so viele Arten von Zahlen . . .

Jens Storjohann

. . . und noch mehr Darstellungen und Operationen mit ihnen? Das klingt manchem vielleicht nach langweiligen ermüdenden Prozeduren. Aber wir müssen ja nicht selbst rechnen, sondern überlassen es dem Computer! Der ist weder interessiert noch gelangweilt, und ermüden tut er auch nicht. Aber verstehen tut der Computer noch (!) nicht, was er tun soll. Das müssen wir. Aber das ist gar nicht so schlimm!

Die in der Computer-Technik verwendeten Zahlen und ihre implementierten Operationen kommen von grundlegenden mathematischen Ideen, die zum größten Teil mehrere Tausend Jahre alt sind.

Um die Zahlen, wie sie uns heute erscheinen, besser darstellen zu können, habe ich die geschichtliche Entwicklung so dargestellt, wie sie gewesen wäre, wenn die Menschen auf mich gehört hätten. An der Art, wie im Römischen Reich Zahlen dargestellt wurden, kann der geneigte Leser unschwer erkennen, dass ich dabei nicht gefragt wurde.

Eine seriöse geschichtliche Betrachtung ist zum Beispiel in [2] zu finden.

Für deutschsprachige Leser gibt es das Kapitel über Arithmetik von Donald Knuths bekannter Reihe „The art of computer programming“ in Übersetzung [3]. Hierin sind auch Hinweise zur Geschichte enthalten.

Etwas aus der Geschichte der Zahlen und ihrer Schreibweise

Der große Mathematiker Leopold Kronecker sagte: „Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk.“

Die natürlichen Zahlen $1, 2, \cdots$ brauchen wir, um Personen, Lebewesen oder Gegenstände zu zählen. Wenn wir die Zahlen als universell einsetzbares Konstrukt begreifen, steckt darin schon eine geistige Leistung, die Kinder erst lernen müssen.

Ziemlich natürlich ist es dann, das Zusammenfassen von Mengen sich in der **Addition** widerspiegeln zu lassen.

Ähnlich kann man das Abzählen von großen Mengen durch Einteilen in gleich kleine Mengen und **Multiplikation** sich plausibel entstanden denken.

Beide Operationen führen nicht aus dem Bereich der natürlichen Zahlen heraus.

Wenn die Menschen anfangen zu tauschen, kommt die **Division** ins Spiel: Ich gebe 3 Pflaumen für einen Apfel. Wie viele Äpfel bekomme ich für eine Pflaume?

Man möchte, dass jede Divisonsaufgabe, außer einer Aufgabe mit einer Null im Nenner, eine Zahl als Lösung hat. Damit sagt man „3 durch 2 gleich 3 Halbe“ statt „3 durch 2 gleich 1 Rest 1“.

Hier würde der Mathematiker sagen, dass die Umkehrung der Multiplikation, nämlich die Division, aus dem Bereich der ganzen Zahlen herausführt.

Damit sind die **Brüche** oder die **rationalen Zahlen** erfunden.

Wenn sich das Wirtschaftsleben vom einfachen Tauschen löst, kommen **negative Zahlen** als Darstellung für Schulden in die Betrachtung. Man kann auch sagen, dass die Umkehrung der Addition, nämlich die Subtraktion aus dem Bereich der positiven Zahlen herausführt.

Man spricht von „ganzen Zahlen“, wenn die natürlichen Zahlen durch ihre negativen Spiegelbilder ergänzt sind.

Es gibt abweichende Konventionen, ob die Null zu den natürlichen Zahlen oder zu den ganzen Zahlen gehört.

Wichtig ist die **Ziffer** 0, die für eine Stellenschreibweise eine wichtige Funktion hat, zu unterscheiden von der **Zahl** Null, die zum Beispiel auch in der Schreibweise 0,00 in Rechnungen auftreten kann.

Die verbreiteten Programmiersprachen reservieren für Zahlen einen Speicherbereich, der je nach Art verschieden, aber fest ist.

Besonders interessant ist die Art, wie Forth mit ganzen Zahlen (`variable`) oder `constant` umgeht. Die gespeicherte Zahl hat eine Standardlänge von n Bits. Um mit kleinen Zahlen argumentieren zu können, nehmen wir 8 Bits als Standardlänge an. Wenn die gespeicherte Zahl als „unsigned integer“ aufgefasst wird, z. B. durch Aufruf von `u+` hat das höchste Bit die Wertigkeit $2^7 = 128$. Damit ist der darstellbare Bereich gegeben durch 0 bis $2^8 - 1 = 255$. Wenn die gespeicherte Zahl wegen eines Aufrufs von z. B. `+` als „signed integer“ aufgefasst wird, hat das höchste Bit die Wertigkeit $-2^7 = -128$. Damit ist der darstellbare Bereich gegeben durch -128 bis +127.

Vorteil ist, dass in jedem Fall nur **eine** Darstellung der Null existiert. Wenn ein Bit als Vorzeichen eingesetzt wird, hätte man die Darstellungen 0 und -0 zu betrachten. Durch die Beschränkung auf eine feste Speicherplatzlänge gibt es also abweichend von der mathematischen Idee der ganzen Zahlen Einschränkungen.

Für technische Berechnungen hat man mit sehr großen Zahlen zu tun, wenn zum Beispiel Längen großer Maschinen in Millimeter gemessen werden oder auch sehr kleine Zahlen, wenn thermisch bedingte Längenveränderungen von Komponenten ebenfalls in Millimeter berechnet werden sollen.

Abhilfe schaffen Gleitkommazahlen (floating point numbers). Sie werden meistens „real“ genannt, stellen aber nur eine Teilmenge der rationalen Zahlen dar.

Wichtige Unterscheidungen

Wir unterscheiden Ziffern und Zahlen. Es gibt im Dezimalsystem 10 Ziffern aber prinzipiell unendlich viele Zahlen, die im Dezimalsystem oder auch in anderen Systemen notiert werden können.

Im Gegensatz zum gebräuchlichen Jargon gibt es keine „Hex-Zahlen“, sondern Zahlen, die im Hexadezimalsystem dargestellt sind.

Was findet sich in diesem Heft?

Albert Nijhof betrachtet Mikrocontroller ohne eingebaute Gleitkommaeinheit und gibt eine ganze Reihe von Forth–Worten, die Bruchrechnen erlauben.

Ulrich Hoffmann gibt ein Beispiel für neue Operationen mit bekannten Zahlen mit der **Saturation Arithmetik**.[1]

Man kann sich leicht Anwendungen vorstellen, zum Beispiel eine Drehzahl eines Motors, die sich aus einer Summe von Einflussgrößen, wie Gashebel–Stellung und Last und einer *Abregelung* ergibt.

Matthias Koch löst das Problem, höhere Genauigkeit zu erhalten, ohne eine aufwändige Skalierung durchführen zu müssen, mit seinen **Fixkomma-Zahlen**.

Vor längerer Zeit hat Martin Tracy mit „Zen floating point in one screen“ gezeigt, dass Forth gut für eine knappe Programmierung ist, die das Wesentliche erkennen lässt. Daher zitiert er im Titel den Begriff „Zen“.

Um über die Gleitkommadarstellung von Zahlen und das Rechnen mit ihnen zu schreiben, müssen wir uns einmal viel Zeit nehmen.

Was gibt es noch zu tun?

Wir können, wie es Ulrich Hoffmann getan hat, neue interessante Operationen mit bekannten Zahlen programmieren.

Ein Beispiel ist die maxplus–Algebra. Wir definieren die Operationen $\oplus$ und $\otimes$ über

$$a \oplus b := \max(a, b)$$

$$a \otimes b := a + b.$$

Was hier wie ein Verwirrspiel aussieht, lässt sich beispielsweise zum Optimieren von Fahrplänen[1] anwenden. Man kann sich vergegenwärtigen, dass viele von den Gesetzen für Addition und Multiplikation auch für $\otimes$ und $\oplus$ gelten. Zum Beispiel:

$$a \oplus b = b \oplus a$$

$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

Sonst gibt es noch genügend viele Aufgaben im Bereich der linearen Algebra. Da gibt es Vektoren und Matrizen mit Gleitkommazahlen und komplexen Zahlen als Einträge. Das ist Thema in der Ausbildung (nicht nur) fast jedes Ingenieurs.

Es gibt mehr Arten, Matrizen zu multiplizieren als hier beschrieben werden können.

Es gibt aber auch die Möglichkeit, **exotische Objekte** zu definieren und für sie Operationen zu programmieren, die den uns bekannten Operationen Addition und Multiplikation entsprechen. Wer sich an solche Themen, die aber auch Anwendungen haben, heranwagen will, kann sich mit Galois–Feldern beschäftigen und z. B. eine Reed–Solomon–Kodierung programmieren, die Übertragungsfehler nicht nur erkennt, sondern auch in vielen Fällen korrigiert.

Ferner gibt es jenseits der komplexen Zahlen noch die Quaternionen und Oktonionen, die früher oft als hyperkomplexe Zahlen bezeichnet wurden.

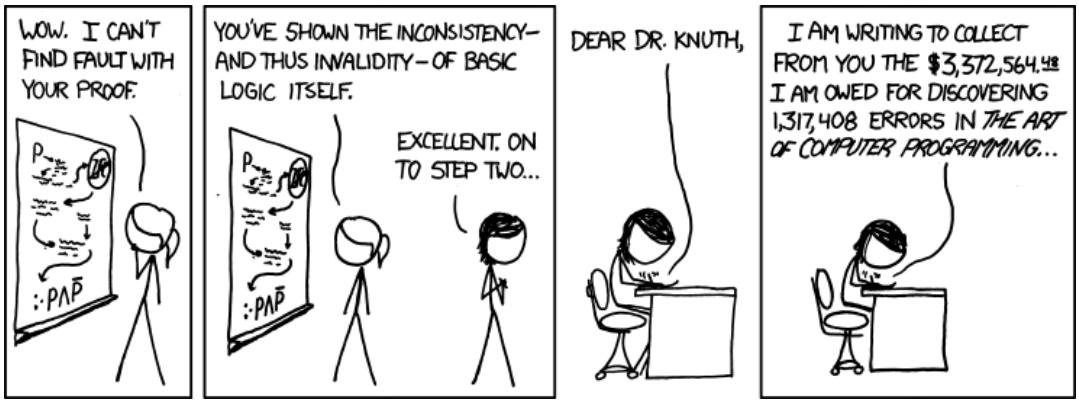
Wie auch die Beispielprogramme in diesem Heft belegen, brauchen nur die wirklich benötigten Operationen programmiert zu werden.

Literatur

[1] Pöppe, Christoph, *„Fahrplan-Algebra“*, Spektrum der Wissenschaft, Digest: Wissenschaftliches Rechnen, Volume 2, Pages 18–21, 1999

[2] Ifrah, Georges, *„Universalgeschichte der Zahlen.“*, Campus-Verlag, 1991

[3] Knuth, Donald E, *„Arithmetik“: Aus der Reihe “The Art of Computer Programming”*, Springer-Verlag, 2013



[1] Der Autor bezieht sich hier auf Hoffmanns Vortrag über Saturation Arithmetic auf der EuroForth 2014. http://www.complang.tuwien.ac.at/anton/euroforth/ef14/papers/

Your Forth has no floating point?

Albert Nijhof

In small Forth applications you can often avoid floating point calculations by scaling properly. Sometimes this is not so easy. For those cases you could try fractions — as is done in these little programs.

The programs . . .

are short and don't depend on each other, it is not necessary to load the whole collection. Just take the one(s) you need. The only exception is SQR2.F

Here are the file names and the subjects:

1. Displaying fractions — `DOT.F DOT-SHORT.F DOT-LAZY.F`
2. Arithmetic — `ARITH.F`
3. Square root — `SQR1.F SQR2.F`
4. SIN COS TAN — `CORDIC.F16 TAYLOR.F16 TAYLOR.F32`

File extensions

.F	for 16– and 32–bit Forth
.F16	for 16–bit Forth; gives 16–bit results in 32–bit Forth
.F32	only for 32–bit Forth

Fraction

A fraction t/n is represented on the stack by `( t n )`, a pair of signed numbers. `t` contains the sign, `n` must be positive.

Errors

The result of an operation can be `t/0`. In that case the value of the fraction is "unknown". Possible causes are:

1. Overflow
2. Trying to divide by zero
3. Trying to take the square root of a negative number
4. The angle for SIN COS or TAN is not in [0,90]
5. The last operation took an already "unknown" argument.

These programs are able to continue their calculations when "unknown" values occur. Of course, the next answer will be "unknown" again. In this way an operation swallows and then reproduces the message. You will get no warnings when this happens.

[1] [rtn] stands for **return** — same as the **enter** key

Displaying fractions, 3 versions

In controller programs, a fraction will be an intermediate result. At the end your aim is to produce a whole number, using `*/` for example. In such a program you may seldom wish to display a fraction, but while developing the program it could be nice. All three versions have `P.R P.` and `PP`

The main word `P.R ( t n r -- )` displays t/n as a decimal fraction with "print precision" `r`. Look at the examples below to grasp the meaning of "print precision". Using `PP` and `P.` is perhaps a bit simpler:

```
4 value PP   \ print precision
: P. ( t n -- ) PP P.R SPACE ;
```

When the denominator is zero ($t/0$):

```
-37 0 P. [rtn] -1/0 ok
```

no message appears.[1]

There are 3 versions:

- `DOT.F`
- `DOT-SHORT.F`
- `DOT-LAZY.F`

`DOT.F` rounds and print precision has no limit.

```
4 to PP
   100 8 P.   [rtn] 12.500
     1 8 P.   [rtn] 0.1250
   1 800 P.   [rtn] 0.0013
   1 8 6 P.R  [rtn] 0.125000
   1 8 2 P.R  [rtn] 0.13
99 100 3 P.R  [rtn] 0.990
99 100 1 P.R  [rtn] 1.0
 10 7 20 P.R  [rtn] 1.42857142857142857143
```

`DOT-LAZY.F` does not round. A silly "+" at the end tells that you should round up.

```
   1 800 P.   [rtn] 0.0012+
   1 8 2 P.R  [rtn] 0.12+
99 100 1 P.R  [rtn] 0.9+
 10 7 20 P.R  [rtn] 1.42857142857142857142+
```

`DOT-SHORT.F` rounds, but can only display a maximum print precision of 9 numerals in 32–bit Forth — which should be sufficient. (4 numerals in 16–bit Forth)

```
 1 8 20 P.R [rtn] 0.125000000 (0.1250)
10 7 20 P.R [rtn] 1.428571429 (1.4286)
```

DOT.F has the most complex code. Therefore, it is absolutely no problem to display PI with it in, let's say, a thousand decimals. ;-)

General arithmetical operations

The file ARITH.F defines *add*, *subtract*, *multiply* and *divide*, and as an option *to the power of*.

P+ P- P* P/ and POWER

```
\ 2/3 + 7/12
2 3 7 12 P+ P. [rtn] 1.250000000 (1.2500)

\ 2/3 - 7/12
2 3 7 12 P- P. [rtn] 0.083333333 (0.0833)

\ 2/3 * 7/12
2 3 7 12 P* P. [rtn] 0.388888889 (0.3889)

\ 2/3 / 7/12
2 3 7 12 P/ P. [rtn] 1.142857143 (1.1429)
```

The burden of scaling numbers is not completely taken away from the programmer. Always be careful that the result of an operation is in the valid region.

```
\ 1,01^30
101 100 30 POWER P. [rtn] 1.347848916 (1.3476)

\ 10^30
10    1 30 POWER P. [rtn] 1/0          (1/0)

\ 10^-30
10 1   -30 POWER P. [rtn] 0.000000000 (0.0000)
```

Positive fractions are in *[1/N,N/1]* with

N = TRUE 1 RSHIFT

Around 1, the "population" of fractions has the greatest density and there the possible precision is optimal. A similar story can be told about negative fractions.

Overflow causes "rounding" to zero or infinity. See how this happens in 32–bit (and in 16–bit) Forth:

```
\  32b: PP=12
\ (16b: PP=6)
TRUE 1 RSHIFT
    DUP . [rtn]  2147483647       \ N
                 (32767)
1 OVER
  2DUP P. [rtn]  0.000000000466   \ 1/N
                 (0.000031)
1 2 P* P. [rtn]  0.000000000000   \ 1/N * 1/2
                 (0.000000)
1 2DUP P. [rtn]  2147483647.000   \ N/1
                 (32767.00)
1 1 P+ P. [rtn]  1/0              \ N/1 + 1/1
                 (1/0)
```

If the numerator or denominator doesn't fit in a cell, REDUCE (dx dy -- t n) is called. It is a sort of automatic scaling, nearly always with some loss of information. On the other hand, calculations are made possible that otherwise would be inexecutable. Even REDUCE is of no help when the expected result is not within the valid region.

The square root

- File SQR1.F defines square root PSQR1 (t n -- t2 n2)
- File SQR2.F defines square root PSQR2 (t n -- t2 n2)

PSQR2 is better than PSQR1, but SQR2.F needs REDUCE (in file ARITH.F). SQR1.F needs no other files.

```
/ 6 1 PSQR P. [rtn] 2.449489743 (2.4495)
  7 1 PSQR P. [rtn] 2.645751310 (2.6458)
```

Of course, the fraction must be positive:

```
-16 1 PSQR P. [rtn] 0/0 ok
```

You get DSQR (du1 -- u2) as an extra, u2 is the (rounded) square root of du1.

```
6 S>D DSQR . [rtn] 2
7 S>D DSQR . [rtn] 3
```

Sin, Cos, Tan

All three files — TAYLOR.F32, TAYLOR.F16 and CORDIC.F16 — expect the angle in degrees as a fraction with a value in [0,90].

Some results in tab.1.

Angle	calculator	Tay32	Tay16	Cor16
11 1 SIN	0.190808995	0.190808996	0.1908	0.1908
33 1 SIN	0.544639035	0.544639035	0.5446	0.5446
44 1 SIN	0.694658370	0.694658371	0.6947	0.6946
55 1 SIN	0.819152044	0.819152045	0.8192	0.8192
66 1 SIN	0.913545457	0.913545458	0.9135	0.9136
88 1 SIN	0.999390827	0.999390827	0.9994	0.9994
91 1 SIN		0/0	0/0	0/0
-1 1 SIN		0/0	0/0	0/0

Table 1: Some results

TAYLOR.F16 uses Taylor series for SIN and COS.

Taylor.F32 takes a few more terms to obtain 32–bit precision.

I have chosen different names, but you can easily replace them by the names you prefer.

- in TAYLOR.F32 — SIN COS TAN
- in TAYLOR.F16 — SN CS TN
- in CORDIC.F16 — SI CO TA

CORDIC.F16 uses the *CORDIC* algorithm. The results with TAYLOR.16 are slightly better than with this CORDIC.F16, but in its kernel the latter only uses the operations add, subtract and right-shift. It doesn't multiply or divide, so it can be written in assembler. I tried to improve the algorithm:

1. A *rounding shift* gives better results: after a multiple right–shift, the last bit that came out is added to the result.
2. Normally, the amount of CORDIC steps is fixed, but results improve when you quit as soon as the desired angle is reached. I do not understand why this is only true in combination with the *rounded shift*. These two seem to reinforce each other.
3. For certain angles, the results are better when a certain CORDIC step is executed twice, but for other angles they get worse, so statistically there is no advantage.

Link

The program files are at `http://home.hccnet.nl/anij`

Listings

arith.f

```
\ Breuken -- Fractions  \ Albert Nijhof, 12feb2007
\ Rekenen -- Arithmetic \ 16 or 32bits

\ P+ P- P* P/    option: POWER

\ rshift until empty hi-cell
: shift>single ( dx -- x i=count )
  0 0 do
    dup if d2/ else drop i leave then
  loop ;

\ dx/dy --> x/y, as good as possible
: reduce ( dx dy -- t n )
  2>r dup r@ xor 0< 2r> rot >r  \ r: signflag
  dabs s>d m+ 2>r dabs s>d m+ 2r>
  2over 2over dnegate d+ nip 0<
  dup >r if 2swap then     \ dlarge dsmall  r: swapflag
  shift>single             \ dlarge small1 i
  swap >r 0 ?do d2/ loop   \ dlarge1        r: small1
  dup dup if drop r@ 1 rshift over / 1 = then
  if   r@ um/mod swap r>
       over - u< 0= - 1      \ large2 1 (ready)
  else shift>single over 0<  \ large3 i hi-bit?
       if   swap dup 1 rshift  ( one more shift )
            swap 1 and + s>d + swap 1+  \ large4
       then
       r> swap
       2dup rshift >r  \ large4 small1 i r: small4
       r@ swap lshift swap \ large4 small1' small1 flag
       over 0<>
       over 0<>
       2over <>
       and and      \ correction of large possible?
       if   >r um* r@ um/mod
            tuck swap r> over - u< 0= - umax
            \ large4a ( = (small1'/small1) * large4 )
       else 2drop     \ large4 (no correction)
       then r>        \ large4(a) small4
  then
  r> if swap then     \ the original order
  r> if swap negate swap then ;

: p+ ( t1 n1 t2 n2 -- t3 n3 )
  rot 2dup m* 2>r
  rot m* 2swap m* d+ 2r> reduce ;
: p- ( t1 n1 t2 n2 -- t3 n3 )
  negate p+ ;
: p* ( t1 n1 t2 n2 -- t3 n3 )
  rot m* 2>r m* 2r> reduce ;
: p/ ( t1 n1 t2 n2 -- t3 n3 )
  swap p* ;

1 [if]
\ Needs P*
: POWER ( t1 n1 m -- t2 n2 )
  dup 0< if abs >r swap r> then
  1 1 2>r
  begin  dup 1 and
         dup if 2over 2r> p* 2>r then
         drop 1 rshift ?dup
  while  >r 2dup p* r>
  repeat 2drop 2r> ;
[then]


cr .( ----- arith.f is geladen ----- )

```

cordic.f16

```
\ Breuken -- Fractions \ Albert Nijhof, jan2007
\ CORDIC               \ 16bits

\ SI CO TA

decimal
: docons does> ( n adr -- value ) swap cells + @ ;
create deltas ( n -- value ) docons  here
32400 , 19127 , 10106 , 5130 , 2575 , 1289 ,
645 , 322 , 161 , 81 , 40 , 20 , 10 , 5 , 3 , 1 ,
here swap - 1 cells / constant #deltas

create noemers docons           28084 , 31399 ,
32365 , 32617 , 32681 , 32697 , 32701 , 32702 ,

: arshift ( x1 i -- x2 )  \ afrondend
  ?dup 0= if exit then   swap s>d >r abs swap
  1- rshift negate 2/    r> if exit then negate ;

: cordic-stap ( s c -h i -- s' c' -h' i' )
  dup >r
  over 0<
  if   2>r 2dup r@ arshift +
       swap rot r@ arshift -
       2r> deltas +
  else 2>r 2dup r@ arshift -
       swap rot r@ arshift +
       2r> deltas -
  then r> ;

: cordic ( -h -- s c z )
  0 dup noemers   rot  0            \ s c -h i
  begin  2dup 1+ #deltas < and
  while  1+ cordic-stap
  repeat nip 7 min noemers >r
  r@ min swap 0 max swap r> ;

: sicota ( t n -- s c z | 0 false false )
  over 0< 0= over 0> and if
     over 1- over / 90 < if
\  [ 0 deltas 45 / ] literal swap u*//
  [ 0 deltas 45 / ] literal dup >r swap */mod
                    swap r> over - < 0= -  \ afronden
  dup  [ 0 deltas ] literal u< if
                    negate cordic exit then
  [ 0 deltas 2*   ] literal -  cordic >r swap r> exit
                    then then 2drop false dup dup ;

: ta sicota drop ;
: si sicota nip ;
: co sicota rot drop ;

.(  ---- cordic.f16 is geladen ----- )

```

dot.f

```
\ Breuken   -- Fractions \ Albert Nijhof, jan2007
\ Afdrukken -- Display   \ 16 or 32bits

\ P.R  PP  P.

: >char ( x -- ch )
  9 over < 7 and + [char] 0 + ;


\ Int     = integer, getal voor de komma,
            -1 = al afgedrukt.
\ NewDig  = het laatst berekende cijfer.
\ OldDig  = het vorige cijfer, -1 = geen vorig cijfer.
\ #negens = aantal uitgestelde negens.
\ Noi     = ( #negens OldDig Int )

: .reservoir ( Noi 9|0 -- )
  >char >r
  ( Int )      invert ?dup if invert 0 .r ." ." then
  ( OldDig )   invert ?dup if invert >char emit then
  r> swap ( "9|0" #negens ) 0 ?do dup emit loop drop ;

: p.r ( t n pp -- )
  >r 2dup xor 0< if ." -" then
     dup 0= if r> 2drop if ." 1/0" exit then
                        ." 0/0" exit then
     0 -1 2swap abs
     swap abs over /mod \ 0 -1  n Rest Int
  r> over 0 <# #s #> nip - 0 max
  2swap rot 0        \ #negens OldDig Int n Rest # 0
  ?do  2>r           \              r: n Rest
       r> base @ um* r@ um/mod swap   \ Volgende cijfer
       >r dup base @ 1- =             \ Een negen?
       if   drop 2>r 1+ 2r>           \ Verhoog #negens
       else >r base @ 1- .reservoir 0 r> -1
       then 2r>             \ #negens OldDig Int n Rest
  loop
  swap >r base @ um* r@ um/mod
  swap r> over - u< 0= -  \ Laatste cijfer
  dup base @ =            \ 10 geworden door afronding?
  if   drop 0 >r >r s>d
       if   r> 1+         \ Verhoog Int
       else 1+ r>         \ Verhoog OldDig
       then 0
  else >r base @ 1-
  then .reservoir r> >char emit ;

\ "Print precision", may be changed by the user.
-1 0 <# #S #> nip 1- value pp

: p. ( y n -- )
  pp p.r space ;

cr .( ----- dot.f is geladen ----- )
```

dot-lazy.f

```
\ Breuken   -- Fractions \ Albert Nijhof, jan2007
\ Afdrukken -- Display   \ 16 or 32bits,
                         \ Lazy Programmer version

\ P.R  P.  PP

: p.r ( t n pp -- )
  >r 2dup xor 0< if ." -" then
     dup 0= if r> 2drop
              if ." 1/0" exit then
                 ." 0/0" exit then
     abs swap abs over /mod            \ n Rest int
     0 <# #s #> tuck type [char] . emit
  r> 1+ swap - 1 max 0
  do   over >r base @
       um* r> um/mod        \ n Rest' Volgende-cijfer
       9 over < 7 and + [char] 0 + emit
  loop
  base @ um* rot um/mod nip            \ Laatste cijfer
  base @ over - < if exit then ." +" ; \ afronden?

\ "Print precision", may be changed by user.
-1 0 <# #S #> nip 1- value pp

: p. ( t n y -- )
  pp p.r space ;

cr .( ----- dot-lazy.f is geladen ----- )
```

dot-short.f

```
\ Breuken   -- Fractions \ Albert Nijhof, jan2007
\ Afdrukken -- Display   \ 16 or 32bits, Short version

\ P.R  P.  PP

\ pp = "printprecision" (16bits:max=4,  32bits:max=9)
: p.r ( t n pp -- )
  [ -1 0 <# #S #> nip 1- ] literal
                          umin >r   \ t n  r: pp'
  2dup xor 0< if ." -" then
  dup 0= if r> 2drop
           if ." 1/0" exit then
         ." 0/0" exit then
  abs swap abs                      \ n t
  base @ 1 r@ 0 ?do over * loop nip  \ n t 10^pp
  um* rot                           \ dx=t*10^pp  n
  >r 0 r@ um/mod r> 2>r
  r@ um/mod 0 rot r>
  over - < 0= - r> swap s>d d+     \ dq= dx/n (rounded)
  <# r> 0 ?do # loop
  [char] . hold #s #> type ;

\ "Print precision", may be changed by user.
-1 0 <# #S #> nip 1- value pp

: p. ( t n -- )
  pp p.r space ;

cr .( ----- dot-short.f is geladen ----- )
```

sqr1.f

```
\ Breuken         -- Fractions   \ Albert Nijhof, 12feb2007
\ Worteltrekken 1 -- Square root 1 \ 16 or 32bits

\ This file SQR1.F needs no other files.

\ PSQR1

\ PSQR2 (in file SQR2.F) is better than this PSQR1

\ u2 is rounded unless FFFF(FFFF)
: dsqr ( du1 -- u2 ) \ sqr(du1) = u2
  2dup 0= negate 2* u< if drop exit then
  -3 over u< if 2drop -1 exit then
  2>r  r@ 1+                \ first-guess = g1
  begin
    dup 2r@ rot um/mod nip   \ g1 du/g1
    1 rshift over 1 rshift + \ better:
                             \ g2=(du1/g1)/2 + g1/2
    tuck - abs 2 u<          \ |g2-g1|<2?
  until dup dup 1+ um*
  dup r@ =                   \ g*(g+1) < du1 ?
  if   drop 2r> drop
  else nip  2r>  nip
  then u< - ;                \ Add 1 when true

```

```
: psqr1 ( t1 n1 -- t2 n2 )   \ t2/n2 = sqr t1/n1
  2dup or 0< if 2drop false dup then
  2dup < dup >r  if swap then  \ big small r: swapflag
  dup
  if   swap 0 0
    do  [ -1 1 rshift dup 1 rshift xor ] \ 01000000...
        literal over and
        if   swap i  lshift leave        \ big small
        then 2* \ shift left as far as possible
                \ for best precision
       loop over um* dsqr    \ X  sqr(Y*X)
  then r> if swap then ;

\ sqr x/y = X / sqr Y*X
\ for X = x*(2^i) and Y = y*(2^i)

cr .( ----- sqr.f is geladen ----- )
```

sqr2.f

```
\ Breuken         -- Fractions   \ Albert Nijhof,
                                 \ 12feb2007
\ Worteltrekken 2 -- Square root 2 \ 16 or 32bits

\ !  This file needs REDUCE from file ARITH.F

\ PSQR2

\ This PSQR2 is better than PSQR1 in file SQR1.F

\ u2 is rounded unless FFFF(FFFF)
: dsqr ( du1 -- u2 ) \ sqr(du1) = u2
  2dup 0= negate 2* u< if drop exit then
  -3 over u< if 2drop -1 exit then
  2>r  r@ 1+                     \ first-guess = g1
  begin dup 2r@ rot um/mod nip   \ g1 du/g1
        1 rshift over 1 rshift + \ better:
                                 \ g2=(du1/g1)/2 + g1/2
        tuck - abs 2 u<          \ |g2-g1|<2?
  until dup dup 1+ um*
  dup r@ =                       \ g*(g+1) < du1 ?
  if   drop 2r> drop
  else nip  2r>  nip
  then u< - ;                    \ Add 1 when true

: PSQR2 ( t1 n1 -- t2 n2 ) \ t2/n2 = sqr t1/n1
  2dup or 0< if 2drop false dup then
  2dup < dup >r  if swap then  \ big small  r: swapflag
  dup
  if  over um*                \ b db*s
      0 0
      do
       [ -1 1 rshift dup 2 rshift xor ]  \ 011000000...
       literal over and
       if   rot 0 i leave   \ db*s db i
       then d2* d2*         \ lshift as far as possible
       loop
       0 ?do d2* loop       \ db*s db
       2swap dsqr 0 reduce  \ db sqr(db*s)
  then r> if swap then ;


\ sqr x/y =  X / sqr Y*X
\ for X = x*(2^i) and Y = y*(2^i)

cr .( ----- sqr.f is geladen ----- )
```

taylor.f16

```
\ Breuken -- Fractions \ Albert Nijhof, jan2007
\ Taylor               \ 16bits

\ SN CS TN  option:  PI

\ rounding unsigned division
: um//  ( du1 u2 -- u3 )
  dup >r um/mod tuck swap r> over - u< 0= - umax ;

\ rounding unsigned star-slash
: u*//  ( u1 u2 u3 -- u4 )
  >r um* r> um// ;

decimal
\ : pi 355 113 ;
25732 constant q1
32763 constant q2   \  q1/q2 = pi/4

: >rad ( t n -- h complement-vlag )   \ x/y in [0..90]
  over 0< 0= over 0> and    if
  over 1- 0 max over / 90 < if
  >r q1 um* r> tuck  0 swap um/mod
  >r swap um// r> 45 um//        \ (x * q1) / (y * 45)
  dup q1 u< if false exit then   \ h 0
  q1 2* swap - true exit         \ h-complement -1
                           then
                           then
  2drop true false ;

: (sin ( h -- z1 q2 )  \  h/q2 in [0..pi/4] rad.
  s>d if drop false dup exit then
  dup q2 dup >r 2>r      \ h            r: q2 h q2
  dup 2r@ u*//           \ h   ^2
  r> 2>r                 \ h            r: q2 h ^2 q2
  2r@ u*//               \ ^3
  dup 2r@ u*//           \ ^3 ^5
  dup 2r> u*//           \ ^3 ^5 ^7     r: q2 h
  r>                     \              r: q2
  swap 0     5040 um//        ( ." S" dup . ) -
  swap 0      120 um// +
  swap 0        6 um// - 0 max r> ;

: (cos ( h -- z2 q2 )  \  h/q2 in [0..pi/4] rad.
  s>d if drop false dup exit then
  q2 >r                  \ h            r: q2
  dup r@ u*//            \ ^2
  dup r@ 2>r             \ ^2           r: q2 ^2 q2
  dup 2r@ u*//           \ ^2 ^4
  dup 2r> u*//           \ ^2 ^4 ^6     r: q2
  r@
  swap 0     720 um//         ( ." C" dup . ) -
  swap 0      24 um// +
  swap 0       2 um// - 0 max r> ;

\ t/n in [0..90]
: sn ( t n -- z1 q2 )
  >rad if (cos exit then (sin ;

: cs ( t n -- z2 q2 )
  >rad if (sin exit then (cos ;

: tn ( t n -- z1 z2 )
  >rad >r dup (sin drop swap (cos drop
       r> if swap then ;

cr .( ----- Taylor.f16 is geladen ----- )
```

taylor.f32

```
\ Breuken -- Fractions \ Albert Nijhof, jan2007
\ Taylor               \ 32bits
```

```
3
4   \ SIN COS TAN   option:  PI
5
6   \ rounding unsigned division
7   : um//  ( du1 u2 -- u3 )
8     dup >r um/mod tuck swap r> over - u< 0= - umax ;
9
10  \ rounding unsigned star-slash
11  : u*//  ( u1 u2 u3 -- u4 )
12    >r um* r> um// ;
13
14  decimal
15  \ : pi 103993 33102 ;
16  1686629695 constant q1
17  2147483625 constant q2   \  q1/q2 = pi/4
18
19  : >rad ( t n -- h complement-vlag )  \ x/y in [0..90]
20    over 0< 0= over 0> and    if
21    over 1- 0 max over / 90 < if
22    >r q1 um* r> tuck  0 swap um/mod
23    >r swap um// r> 45 um//      \ (x*q1)/(y*45)
24    dup q1 u< if false exit then \ h 0
25    q1 2* swap - true exit       \ h-complement -1
26                               then
27                               then
28    2drop true false ;
29
30  : (sin ( h -- z q2 )  \  h/q2 in [0..pi/4] rad.
31    s>d if drop false dup exit then
32    dup q2 dup >r 2>r   \ h       r: q2 h q2
33    dup 2r@ u*//        \ h  ^2
34    r> 2r               \ h       r: q2 h ^2 q2
35    2r@ u*//            \ ^3
36    dup 2r@ u*//  dup 2r@ u*//    \ ^3 ^5 ^7
37    dup 2r@ u*//  dup 2r> u*//    \ ^3 ^5 ^7 ^9 ^11
38                                 \ r: q2 h
39    r>                           \ r: q2
40    swap 0 39916800 um// -  swap 0   362880 um// +
41    swap 0     5040 um// -  swap 0      120 um// +
42    swap 0        6 um// -  0 max r> ;
43
44  : (cos ( h -- z q2 )  \  h/q2 in [0..pi/4] rad.
45    s>d if drop false dup exit then
46    q2 >r               \ h         r: q2
47    dup r@ u*//         \ ^2
48    dup r@ 2>r          \ ^2        r: q2 ^2 q2
49    dup 2r@ u*//  dup 2r@ u*//    \ ^2 ^4 ^6
50    dup 2r@ u*//  dup 2r> u*//    \ ^2 ^4 ^6 ^8 ^10
51                                 \ r: q2
52    r@
53    swap 0 3628800 um// -  swap 0   40320 um// +
54    swap 0     720 um// -  swap 0      24 um// +
55    swap 0       2 um// -  0 max r> ;
56
57  \ t/n in [0..90]
58  : sin ( t n -- z1 q2 )
59    >rad if (cos exit then (sin ;
60  : cos ( t n -- z2 q2 )
61    >rad if (sin exit then (cos ;
62  : tan ( t n -- z1 z2 )
63    >rad >r dup (sin drop swap (cos drop
64         r> if swap then ;
65
66  cr .( ----- Taylor.f32 is geladen ----- )
```

Vintage Computing — Zen Floating Point

```
SCREEN   #18.6
\ Very compressed one-screen floating-point       21NOV 84MJT
: D10* D2* 2DUP D2* D2* D+ ; : D+- 0< IF DNEGATE THEN ;
: TRIM >R TUCK DABS BEGIN OVER 0< OVER OR WHILE 0 10 UM/MOD >R
  10 UM/MOD NIP R> R> 1+ >R REPEAT ROT D+- DROP R> ;
: F+  ROT 2DUP - DUP 0< IF NEGATE ROT >R NIP >R SWAP R> ELSE
  SWAP >R NIP THEN >R S>D R> DUP 0 ?DO >R D10* R> 1- OVER ABS
  6553 > IF LEAVE THEN LOOP R> OVER + >R IF ROT DROP ELSE ROT
  S>D D+ THEN R> TRIM ;
: FNEGATE  >R NEGATE R> ; : F- FNEGATE F+ ;
: F*  ROT + >R 2DUP XOR >R ABS SWAP ABS UM* R> D+- R> TRIM ;
: F/  OVER 0= ABORT" 0/" ROT SWAP - >R 2DUP XOR -ROT ABS DUP
  6553 MIN  ROT ABS 0 BEGIN 2DUP D10* NIP 3 PICK < WHILE D10*
  R> 1- >R REPEAT 2SWAP DROP UM/MOD NIP 0 ROT D+- R> TRIM ;
: FLOAT  DPL @ NEGATE TRIM ; : F.  >R DUP ABS 0 <# R@ 0 MAX 0
  ?DO ASCII 0 HOLD LOOP R@ 0< IF R@ NEGATE 0 MAX 0 ?DO # LOOP
  ASCII . HOLD THEN R> DROP #S ROT SIGN #> TYPE SPACE ;
```

Martin Tracy hat 1984 das Kunststück geschafft, die basalen Floating–Point–Routinen in einen einzigen Forth–Screen zu packen! Gut, Kommentare und Erklärungen passten da nicht auch noch rein. Wer also wissen wollte, was es damit auf sich hatte und wie man das verwendete, brauchte auch damals schon die kommentierte fünf Screens lange Variante aus *Dr. Dobb's Toolbook of Forth, Volume II*. In Kapitel 18 beschreibt Martin Tracy das kleine Paket aus vier Fließkomma–Funktionen einfacher Rechen–Genauigkeit (16 Bit) und mit vier signifikanten Nachkommastellen. Diese wurden durch zwei Werte auf dem Stack, einer Mantisse mit Vorzeichen und darüber der Zehner–Exponent repräsentiert. Ihr seht, das Prinzip ist geblieben bis heute.

Die Verwendung ging einfach so:

```
3.1415 FLOAT   12.5 FLOAT   F*  F.
```

Dazu schrieb er damals:

> „Fun — yes, it is fun. Imagine silencing critics who say that Forth has no floating–point by demonstrating that it can be done in one screen! But don't try to impress them with its speed (slow) or accuracy (about three digits). For more fun, see Dr. Nathanial Grossman's *Zen Slide Rule* (1984 FORML), which adds trancendentals and exponentials!"

Quelle: Dr. Dobb's Toolbook of Forth, Volume II, 1987 by M&T Publishing, Inc., California. Seite 261ff mk

Fixkommazahlen nach Art des Hauses: s31.32

Matthias Koch

Immer wieder stellt sich die Aufgabe, die mit Forth gesammelten Messwerte auch sogleich im Mikrocontroller weiterzuverarbeiten. Traditionell werden passend skalierte Ganzzahlen dafür verwendet — doch manchmal stößt die Methode an Grenzen, vor allem, wenn nur schnell mal etwas ausprobiert werden soll oder mathematische Funktionen benötigt werden, die über die Grundrechenarten hinausgehen.

Für genau diese Fälle bieten Mecrisp–Stellaris und Mecrisp–Quintus eingebaute Unterstützung für das s31.32–Fixkommazahlenformat an, welches in diesem Artikel einmal gründlich beschrieben werden soll, nachdem es schon so lange im Einsatz ist.

Aufbau einer s31.32–Fixkommazahl

Der grundsätzliche Aufbau ist — wie so vieles Nützliche — recht einfach: Eine *s31.32–Fixkommazahl* wird mit Komma eingegeben und liegt als Double auf dem Stack, wobei *s31*, also der Ganzzahlteil, obenauf liegt. Die Nachkommastellen *.32* liegen an zweiter Stelle auf dem Stack. Jedes `n` kann also mit Hilfe von `0 swap` in eine Fixkommazahl verwandelt werden. Mit `f.` schließlich wird eine Fixkommazahl in der aktuellen Basis ausgegeben.

```
42,0 . . 42 0 ok.
0 1 f. 1,00000000000000000000000000000000 ok.
```

Mit dem Ganzzahlteil (der im Double dem High–Teil entsprechen würde) geschieht also nichts Verwunderliches.

Schauen wir uns die Nachkommastellen an[1]:

```
decimal 0,5 hex u. u. 0 80000000 ok.
decimal 0,25 hex u. u. 0 40000000 ok.
decimal 0,125 hex u. u. 0 20000000 ok.
decimal 1 0 f.
0,00000000023283064365386962890625 ok.
```

Die Bits in den Nachkommastellen entsprechen also der Reihe nach 1/2, 1/4, 1/8 ... bis hin zu 2^{-32}.

Zusammen mit dem allergrößten möglichen Wert ist der verwendbare Wertebereich bestimmt:

```
$7FFFFFFF,FFFFFFFF f.
2147483647,99999999976716935634613037109375 ok.
```

Mehr geht nicht, denn das oberste Bit gibt schließlich im Zweierkomplement das Vorzeichen an:

```
$80000000,00000000 f.
-2147483648,00000000000000000000000000000000 ok.
```

Handhabung auf dem Stack

Da es sich — von der Bedeutung der Bits abgesehen — um Zweierkomplement–Doubles handelt, können sie wie gewohnt auf dem Stack herumjongliert sowie mit `d+` und `d-` addiert und subtrahiert werden. Auch die anderen altbekannten Helferlein wie `dnegate`, `dabs`, `d2*`, `d2/`, `d=`, `d<>`, `d>` und `d<` funktionieren genau so, wie der Leser (oder die Leserin) es sich jetzt wohl denkt.

Bei der Punktrechnung muss jedoch darauf geachtet werden, dass das *Komma* in der Mitte bleibt — zu diesem Zweck sind `f*` und `f/` an Bord:

```
3,141592653589793 2constant pi ok.
pi 0,5 f* f.
1,57079632673412561416625976562500 ok.
```

Wie war das mit der Umrechnung von Bogenmaß in Grad?

```
: rad>grad ( f-rad -- f-grad )
  180,0 f* pi f/ ; ok.
1,5 pi f* rad>grad f.
270,00000000000000000000000000000000 ok.
```

Zahlenausgabe

Wer sich wundert, wieso `f.` immer so viele Stellen ausgibt, soll daran erinnert werden, dass in guter Forth–Tradition natürlich auch andere Zahlenbasen verwendet werden können.

```
hex pi f.
 3,243F6A8800000000000000000000000 ok.
binary pi f.
11,00100100001111110110101010001000 ok.
```

Im Binärformat werden schließlich alle 32 Nachkommastellen tatsächlich benötigt. Wem das zu lang ist, der mag stattdessen `f.n` verwenden, wobei die gewünschte Zahl der Nachkommastellen mit angegeben werden kann:

```
decimal pi 5 f.n 3,14159 ok.
```

Für ganz besonders gestaltete Zahlenausgaben wurde der ohnehin schon mächtige Zahlenausgabe-Baukasten von Forth noch um drei Zutaten erweitert:

`hold<`	(Zeichen --) Fügt ein Zeichen von hinten an den Zahlenausgabepuffer an
`f#`	(n -- n) Fügt eine Nachkommastelle an
`f#s`	(n -- n) Fügt 32 Nachkommastellen an

Ein kleines Beispiel im Vergleich zum alten Bekannten `d.` hilft bestimmt:

[1] In den nachfolgenden Beispielen wurden die langen Fixkommazahlen in die nächste Zeile gerückt, damit sie ins zweispaltige Heftformat passen. Im Terminal folgt die Ausgabe natürlich gleich auf das `f.`

```
: d.  ( d -- )
      tuck dabs <# #s rot sign #> type space ;
: f.3 ( n-komma n-ganz )
      tuck     ( n-ganz n-komma n-ganz )
      dabs     ( n-ganz u-komma u-ganz )
      0        ( n-ganz u-komma u-ganz 0 )
      <# #s    ( n-ganz u-komma 0 0 )

     [char] , hold<
     rot       ( n-ganz 0 0 u-komma )
     f# f# f#  ( n-ganz 0 0 u-komma )
     drop      ( n-ganz 0 0 )
     rot       ( 0 0 n-ganz )
     sign      ( 0 0 )

     #> type space ;
```

Zunächst wird das Vorzeichen abgetrennt; die Vorkommastellen werden ihrerseits in eine Double–Zahl verwandelt und wie gewohnt bearbeitet. Anschließend wird — von hinten — ein Komma angefügt, worauf drei Nachkommastellen folgen. Ganz zum Schluss wird das Vorzeichen angefügt.

Und, wie gewünscht, ergibt sich

```
pi f.3 3,141 ok.
```

wobei allerdings abgeschnitten und nicht gerundet wird, da die korrekte Rundung in beliebigen anderen Basen Kopfzerbrechen bereiten dürfte.

Hier noch einmal zum Vergleich `#` und `f#`:

```
: #  ( ud -- ud' )
  base @ 0 ud/mod 2swap drop .digit hold  ;
: f# ( u  -- u'  )
  base @   um*               .digit hold< ;
```

Während bei einer Ganzzahl durch die Basis geteilt wird und der Divisionsrest als Ziffer ausgegeben wird, muss bei Nachkommastellen entsprechend mit der Basis multipliziert und die neue Ziffer von hinten angefügt werden.

Zahleneingabe

Die Zahleneingabe ist ein bisschen ungewöhnlich gelöst, was schon am Stackkommentar zu sehen ist:

```
number ( c-addr length -- 0 )
                       -- n 1 )
                       -- n-low n-high 2 )
```

`number` ist eine recht große Assembler–Routine aus einem Guss, in der mehrere Zahlenformate erkannt werden, wobei die Länge der umgewandelten Zahl in Zellen zurückgegeben wird. Eine auf dem Stack null Zellen lange Zahl war ein Fehler, der String konnte also gar nicht als Zahl interpretiert werden. Gewöhnlicherweise belegen Zahlen in Forth eine Zelle, und falls ein `.` (Double) oder ein `,` (Fixkomma) enthalten gewesen ist, kümmert sich `number` um die richtige Deutung der Ziffern und gibt ein zwei Zellen langes Ergebnis auf dem Stack zurück.

Genauigkeit

Im Vergleich zum weitverbreiteten Fließkommaformat lassen sich die Fixkommazahlen auch in Prozessoren ohne Hardware–Fließkommaeinheit effizient implementieren. Doch der größte Vorteil ist, dass sich im Fixkommaformat bei der Addition und Subtraktion keine Rundungsfehler einschleichen. Diese Eigenschaft ist besonders dann nützlich, wenn Werte nummerisch integriert werden müssen — Fließkommazahlen haben nämlich das Problem, dass bei einer Addition einer sehr großen mit einer sehr kleinen Zahl die sehr kleine Zahl einfach weggerundet werden kann.

Zwei Fließkommaformate sind verbreitet:

- *IEEE 754 Single* — 32 Bits lang, wobei 1 Bit Vorzeichen ist, 8 Bits für den Exponenten verwendet werden und 23 Bits für die Mantisse, also die eigentlichen Ziffern.

- *IEEE 754 Double* — 64 Bits lang, 1 Bit füs Vorzeichen, 11 Bits für den Exponenten und 52 Bits für die Mantisse.

Im Vergleich dazu hat *s31.s32* ein Vorzeichenbit und 63 Bits für die Mantisse.

Leider ist in allen derzeit von Mecrisp-Stellaris unterstützten Mikrocontrollern bestenfalls eine Fließkommaeinheit für *IEEE 754 Single* eingebaut, die im Dezimalsystem höchstens 11 * log(2)/log(10) = 3.3113, also insgesamt 3 geltende Ziffern Genauigkeit aufweist. Mit dem in Mecrisp–Stellaris und Mecrisp–Quintus vorhandenen *s31.32* ergeben sich — passende Skalierung vorausgesetzt — mit 63 * log(2)/log(10) = 18.965 insgesamt 18 geltende Ziffern Genauigkeit — und das ohne weitere Rundungsfehler bei der Addition und Subtraktion!

Wenn also der verfügbare Wertebereich zwischen 2^{31} = 2,1475 * 10^{9} und 2^{-32} = 2,3283 * 10^{-10} genügt, was für einen mit Skalierung aufgewachsenen Forth–Programmierer keine Schwierigkeit darstellen sollte, steht einer gelungenen Kalkulation nichts mehr im Wege.

Ergänzende Fixkomma–Bibliotheken

Zu guter Letzt gibt es momentan zwei unterschiedliche nachladbare Bibliotheken mit mathematischen Funktionen, welche die Arbeit mit dem *s31.32*–Fixkommaformat auch bei komplizierten Rechnungen angenehm gestalten. Meine eigene (`cordic.txt`) ist besonders für die digitale Signalverarbeitung geeignet, erwartet Argumente im Bogenmaß und basiert auf dem Cordic–Algorithmus, welcher eine Auswahl trigonometrischer Funktionen zur Verfügung stellt. Andrew Palm hat sich die Mühe gemacht, eine umfangreichere Fixkomma–Bibliothek (`fixpt-math-lib.fs`) unter Verwendung vieler verschiedener Algorithmen zusammenzutragen, die sehr komfortabel in der Verwendung ist und bei der Winkel in Grad angegeben werden.

Beispiele

Ein kleines Anwendungsbeispiel soll verdeutlichen, was damit möglich wird:

Bei mir in der Werkstatt hängt der Stamm meines alten Flieders an Ketten baumelnd über der Werkbank. In eine ins Holz hineingestemmte Nut ist ein Leuchtdiodenband eingelassen und an einem Ende ist ein Mikrocontrollerboard mit einem Beschleunigungssensor, welcher von Mecrisp–Stellaris ausgelesen wird. Wenn ich den Stamm ein kleines bisschen drehe, dimmt die Lampe — dafür brauche ich eine Funktion, welche anhand der Richtung der Schwerkraft den Auslenkungswinkel der Lampe bestimmt:

```
: ruhewinkel-bestimmen ( -- )
  accel-mittelung ruhe-z ! ruhe-y ! ruhe-x !
;

: auslenkung ( -- f-winkel )
  accel-mittelung g-z ! g-y ! g-x !

  \ Winkel zwischen der Ruhelage
   \ und der aktuellen Lage

    ruhe-x @ g-x @ *
    ruhe-y @ g-y @ * +
    ruhe-z @ g-z @ * +
    s>f

    ruhe-x @ dup *
    ruhe-y @ dup * +
    ruhe-z @ dup * +
    s>f sqrt

    g-x @ dup *
    g-y @ dup * +
    g-z @ dup * +
    s>f sqrt

    f* f/ acos
;

: drehrichtung ( -- ? )
  g-x @ ruhe-x @ <
;
```

Wer nicht sogleich darauf kommt, was hier geschieht, möge sich die Definition des Skalarproduktes in Wikipedia anschauen. Dann fällt der Groschen bestimmt!

Und wer ein richtig umfangreiches Beispiel sehen möchte, mag in `sunrise-sunset.fs` eintauchen, worin Andrew Palm unter Verwendung seiner Bibliothek und astronomischer Gleichungen nach Eingabe von Koordinaten, Datum und Zeitzone die Zeitpunkte von Sonnenauf- und Untergang berechnet.

PS: Das in Mecrisp für MSP430 verwendete *s15.16–Format* funktioniert analog, nur mit einem entsprechend kleineren Wertebereich:

```
       1 0 f.     0,0000152587890625  ok.
$7FFF,FFFF f. 32767,9999847412109375  ok.
```

Links

`http://hightechdoc.net/mecrisp-stellaris/_build/html/fixed-point.html`

... und noch eine Meldung:

Forth–Bibliotheken

In seinem Github–Repository hat Lars-Erik Svahn eine Reihe von Forth-Bibliotheken im Quellcode gesammelt. Diese Bibliotheken sind für ANS-Forth geschrieben und getestet. Sie decken mathematische Funktionen und String-Manipulationen ab.

alphasets.4th Sets of strings

bigintegers.4th Stackbased unsigned big operations

calendar.4th Modified Julian calendar

gaussian.4th Small Gaussian numbers

nestedsets.4th Stack based nested sets of non–negative integers

numbertheory.4th Basic unsigned single words on number theory

primecounting.4th Fast primes, squeezed prime tables

signedbig.4th Signed big integers

simplestring.4th Storing and concatenating strings

stringstack.4th Stack and stackoperations for strings

tautologies.4th Testing tautologies

`https://github.com/Lehs/ANS-Forth-libraries`

Carsten Strotmann

Fortsetzung auf Seite 26

Fixpoint Math Library

Andrew Palm

The following comments are a text excerpt from Andrews `fixpt-math-lib.fs` (2018.04.09) *to make it more readable in our magazine. I took great care when formatting the code to keep it functional. The code was written for Mecrisp.[mk]*

Comments on sqrt and trig functions

All angles are in degrees.

Accuracy is good rounded to 7 significant digits, with some exceptions. In particular, the `asin` and `acos` functions have reduced accuracy near the endpoints of the range of their inputs (± 1) due to their very large slopes there. See the tests in `fixpt-mat-lib-tests.fs`.

The `sin` function is based on *Maclaurin series* for sin and cos over the interval $[0, \pi/4]$, evaluated as polynomials with the *Horner method*. This is extended to all angles using (anti)symmetry. Cos is calculated from sin with $\cos(x) = \sin(x + 90)$, and tan is calculated as sin/cos.

Atan is based on the first 7 terms of its *Euler series* over the interval $[0, 1/8]$. It is extended to $[0, 1]$ using the identity

$\arctan(x) = \arctan(c) + \arctan\left(\frac{x-c}{1+x*c}\right), c = 1/8, 2/8, ..., 7/8, 1.$

For $x > 1$ we use $\arctan(x) = 90 - \arctan(\frac{1}{x})$. Negative arguments are handled by antisymmetry. Asin and acos are calculated using the formulas

$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right), x^2 <= \frac{1}{2}$

$\arcsin(x) = 90 - \arctan\left(\frac{\sqrt{1-x^2}}{x}\right), x^2 > \frac{1}{2}$

$\arccos(x) = 90 - \arcsin(x)$

The square root is calculated bitwise with a standard algorithm over the interval $[0, 1]$ and is extended to all positive x by division by 4 until the quotient is in $[0, 1]$.

Comments on the log and power functions

The user can check for accuracy by running the test functions in the file `fixpt-math-lib-tests.fs`, or by running tests tailored to their use. Generally, the functions are accurate when rounded to about 7 significant digits. However, the user should not expect good accuracy when dealing with very small fractional values due to the limitations of fixed point. In particular, this affects the values of the power and exponential functions for larger negative inputs, when the relative accuracy decreases significantly.

If the argument to a log function is non–positive, the function returns "minus infinity", the largest negative s31.32 value. This is the only signal that an invalid input has been used. Large negative inputs to the power and exponential functions will return zero. Large positive inputs will return "plus infinity", the largest positive s31.32 value. The code shows the specific values used to determine "large" in each case.

The algorithm for calculating the *base*–2 log is taken from pseudocode in the Wikipedia article "Binary logarithm" which is based on [1].

The $\log_{10}(x)$ and natural logarithm $\ln(x)$ make use of the identities

$\log_{10}(x) = \log_{10}(2) * \log_2(x), \ln(x) = \ln(2) * \log_2(x)$

where $\log_{10}(2)$ and $\ln(2)$ are given constants.

The $\text{pow}_2(x) = 2^x$ function is calculated as

$2^x = (2^z) * (2^n)$

where $x = z + n, n$ is an integer with

$n <= x < n + 1$, and $0 <= z < 1$.

The factor 2^z is calculated by the identity

$2^z = \exp(\ln(2) * z)$

where $\exp(y)$ is calculated using its Maclaurin series. The other factor is accounted for by shifting 2^z n times (shift left for $n > 0$, shift right for $n < 0$).

The $\text{pow}_{10}(x) = 10^x$ function is calculated using the identity

$10^x = 2^{x * \frac{\ln(10)}{\ln(2)}}$

except for positive integer values of x, where simple multiplication is used.

The $\exp(y)$ function is calculated using the series above if y is between -0.36 and $+0.36$. Otherwise it is calculated from pow2 using the identity

$\exp(y) = \text{pow}_2\left(\frac{y}{\ln(2)}\right)$.

Note: Some s31.32 constant values were rounded from theoretical values and entered below as (comma–part) integers rather calculating them using Forth conversions, which truncate.

Literatur

[1] Majithia, J. C.; Levan, D. (1973), *"A note on base–2 logarithm computations",* Proceedings of the IEEE, 61 (10): 1519–1520, doi:10.1109/PROC.1973.9318

Listing

```
\ ================================================
\  File: fixpt-math-lib.fs for Mecrisp-Stellaris
\
\  This file contains these functions for s31.32
\  fixed point numbers:
\           sqrt, sin, cos, tan, asin, acos, atan
\           log2, log10, ln, pow2, pow10, exp
\
\ ------------------------------------------------
\  Misc. helper words, constants, and variables
\ ------------------------------------------------

\ Most positive and negative s31.32 values possible
\ 2147483647,9999999999
$FFFFFFFF $7FFFFFFF 2constant +inf
\ 2147483648,0
$0 $80000000 2constant -inf

\ Return the floor of an s31.32 value df
: floor ( df -- df ) nip 0 swap 2-foldable ;

\ Convert an s31.32 angle df1 in degrees to an
\ angle df2 in [0, 360) such that
\ df1 = df2 + n*360 where n is an integer
: deg0to360 ( df1 -- df2 )
  360,0 d/mod 2drop 2dup d0< if 360,0 d+ then
  2-foldable
;

\ Convert an s31.32 angle df1 in degrees to an
\ angle df2 in [-90, 90) such that
\ df1 = df2 + n*180 where n is an integer.
\ (For tan only.)
: deg-90to90 ( df1 -- df2 )
  180,0 d/mod 2drop
  2dup 90,0 d< not if
    180,0 d-
  else
    2dup -90,0 d< if
      180,0 d+
    then
  then
  2-foldable
;

\ From common directory of Mecrisp-Stellaris
\ Forth 2.4.0
: numbertable <builds does> swap 2 lshift + @ ;

\ ------------------------------------------------
\  Square root functions
\ ------------------------------------------------
: 0to1sqrt ( x -- sqrtx )
  \ Take square root of s31.32 number x
  \ with x in interval [0, 1]
  \ Special cases x = 0 and x = 1
  2dup d0= if exit then
  2dup 1,0 d= if exit then

  swap    \ Put x in MSW of unsigned 64-bit integer u

  \ Find square root of u as 64-bit unsigned int
  0,0 2swap 1,0 30 lshift  \ Stack: ( res  u  bit )
  \ Start value of bit is highest power of 4 <= u
  begin 2over 2over du< while dshr dshr repeat

  \ Do while bit not zero
  begin 2dup 0,0 d<> while
    2rot 2over 2over d+ 7 pick 7 pick
      du> not if  \ u >= res+bit ?
      2rot 2over d- 2rot 2tuck d- \ u = u - res - bit
      2swap 2rot dshr
      2over d+ \ res = (res >> 1) + bit
    else
      dshr    \ res = res >> 1
    then
    2-rot       \ Return stack to ( res u bit )
    dshr dshr   \ bit = bit >> 2
  repeat

  \ Drop u and bit, res is s31.32 square root of x
  2drop 2drop
  2-foldable
;

: sqrt ( x -- sqrtx )
  \ Find square root of non-negative s31.32 number x
  \ If x in interval [0, 1], use 0to1sqrt
  2dup 1,0 d> not if
    0to1sqrt
  else
    \ Divide x by 4 until result is in interval [0, 1]
    0,0 2swap   \ Init count of divides ndiv
                \ (use double for convenience)
    begin 2dup 1,0 d> while
      2swap 1,0 d+ 2swap    \ Incr count
      dshr dshr             \ Divide by 4
    repeat
    0to1sqrt
    2swap nip 0 do    \ ndiv consumed
      dshl                \ Multiply by 2 ndiv times
    loop
  then
  2-foldable
;

\ ------------------------------------------------
\  Helpers and constants for trig functions
\ ------------------------------------------------
: deg2rad ( deg -- rad )
  \ Convert s31.32 in degress to s31.32 in radians
  74961321 0 f*
  2-foldable
;

: rad2deg ( rad -- deg )
  \ Convert s31.32 in radians to s31.32 in degrees
  1270363336 57 f*
  2-foldable
;

\ pi/2 and pi/4 as s31.32 numbers
 \ (whole part first for retrieval with 2@)
2451551556 1 2constant pi/2
3373259426 0 2constant pi/4

\ s31.32 comma parts of coefficients in Horner
\ expression of 7-term series expansion of sine
\ after an x is factored out.  The whole parts are
\ 0 and are supplied in code.
numbertable sin-coef
   20452225 ,   \  1/(14*15)
   27531842 ,   \  1/(12*13)
   39045157 ,   \  1/(10*11)
   59652324 ,   \  1/(8*9)
  102261126 ,   \  1/(6*7)
  214748365 ,   \  1/(4*5)
  715827883 ,   \  1/(2*3)

: half-q1-sin-rad  ( x -- sinx )
  \ Sin(x) for x in first half of first quadrant Q1
  \ and its negative x is a s31.32 angle in radians
  \ between -pi/4 and pi/4
  2dup 2dup f*          \  x and x^2 on stack as dfs
  \ Calculate Horner terms
  -1,0   \ Starting Horner term is -1
  7 0 do
    \ Multiply last term by x^2 and coefficient,
     \ then add +1 or -1 to get new term
    2over f* i sin-coef 0 f* 0 1
    i 2 mod 0= if d+ else d- then
  loop
  \ Last term is multiplied by x
  2nip f*
  2-foldable
;

\ s31.32 comma parts of coefficients in Horner
\ expression of 8-term series expansion of cosine.
\ The whole parts are 0 and are supplied in code.
numbertable cos-coef
   17895697 ,   \  1/(15*16)
   23598721 ,   \  1/(13*14)
   32537631 ,   \  1/(11*12)
   47721859 ,   \  1/(9*10)
   76695845 ,   \  1/(7*8)
  143165577 ,   \  1/(5*6)
  357913941 ,   \  1/(3*4)
 2147483648 ,   \  1/2

: half-q1-cos-rad  ( x -- cosx )
  \ Cos(x) for x in first half of first quadrant Q1
  \ and its negative x is a s31.32 angle in radians
  \ between -pi/4 and pi/4
  2dup f*          \  x^2 on stack
  \ Calculate Horner terms
  1,0   \ Starting Horner term is 1
  8 0 do
```

```
    \ Multiply last term by x^2 and coefficient,
     \ then add +1 or -1 to get new term
    2over f* i cos-coef 0 f* 0 1
    i 2 mod 0= if d- else d+ then
  loop
  2nip
  2-foldable
;

: q1-sin-rad ( x -- sinx )
  \ Sin(x) for x in first quadrant Q1 and its negative
  \ x is a s31.32 angle in radians between -pi/2
  \ and pi/2
  2dup pi/4 d< if
    half-q1-sin-rad
  else
    pi/2 2swap d- half-q1-cos-rad
  then
  \ Apply max/min limits
  \ 2dup 1,0 d> if 2drop 1,0 exit then
  \ 2dup -1,0 d< if 2drop -1,0 exit then
  2-foldable
;

: q1toq4-sin ( x -- sinx )
  \ Sin(x) for x in quadrants Q1 through Q4
  \ x is a s31.32 angle in degrees between 0 and 360
  2dup 270,0 d> if
    360,0 d- deg2rad q1-sin-rad
  else 2dup 90,0 d> if
    180,0 d- deg2rad q1-sin-rad dnegate
  else
    deg2rad q1-sin-rad
  then then
  2-foldable
;

\ s31.32 comma parts of coefficients in Horner
\ expression of 6-term Euler expansion of atan
\ after x/(x^2+1) is factored out.
\ The whole parts are 0 and are supplied in code.
\ The series variable is y = (x^2)/(x^2+1).
numbertable atan-coef
  3964585196 ,    \   12/13
  3904515724 ,    \   10/11
  3817748708 ,    \   8/9
  3681400539 ,    \   6/7
  3435973837 ,    \   4/5
  2863311531 ,    \   2/3

: base-ivl-atan ( x -- atanx )
  \ Calc atan for s32.31 x in base interval 0 to 1/8.
  2dup 2dup f* 2dup 1,0 d+ \ Stack: ( x  x^2  x^2+1 )
  2rot 2swap f/            \ Stack: ( x^2  x/(x^2+1) )
  2swap
  2dup 1,0 d+ f/  \ Stack: ( x/(x^2+1) (x^2)/(x^2+1) )
  \ Calc Horner terms for powers of y = (x^2)/(x^2+1)
  1,0   \ Starting Horner term is 1
  6 0 do
    \ Multiply last term by y and coefficient,
     \ then add 1 to get new term
    2over f* i atan-coef 0 f* 1,0 d+
  loop
  \ Last term is multiplied by x/(x^2+1)
  2nip f*
  2-foldable
;

\ Table of atan(i/8), i = 0, 1, ..., 8,
\ values in radians.
\ Only comma parts given, all whole parts are 0.
numbertable atan-table
           0 ,
   534100635 ,
  1052175346 ,
  1540908296 ,
  1991351318 ,
  2399165791 ,
  2763816217 ,
  3087351340 ,
  3373259426 ,

: 0to1-atan ( x -- atanx )
  \ Calc atan for s31.32 x in interval [0, 1]
  2dup 1,0 d= if
    2drop
    8 atan-table 0
  else
    \ Find interval [i/8, (i+1)/8) containing x,
    \ then use formula
    \   atan(x) =
    \   atan(i/8) + atan((x - (i/8))/(1 + (x*i/8)))
    \ where the argument in the second term is
    \ in [0, 1/8].
    0 7 do
      0 i 8,0 f/ 2over 2over d< not if
        2over 2over d-
        2-rot f* 1,0 d+
        f/ base-ivl-atan
        i atan-table 0 d+
        leave
      else
        2drop
      then
    -1 +loop
  then
  2-foldable
;

\ ------------------------------------------------
\  Trig functions
\ ------------------------------------------------
: sin ( x -- sinx )
  \ x is any s31.32 angle in degrees
  2dup 2dup d0< if dabs then
  \ Stack is ( x |x| )
  360,0 ud/mod 2drop
  q1toq4-sin    \ sin|x|
  \ Negate if x is negative
  2swap d0< if dnegate then
  2-foldable
;

: cos ( x -- cosx )
  \ x is any s31.32 angle in degrees
  90,0 d+ sin
  2-foldable
;

: tan ( x -- tanx )
  \ x is any s31.32 angle in degrees
  \ Move x to equivalent value in [-90, 90)
  deg-90to90
  \ If |x| > 89,9 deg, use approximation
  \   sgn(x)(180/pi)/(90-|x|)
  2dup dabs 2dup 89,8 d> if
    90,0 2swap d- 608135817 3 f* 180,0 2swap f/
    2swap d0< if dnegate then
  else
    2drop 2dup sin 2swap cos f/
  then
  2-foldable
;

: atan ( x -- atanx )
  \ Calc atan for s31.32 x, return result in degrees
  2dup 2dup d0< if dabs then   \ Stack: ( x |x| )
  \ Find atan(|x|)
  2dup 1,0 d> if
    \ |x| > 1, use
    \   atan(|x|) =
    \   (pi/2) - atan(1/|x|)
    \ with 1/|x| in [0, 1]
    1,0 2swap f/ 0to1-atan pi/2 2swap d-
  else
    \ |x| <= 1
    0to1-atan
  then
  \ Negate if x is negative
  2swap d0< if dnegate then
  rad2deg
  2-foldable
;

: asin ( x -- asinx )
  \ Calc asin for s31.32 x in interval [-1, 1],
  \ return result in degrees
  2dup 2dup d0< if dabs then
  \ Stack is ( x |x| )
  2dup 1,0 d> if drop exit then   \ Exit if |x|>1
                                  \ with x on stack
  2dup 2dup f* 1,0
  2swap d- 0to1sqrt   \ Stack: ( x  |x|  sqrt(1-x^2) )
  2over 2dup f* 0,5 d> if           \ x^2 > (1/2) ?
    2swap f/ atan 90,0 2swap d-
  else
    f/ atan
  then
  \ Negate if x is negative
  2swap d0< if dnegate then
```

```
  2-foldable
;

: acos ( x -- acosx )
  \ Calc acos for s31.32 x in interval [-1, 1],
  \ return result in degrees
  90,0 2swap asin d-
  2-foldable
;

\ -----------------------------------------------
\  Helper for logarithmic functions
\ -----------------------------------------------
: log2-1to2 ( y -- log2y )
  \ Helper function that requires y is s31.32 value
  \ with 1 <= y < 2
  0 0 2swap 0
  ( retval y cum_m )
  \ while((cum_m < 33) && (y > 1))
  begin dup 2over
    ( retval y cum_m cum_m y )
    1,0 d> swap 33 < and while
    ( retval y cum_m )
    rot rot 0 -rot        \ m = 0, z = y
    ( retval cum_m m z)
    \ Do z = z*z, m = m+1 until 2 <= z.
    \ We also get z < 4
    begin
      2dup f* rot 1 + -rot
      ( retval cum_m m z )
      2dup 2,0 d< not
    until
    \ At this point z = y^(2^m) so that
    \ log2(y) = (2^(-m))*log2(z)
    \ = (2^(-m))*(1 + log2(z/2)) and 1 <= z/2 < 2
    \ We will add m to cum_m and add 2*(-cum_m)
    \ to the returned value,
    \ then iterate with a new y = z/2
    ( retval cum_m m z )
    2swap + -rot dshr 2>r \ cum_m = cum_m + m, y = z/2
    ( retval cum_m ) ( R: y=z/2 )
    \ retval = retval + 2^-cum_m
    1,0 2 pick 0 do dshr loop
    ( retval cum_m 2^-cum_m )
    rot >r d+
    ( retval ) ( R: y cum_m )
    r> 2r> rot
    ( retval y cum_m )
  repeat
  drop 2drop
  2-foldable
;

\ -----------------------------------------------
\  Logarithmic functions
\ -----------------------------------------------
: log2 ( x -- log2x )
  \ Calculates base 2 logarithm of
  \ positive s31.32 value x

  \ Treat error and special cases
  \ Check that x > 0.  If not, return "minus infinity"
  2dup 0,0 d> not if 2drop -inf exit then
  \ If x = 1, return 0
  2dup 1,0 d= if 2drop 0,0 exit then

  \ Find the n such that 1 <= (2^(-n))*x < 2
  \ This n is the integer part (characteristic)
  \ of log2(x)
  0 -rot
  ( n=0 y=x )
  2dup 1,0 d> if
    \ Do n = n+1, y = y/2 while (y >= 2)
    begin 2dup 2,0 d< not while
      ( n y )
      dshr rot 1 + -rot
    repeat
  else
    \ Do n = n-1, y = 2*y while (y < 1)
    begin 2dup 1,0 d< while
      ( n y )
      dshl rot 1 - -rot
    repeat
  then

  \ Now y = (2^(-n))*x so log2(x) = n + log2(y) and
  \ we use the helper function to get log2(y)
  \ since 1 <= y < 2
  log2-1to2 rot 0 swap d+
  ( log2x )
  2-foldable
;

1292913986 0 2constant log10of2

: log10 ( x -- log10x )
  \ Calculates base 10 logarithm of positive s31.32
  \ value x

  \ Treat error and special cases
  \ Check that x > 0.  If not, return "minus infinity"
  2dup 0,0 d> not if 2drop -inf exit then
  \ If x = 1, return 0
  2dup 1,0 d= if 2drop 0,0 exit then

  \ Find the n such that 1 <= (10^(-n))*x < 10
  \ This n is the integer part (characteristic)
  \ of log2(x)
  0 -rot
  ( n=0 y=x )
  2dup 1,0 d> if
    \ Do n = n+1, y = y/10 while (y >= 10)
    begin 2dup 10,0 d< not while
      ( n y )
      10,0 f/ rot 1 + -rot
    repeat
  else
    \ Do n = n-1, y = 10*y while (y < 1)
    begin 2dup 1,0 d<  while
      ( n y )
      10,0 f* rot 1 - -rot
    repeat
  then

  \ Now y = (10^(-n))*x so log10(x) = n + log10(y)
  \ and we use the identity
  \ log10(y) = log10(2)*log2(y)
  log2 log10of2 f* rot 0 swap d+
  ( log10x )
  2-foldable
;

2977044472 0 2constant lnof2

: ln ( x -- lnx )
  \ Return the natural logarithm of a postive s31.32
  \ value x

  \ Treat error and special cases
  \ Check that x > 0.  If not, return "minus infinity"
  2dup 0,0 d> not if 2drop -inf exit then
  \ If x = 1, return 0
  2dup 1,0 d= if 2drop 0,0 exit then

  log2 lnof2 f*
  2-foldable
;

\ -----------------------------------------------
\  Power functions
\ -----------------------------------------------
\ s31.32 comma parts of all but first coefficient
\ in Horner expansion of a partial sum of the series
\ expansion of exp(x).  The whole parts are 0
\ and are supplied in code.
numbertable exp-coef
   390451572 ,   \  1/11
   429496730 ,   \  1/10
   477218588 ,   \  1/9
   536870912 ,   \  1/8
   615366757 ,   \  1/7
   715827883 ,   \  1/6
   858993459 ,   \  1/5
  1073741824 ,   \  1/4
  1431655765 ,   \  1/3
  2147483648 ,   \  1/2

: exp-1to1 ( x -- expx )
  \ Calculate exp(x) for x an s31.32 value.
  \ Values are correct when rounded to six decimal
  \ places when x is between +/-0.7. Uses an 11-term
  \ partial sum evaluated using Horner's method.
  \ Calculate Horner terms
  1,0   \ Starting Horner term is 1
  10 0 do
    \ Multiply last term by x and coefficient,
    \ then add to get new term
    2over f* i exp-coef 0 f* 0 1 d+
  loop
  \ Last part of expansion
```

```
    2over f* 0 1 d+
    2nip
    2-foldable
  ;

  : pow2 ( x -- 2^x )
    \ Return 2 raised to the power x where x is s31.32
    \ If x is 0, return 1
    2dup 0,0 d= if 2drop 1,0 exit then
    \ If x < -32, 0 is returned.
    \ If x >= 31, returns s31.32 ceiling
    2dup -32,0 d< if 2drop 0,0 exit then
    2dup 31,0 d< not if 2drop +inf exit then
    \ Get largest integer n such that
    \ n <= x so x = z + n, 0 <= z < 1
    2dup floor 2swap 2over d-
    ( n z )
    \ Get exp(z*ln2) = 2^z,
    \ then shift n times to get 2^x = (2^n)*(2^z)
    lnof2 f* exp-1to1 2swap nip
    ( 2^z n )  \ n now a single
    dup 0= if
      drop
    else
      dup 0< if
        negate 0 do dshr loop
      else
        0 do dshl loop
      then
    then
    2-foldable
  ;

  1901360723 1 2constant 1overlnof2

  : exp ( x -- expx )
    \ Return the exponential e^x of the s31.32 value x
    \ If x is 0, return 1
    2dup 0,0 d= if 2drop 1,0 exit then
    \ Return s31.32 ceiling for large pos. exponents,
    \ 0 for large neg.
    2dup 21,5 d> if 2drop +inf exit then
    2dup -22,2 d< if 2drop 0,0 exit then
    \ If |x| < 0.36, use exponential series
    \ approximation
    \ Otherwise, use exp(x) = pow2(x/ln(2))
    2dup dabs 0,36 d< if
      exp-1to1
    else
      1overlnof2 f* pow2
    then
    2-foldable
  ;

  1382670639 3 2constant ln10overln2

  : pow10 ( x -- 10^x )
    \ Return 10 raised to the power x where x is s31.32
    \ If x is 0, return 1
    2dup 0,0 d= if 2drop 1,0 exit then
    \ Return s31.32 ceiling for large pos. exponents,
    \ 0 for large neg.
    2dup 9,35 d> if 2drop +inf exit then
    2dup -9,64 d< if 2drop 0,0 exit then
    \ If x is a positive integer generate powers of 10
    \ with multiplications
    \ Otherwise use 10^x = 2^(x*ln(10)/ln(2))
    2dup 2dup floor d= if
      2dup 0,0 d> if
        1,0 2swap nip
        0 do 10,0 f* loop
      else
        ln10overln2 f* pow2
      then
    else
      ln10overln2 f* pow2
    then
    2-foldable
  ;

  \ ------------------------------------------------
```

Bare Metal Forth — EULEX

Carsten Strotmann

Bare Metal Forth[1] *— Forth ohne Betriebssystem direkt auf der Hardware — was auf Mikrokontrollern normal ist, ist in der Welt der PCs eine Seltenheit. Klar, vor 40 Jahren zu Zeiten von MS–DOS und CP/M hat es solche Forth–Systeme auch schon gegeben; diese sind dann in Zeiten von Windows, macOS und Linux aber von der Bildfläche verschwunden.*

Eulex *ist ein modernes „Bare–Metal" Forth für 32–Bit–PCs mit Intel–kompatibler CPU, vom i386 bis zum modernen AMD Ryzen. Erschaffen wurde Eulex von* David Vázquez. *Wie er in seinem Blog[1] erzählt, hatten er und ein Kollege die Idee, ein Hobby–Betriebssystem in der Sprache „C" zu entwickeln. Das Betriebssystem–Projekt ist ihrem Zeitmangel zum Opfer gefallen, aber das für dieses System geschaffene Forth hat überlebt und kann nun im Quellcode als freie Software (GPLv3 Lizenz) von Github[2] bezogen werden.*

Was macht Eulex besonders?

Eulex besteht aus einem kleinen Kern aus Maschinensprache–Primitiven (ca. 36 KB), 1635 Zeilen Assembler–Code in den Dateien `boot.S` und `forth.S` und ca. 6000 Zeilen Forth–Quellcode. Dabei ist ein großer Teil des Forth–Systems in Forth definiert und wird nach dem Starten des Forth–Kerns aus den Quellen kompiliert. Dazu bringt das Eulex–Programm den Forth–Quellcode als Text–Dateien mit, diese werden in den Hauptspeicher geladen und von dort interpretiert und kompiliert.

Aus historischen Gründen — das Projekt sollte ja ein in „C" geschriebenes Betriebssystem werden — wird Eulex noch mit dem GNU–C–Compiler übersetzt, auch wenn Eulex heute keinen C–Code mehr beinhaltet. Das geht am besten auf einem Linux–System mit dem GNU–C–Compiler, GNU–make und dem GNU–Linker *ld*. Nach dem Kompilieren bekommt der Benutzer eine ca. 230 KB große Programmdatei, welche dann direkt per Bootloader auf einem modernen PC gestartet werden kann. Die Programmdatei entspricht dabei dem ELF–Dateistandard[3] mit der Multiboot[4]–Erweiterung. Als Bootloader bieten sich *Syslinux* oder *Grub2* an.

Nach dem Start findet der Benutzer ein minimales Forth–Vokabular vor, welches den Grundwortschatz eines Forth–Systems bietet. Ähnlich wie das *minimal Forth*[5] Projekt von Ulli Hoffmann vermeidet dieses Vokabular, dass der Forth–Anfänger durch eine Wortliste mit hunderten von Einträgen abgeschreckt wird. Nach der Eingabe des Wortes `eulex` wird aber das gesamte Vokabular dieses Forth–Systems freigeschaltet und sichtbar.

Als Benutzer des Emacs–Editors hat David Vázquez auch dem Eulex–Forth einige bekannte Tastenkombinationen aus Emacs mitgegeben. So springt die Tastenkombination `CTRL+E` an das Ende der Eingabezeile, ein `CTRL+A` an den Anfang. Der mitgelieferte Block–Editor wird mit `CTRL+X+C` verlassen. Für Anhänger der Emacs–Kirche[6] ein Pluspunkt. Forth–Worte lassen sich per Tab–Taste vervollständigen, so wie man es von modernen Unix–Shells — zsh, mksh, bash etc. — gewohnt ist.

[1] engl. für blankes Metall; synonym für „nichts mehr dazwischen"

Kommunikation mit der Welt

Eulex ist „bare metal". Es gibt unter dem Forth kein Betriebssystem. Kein Linux, kein Windows, nicht einmal ein MS-DOS. Und auch kein BIOS, da wir uns im 32–Bit–Modus der CPU befinden. Jegliche Kommunikation mit der Außenwelt muss direkt vom Forth aus über die Hardware geschehen.

Ein Treiber für die Textausgabe im Zeichenmodus der Grafikkarte ist eingebaut, wie auch Worte, um von der Tastatur zu lesen. Die erste serielle Schnittstelle `COM1` kann angesprochen werden. Aber es muss eine *echte* serielle Schnittstelle sein, USB–zu–Seriell–Wandler funktionieren mangels USB–Treiber nicht. Datenspeicherung mittels Forth–Blöcken im Hauptspeicher, eine Art „Ramdisk", wird mittels `use-memory` bereitgestellt; Forth–Blöcke auf dem ersten Diskettenlaufwerk per `use-floppy`. Der Floppy–Kontroller wird direkt programmiert. Auch hier muss es ein echtes Hardware–Diskettenlaufwerk sein, BIOS–Emulation oder USB–Floppy können nicht funktionieren.

Anwendungen und Anpassungen

Eulex–Forth kommt mit zwei Beispiel–Anwendungen: Das Spiel `Sokoban`, welches von GNU/Forth (gforth) portiert wurde, und ein einfacher in Forth geschriebener Interpreter für die Sprache LISP.

Der gesamte Forth–Quellcode des Eulex–Systems wird beim Kompilieren in die Eulex–Programmdatei gepackt und beim Laden des Eulex–Systems in den Speicher geladen. Die einzelnen Forth–Dateien lassen sich mit einem Forth–Wort, bestehend aus dem Zeichen `@`, dem Pfadnamen und dem Dateinamen laden. Sokoban wird durch Ausführen des Wortes `@app/sokoban.fs` geladen, das LISP mittels `@lisp/lisp.fs`.

Eigener Forth–Quellcode kann in die Datei `eulexrc.fs` eingetragen werden. Der Inhalt dieser Datei wird am Ende des Startvorgangs geladen und interpretiert. Alternativ kann eine neue Forth–Datei angelegt werden. Damit auch diese Datei Bestandteil des Forth–Systems wird, muss der Dateiname in der Make–Datei `GNUmakefile` in der

Variable `FORTH_SRC` eingetragen werden. Für alle in dieser Variablen aufgeführten Dateien wird das Shell–Skript `generate-builtin-files.sh` aufgerufen. Dieses Skript erstellt einen kleinen Header in Assembler für jede Datei, damit diese per eigenem Forth–Wort in Eulex angesprochen und geladen werden kann. Nach jeder Änderung der Datei `eulexrc.fs` oder der Make–Datei muss Eulex mit dem Befehl `make` neu aus den Quellen übersetzt werden.

Nachfolgend ist die Ladereihenfolge der Forth–Quelldateien beim Systemstart von Eulex dokumentiert. Eine Einrückung bedeutet dabei, dass die Datei aus dem Quellcode der oberen Ebene als Abhängigkeit geladen wird:

```
@core.fs
@structures.fs
@exceptions.fs
@interpreter.fs
@math.fs
@string.fs
@vocabulary.fs
@kernel/multiboot.fs
@memory.fs
@kernel/console.fs
@colors.fs
@output.fs
@corestage2.fs
@tools.fs
@disassem.fs
@kernel/interrupts.fs
@kernel/exceptions.fs
@debugger.fs
@kernel/irq.fs
@kernel/timer.fs
@kernel/floppy.fs
@kernel/keyboard.fs
@kernel/serial.fs
@kernel/speaker.fs
@tests/tests.fs
@kernel/cpuid.fs
@blocks.fs
@input.fs
@linedit.fs
@user.fs
@editor.fs
@eulexrc.fs
```

Zur Laufzeit kann Eulex Forth–Quelltext aus Forth–Blöcken von Diskette laden oder per serieller Schnittstelle empfangen.

Tritt ein Fehler auf, z. B. im Forth–Compiler, weil ein Wort nicht gefunden wird, so gibt Eulex den Fehler zusammen mit dem Aufrufstapel der Forth-Wörter aus:

```
bla
ERROR: Unknown word
>>>bla<<<
Backtrace:
throw
execute
evaluate
evaluate
execute
catch
evaluate
```

Bei der Programmentwicklung hilft der Debugger `see` :

```
: square dup * ; ok

see square
$001579D3: CALL $001003D2 <dup>
$001579D8: CALL $0010025E <*>
$001579DD: RET
ok

see cpuflags
$0014F7B6: JMP $0014F7BF
$0014F7BB: [unkown opcode 102 'f']
$0014F7BC: [unkown opcode 112 'p']
$0014F7BD: [unkown opcode 117 'u']
$0014F7BE: [unkown opcode 32 ' ']
$0014F7BF: SUBL $4, %ESI
$0014F7C2: MOVL $1374139, (%ESI)
$0014F7C8: SUBL $4, %ESI
$0014F7CB: MOVL $4, (%ESI)
$0014F7D1: CALL $00142231 <type>
$0014F7D6: JMP $0014F7DF
$0014F7DB: [unkown opcode 112 'p']
$0014F7DC: [unkown opcode 115 's']
$0014F7DD: [unkown opcode 101 'e']
$0014F7DE: [unkown opcode 32 ' ']
$0014F7DF: SUBL $4, %ESI
$0014F7E2: MOVL $1374171, (%ESI)
$0014F7E8: SUBL $4, %ESI
$0014F7EB: MOVL $4, (%ESI)
$0014F7F1: CALL $00142231 <type>
$0014F7F6: RET
```

Entwicklung von und mit Eulex

Zur Programmentwicklung von Forth–Programmen unter Eulex empfiehlt sich ein Virtual–Machine–Hypervisor wie QEMU/KVM[7], VirtualBox oder ein PC–Emulator wie Bochs[8]. Mit einer solchen Umgebung lässt sich Eulex nach Anpassungen schnell starten und der neue Programmcode testen. Im Eulex–Quellcode–Repository befindet sich das Skript `run-eulex.sh`, ein Beispiel, wie Eulex innerhalb des QEMU/KVM–Hypervisors gestartet werden kann.

Bootloader

Eulex kann über Bootloader für 32–Bit–Betriebssysteme gestartet werden, die ELF oder Multi–Boot–Programme starten können, z. B. Syslinux[9] oder Grub2[10]. Der Syslinux–Bootloader passt auf eine 1.44–MB–Diskette und ist ideal für ältere Systeme (486, 586, 686) mit BIOS. Damit Eulex von Syslinux startbar ist, muss das Programm mit dem GNU–Linker *ld* gelinkt werden, nicht mit dem GCC–C–Compiler. Eine abgeänderte Makedatei und ein Linker–Skript finden sich im Github–Repository

des Autors[11]. Im Syslinux wird Eulex mit dem Kommando `elf eulex` gestartet.

Der Grub2–Bootloader ist ein moderner Bootloader für Linux und andere Betriebssysteme. Eulex–Forth entspricht der Multiboot–Spezifikation 2.0 und kann von Grub2 gestartet werden. Grub2 passt auf keine Diskette, kann aber Eulex von USB–Speichermedien, SD–Karten, CD/DVD–ROM oder sogar aus dem Netzwerk starten. Grub2 ist die ideale Lösung für moderne Systeme mit BIOS oder EFI–Firmware.

```
menuentry "Eulex-Forth" {
multiboot (hd1,msdos0)/eulex
}
```

Eulex–Forth zu entdecken und mit Eulex–Forth zu arbeiten macht Spaß. Das System ist übersichtlich. Und obwohl wenig echte Dokumentation existiert, ist es verständlich, weil der Quellcode sauber geschrieben und gut kommentiert wurde.

Und zum Schluss noch die Beweisfotos ...

Hab's ausprobiert, die beiden Sreenshots sollen zeigen, dass es wirklich geht.

Abbildung 1: Die Startmeldung im QEMU-Fenster

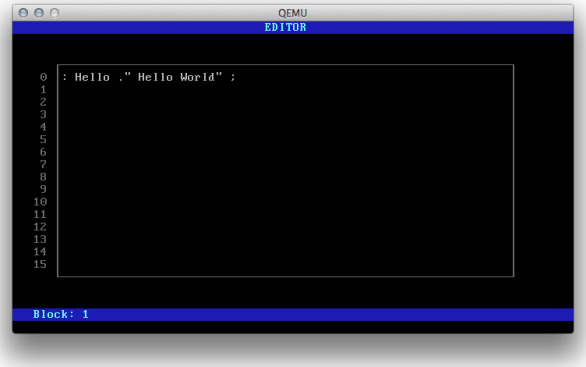


Abbildung 2: Look 'n feel des Eulex Screen-Editors

Quellenangaben und Links

[1] "I wrote a Forth implementation for x86" — `https://davazp.net/2012/12/08/eulex-forth-implementation.html`

[2] Eulex Github Quellcode — `https://github.com/davazp/eulex`

[3] ELF — Executable and Linking Format — `https://de.wikipedia.org/wiki/Executable_and_Linking_Format`

[4] Multiboot Spezifikation — `https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html`

[5] Minimal Forth — `https://github.com/uho/minimal`

[6] Church of Emacs — `https://www.emacswiki.org/emacs/ChurchOfEmacs`

[7] QEMU — `https://www.qemu.org/`

[8] Bochs — `http://bochs.sourceforge.net/`

[9] Syslinux — `https://wiki.syslinux.org/`

[10] Grub2 — `https://www.gnu.org/software/grub/`

[11] Eulex für Syslinux Anpassungen — `https://github.com/cstrotm/eulex`

Arithmetik[2] in Forth - wo finde ich, was Forth da hat?

Der Forth–Standard enthält auch die grundlegenden Forth–Worte für arithmetische Operationen. Sie sind im *core word set* und weiteren word sets enthalten.

`https://forth-standard.org/standard/words`

Doch die Einteilung in word sets ist an der Implementierung der Sprache orientiert, nicht an deren Verwendung.

Wie man Forth verwendet, muss man sich aus Lehrbüchern oder Tutorials holen. Ein solches Tutorial ist im *gforth–Manual* enthalten, eine umfangreiche und mustergültige Forth–Implementation unter der *GNU Free Documentation License*.

`https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/`

Das gforth–Tutorial ist nach Funktionsgruppen gegliedert, also genau das, was man haben will, wenn man sich einarbeitet in die Sprache.

Es enthält ein sehr kompaktes Arithmetics–Tutorial von nur einer Seite Text mit Code–Beispielen.

`https://www.complang.tuwien.ac.at/forth/gforth/Docs-html/Arithmetics-Tutorial.html#`

Damit kann man sich hochhangeln zum gehobenen Gebrauch all der Forth–Worte, die für arithmetische Operationen da sind:

> 5.5 Arithmetic
>
> Forth arithmetic is not checked, i. e., you will not hear about integer overflow on addition or multiplication, you may hear about division by zero if you are lucky. The operator is written after the operands, but the operands are still in the original order. I. e., the infix 2 − 1 corresponds to 2 1 -. Forth offers a variety of division operators. If you perform division with potentially negative operands, you do not want to use / or /mod with its undefined behaviour, but rather fm/mod or sm/mod (probably the former, see Mixed precision).

- Single precision
- Double precision: Double-cell integer arithmetic
- Bitwise operations
- Numeric comparison
- Mixed precision: Operations with single and double-cell integers
- Floating Point

Folgt man im Tutorial den Links jener Auflistung, bekommt man alle arithmetischen Worte des Forth angezeigt und deren Funktion erklärt. Vom einfachen `+ - * /` ... bis hin zu `m+ */ */mod` ... `um/mod` und schließlich den floating point Worten `f+ f- f* f/` ... `fsin fcos` und höhere sowie deren Vergleichsoperatoren `f~ f= f<>` und ähnlichen.

Viel Erfolg beim Studium. mk

Versand der *Vierten Dimension*

Wir wünschen euch allen ein gutes Neues Jahr — bleibt gesund!

Der Versand geht so: Nach Fertigstellung der VD erfolgt der Upload der Datei und die Druck–Beauftragung online beim Reprozentrum Marquardt GmbH, Lauteschlägerstr. 6, 64289 Darmstadt. Derzeit haben wir eine Auflage von 115 Heften, mittig geklammert, außen Farbdruck und innen Schwarz–Weiß–Druck, mit Randbeschnitt.

In der Regel kommt die gedruckte VD nach einem Tag per DHL–Express zu uns. Zwischenzeitlich werden die Briefmarken für den Versand in Deutschland gekauft, Postboxen geholt und Adressetiketten gedruckt. Für 110 VDs brauche ich ca. 2 bis 3 Stunden zum Einpacken, Zukleben und Frankieren. Anschließend werden die beiden Boxen zur Post gefahren. Dort werden auch die Hefte für den Auslandsversand (~14 Hefte) frankiert und bezahlt. Dann geht die VD auf die Reise.

[Und damit ihr mal sehen könnt, wie es dabei so zugeht bei Ewald, hat er ein kleines Zeitraffer–Video davon gemacht.[3] mk]

Andrea & Ewald Rieger

JeeNode Zero — Mecrisp Forth and radio module drivers pre-installed

Im Internet aufgestöbert bei den Digitalsmarties[4]:

> Go Wireless with the new JeeNode Zero — an ARM M0+ based board with fully integrated radio opens up great new projects. These compact units have Mecrisp Forth and radio module drivers pre-installed — just connect up and right away you get a prompt for typing commands; no compiler tool chain to install and maintain! ...

Jean–Claude Wippler hat sich leider komplett aus Forth zurückgezogen. Die Platinen gibt es zwar noch, aber es ist ein aufgegebenes Projekt und wird nicht mehr weiterentwickelt. Im Endeffekt ist es also eine Mikrocontrollerplatine mit Radiomodul, wofür die meiste Software noch selbst geschrieben werden muss. Wer hat's ausprobiert?

Erich Wälde

[2] Definition der Arithmetik — der Zweig der Mathematik, der sich mit den Eigenschaften und der Manipulation von Zahlen befasst.

[3] `https://wiki.forth-ev.de/doku.php/vd-archiv#versand_der_vierten_dimension`

[4] `https://www.digitalsmarties.net/products/jeenode-zero`

Forth–Gruppen regional

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

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

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

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

Gruppengründungen, Kontakte

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

μP–Controller Verleih

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

Spezielle Fachgebiete

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

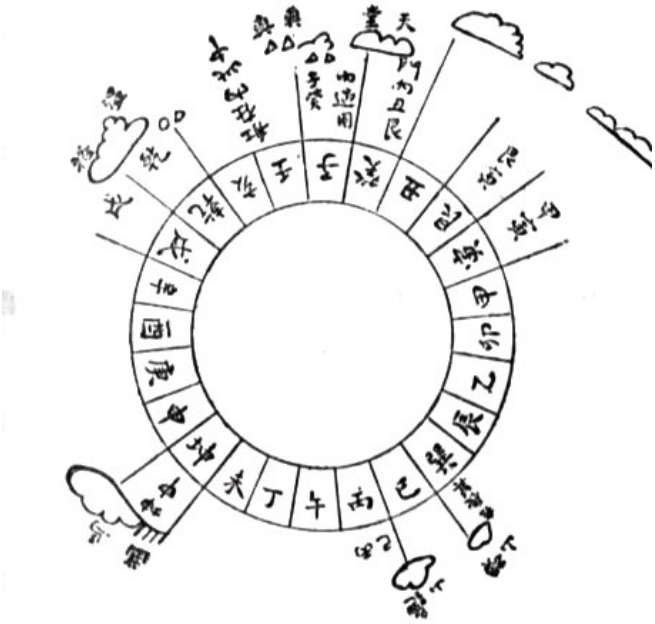
Termine

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

11.–14.4.2019
Forth–Tagung in Worms
`https://tagung.forth-ev.de`

27.–28.4.2019
VCFe in München
`https://vcfe.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.

Forth–Tagung in Worms

Ewald und Andrea Rieger

Die Tagung findet vom 11. bis 14. April 2019 im Hotel–Weingut Sandwiese *in der Nibelungen-, Dom-, Luther- und Weinstadt Worms statt. Das Hotel Sandwiese — Fahrweg 19, 67550 Worms–Herrnsheim — liegt am Rande von Worms–Herrnsheim. In unmittelbarer Nähe befindet sich das Herrnsheimer Schloss mit seinem Park im Stil eines englischen Landschaftsgartens.*

Anreise

Worms ist gut über die A 61, Abfahrt Worms–Mitte erreichbar. Bahnreisende erreichen den HBF Worms über Mainz oder Mannheim kommend und steigen dort in den Bus nach Worms–Herrnsheim um.

Anmeldung

Unter dem Link `https://tagung.forth-ev.de/` könnt ihr euch anmelden.