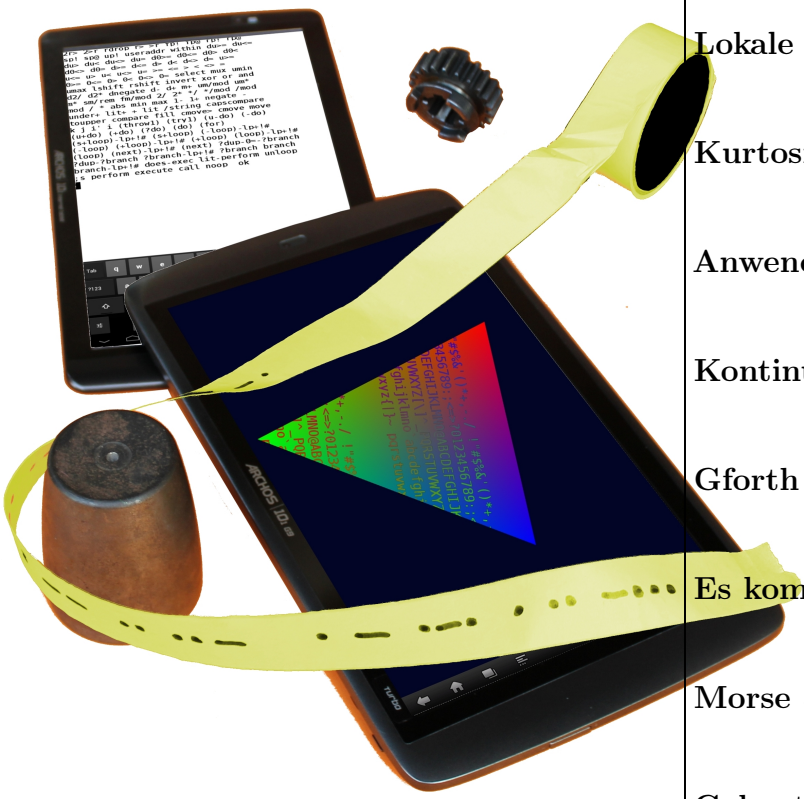
Vierte Dimension

3/2012

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

tematik GmbH
Technische Informatik

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

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

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

LEGO RCX–Verleih

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

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

Inhalt:

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

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

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

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

KIMA Echtzeitsysteme GmbH

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

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

FORTecH Software GmbH
Entwicklungsbüro Dr.–Ing. Egmont Woitzel

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

Ingenieurbüro
Klaus Kohl–Schöpe

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

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

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,

Bernd Paysan schrieb in der vorigen *Vierten Dimension*: „so übergebe ich dann den Stab an Martin Bitter“. Er erwähnte ebenfalls, dass die nächste *Vierte Dimension* (also die, die Sie nun in der Hand halten) mit Fossil als Versionsverwaltung erstellt würde. Gereiftere Leser mögen sich erinnern, dass ich bereits vor Jahren einige Ausgaben der *Vierten Dimension* gestalten durfte. Damals war dies eine zeitraubende und fehlerträchtige Aufgabe. Als Redaktionslaie arbeitete ich mit OpenOffice, was einige technische Probleme aufwarf. Unter anderem gelang das WYSIWYG selten perfekt. Oftmals sah die fertig gedruckte und verteilte *Vierte Dimension* nicht so aus wie auf den Druckfahnen bei mir zu Hause. Manchmal wurden Sonderzeichen vertauscht, was zu grässlichen Entstellungen und echten Falschdarstellungen führte.

Nun – mit der Versionsverwaltung und der Entscheidung, die *Vierte Dimension* als PDF–Datei zu gestalten, gehören solche Fehlerquellen der Vergangenheit an. Bernd Paysan hat ein makefile geschrieben, das aus den vorliegenden Texten mittels pdflatex und TEX eine gefällige PDF-Datei erzeugt. Warum berichte ich das? – Nun, um Ihnen Lust zu machen, aktiv bei der Gestaltung einer *Vierten Dimension* mitzumachen!

Fossil installiert sich leicht. Um die eingereichten Artikel zu bearbeiten, muss man sich nicht mit TEX oder Verwandtschaft auskennen. Dieses Editorial entsteht z.B. mit einem einfachen Texteditor. Die Arbeit des Editors ist einfacher geworden: Autoren reichen Texte ein, gewiefte Autoren legen diese als TEX-Datei vor und haben einen sehr dezidierten Einfluss auf die Gestaltung, andere überlassen dies getrost dem Redaktionsteam: Fred Behringer liest Korrektur, der Editor pflegt diese ein, Bernd Paysan kümmert sich um die vielen technischen Dinge (Satz, Layout). Ihm vor allen muss ich danken, denn er hat mir den Umgang mit den Programmen *Fossil, TEX, Lyx, pdflatex* sehr leicht gemacht. Wenn's einmal läuft, benötigt man nur einen Texteditor s.o.

So ist diese Ausgabe der *Vierten Dimension* (mal wieder) ein gelungenes Beispiel für intensive Teamarbeit.

Nicht nur das. Sie bietet ein vielfältiges Angebot. Bernd Paysan und Matthias Trute schreiben über systemspezifische Debuggingtechniken und Fred Behringer stellt einen Ansatz zur Implementierung lokaler Variablen in High-Level-Forth vor. Maschinenbauer dürfte die dreiteilige Reihe zur Kurtosis und ihrer Umsetzung bei der Analyse von Wälzlagerdaten interessieren. Mit einem 'sprachphilosophischen' Beitrag meldet sich Steven Pelc zu Wort. Ein Thema, mit dem sich ebenfalls Erich Wälde auseinandersetzt, während uns Hannes Teich durch die Buchhaltung von Geburtstagsdaten im Laufe der Jahrtausende führt.

Ein solches Heft wie die *Vierte Dimension* ist immer das Ergebnis der Arbeit von Enthusiasten, die bereitwillig für andere etwas aufschreiben. Ohne deren Beiträge nützten alle schönen Text–, Satz– und Versionsverwaltungsprogramme nichts. Was zählt, ist der Inhalt – neudeutsch Content. Überprüfen Sie doch einmal, lieber Leser, ob Ihnen in Ihrer Werkstatt, in Ihrem Büro oder bei Ihrem Hobby nicht Inhalte begegnen, die interessant und mitteilungswürdig sind. Zögern Sie nicht, sich an die Redaktion zu wenden – Ihnen wird geholfen werden!

Martin Bitter

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-2012-03

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

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

Grundlegende Experimente mit einer MCU

`http://www.forth-ev.de/wiki/doku.php/projects:4e4th:4e4th:start:msp430g2553_experimente`

Die dort angegebenen Experimente wurden mit der MCU MSP430G2553 von Texas Instruments auf deren LaunchPad gemacht. Die Experimente können aber ebenso gut mit anderen MCUs angestellt werden. Es wird aber für nichts gehaftet. Kleine Forth–Programme (4e4th) für die Experimente mit der MCU sind dort angegeben.

Für diese Experimente brauchte ich ein TI–LaunchPad, ein Multimeter, ein Oszilloskop und einige Kleinteile, wie Widerstände und Kondensatoren, und einen kleinen Lautsprecher. Ein Steckbrett kann nicht schaden, ebenso ein Bündel Prüfklipp-Kabel, die einfach an das LaunchPad angeklemmt werden können.

Inzwischen sind schon folgende Experimente beschrieben:

- Portpin über Widerstand an Masse legen
- Strom durch P1.4 mittels bekannter Widerstände abschätzen
- Kurzschluss
- Strom von P1.4 nach P1.5 mittels bekannter Widerstände abschätzen
- Wie verhält sich ein Portpin an einem großen Kondensator?
- Wie groß ist der Widerstand, über den geladen und entladen wird?
- LED anschließen
- Lautsprecher anschließen
- Pegelveränderung und Spannungsspitzen
- Schwingung der Membran
- Resonanz der Membran
- Pegel an Portpin P1.4 und P1.5
- PWM an Portpin P2.2
- Einfache Forth-Programme für die Experimente

Euer mk

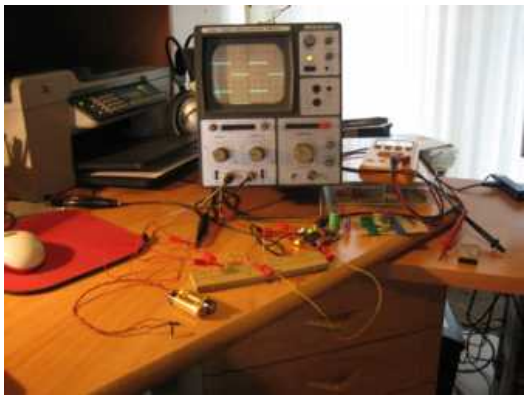


Abbildung 1: So sieht's bei mir dann aus...

Die Vuolasaho–Aufgabe: String Tutorial

Neulich fragte Hannu Vuolasaho [1] auf der amforth developer list [2] nach einem Beispiel dafür, wie man mit strings umgeht, und regte ein simples *sample string tutorial* an. Das sollte machen:

```
Who are you? user
How old are you? 25
Nice to meet you 25 year old user.
```

Nun, dachte ich, das kann so schwer nicht sein, nahm das gforth zur Hand und ruck zuck kam dabei heraus:

```
create namestring  20 allot
: getname  ( -- n )
   cr ." Who are you? "
   namestring 20 accept  ;
: getage   ( -- n )
   cr ." How old are you? "
   here 3 accept  here swap evaluate ;
: .user    ( n -- )
   namestring swap type ;
: .age     ( n -- )
   here swap type ;
: .hallo   ( -- )
   cr ." Nice to meet you "
   . ." year old " .user $2E emit cr ;
: ask      ( -- )
   getname getage .hallo ;
```

Dann hat Ulli sich das kritisch angesehen, und meinte: „... nicht schlecht. Intensiv faktorisiert." Doch nicht gefallen hatte ihm, dass

- der Wert 20 darin 2x auftaucht.
- `CR ."  ADR LEN ACCEPT` auch mehrfach da ist, also auch faktorisiert werden könnte.
- `EVALUATE` Ärger bereiten kann, sodass man besser `>NUMBER` benutzt.
- die Längen von Name und Alter auf dem Stack liegen, die Adresse aber nicht.

Zusammen genommen, ergab das dann:

```
20 chars Constant namestring-len
Create namestring  namestring-len allot
: input ( buf-addr buf-len prompt-addr
          promt-len -- buf-add len )
    cr type over swap  accept ;
: get-name ( -- addr len )
    namestring namestring-len
    s" Who are you? "  input ;
: get-age ( -- n )
    0.   here 3 chars
    s" How old are you? " input
    >number 2drop drop ;
: .hallo ( user user-len age -- )
    cr ." Nice to meet you "
    . ." year old " type ." "  cr ;
: ask ( -- )  get-name get-age .hallo ;
```

Schon tickiger. Vor allem `INPUT` ist interessant. Es macht stark Gebrauch davon, dass strings gerne als Startadresse und Länge auf dem Stack angegeben werden. Eine Philosophie, die zu einer tieferen Faktorisierung solcher Codestückchen führt, wie man sieht. `INPUT` hat das Zeug zu einem Klassiker.

Zu `>NUMBER` sollte man wissen, dass es eine doppeltgenaue Null auf dem Stack erwartet, in die hinein der Zahlenstring in eine Zahl konvertiert wird. Die von `>NUMBER` nach der Konvertierung noch hinterlassenen Informationen für den Compiler, ob und bis wo die Zeichenkonvertierung ging, kann man hier einfach mal wegwerfen. Aber um damit umgehen zu können, muss man schon tiefer in sein Forthsystem eingedrungen sein.

Im 4e4th für das TI–LaunchPad funktioniert meine einfache Lösung, wenn man beachtet, dass es kein Präfix vor einer Zahl verarbeitet. Die $2E würde dann eben als 46 geschrieben, wollte man dezimal bleiben. Für Ullis Lösung ist noch zu bedenken, dass das 4e4th ja im flash der MCU ist. So dass IS“ genommen werden muss, um den Abfragetext zu kompilieren. Und es muss auch beachtet werden, dass so ein kleines System nur ganze Zahlen nimmt. Damit gings dann:

```
..
: get-name ( -- addr len )
    namestring namestring-len
    is" Who are you? "  input ;
: get-age ( -- n )
    0 0  here 3 chars
    is" How old are you? " input
    >number 2drop drop ;
..
```

Dabei lernt man dann auch gleich noch, dass `CHARS` im 4e4th einfach ein `NOOP` ist.

Und Matthias meinte, was das amforth und AVR angeht, schon dazu: „Inhaltlich wären zwei Dinge anzumerken: `create .. allot` geht so nicht (siehe Rezeptsammlung), forth200x–konform ist der Einsatz von `buffer:` Außerdem ist `." ."` nicht dasselbe wie `$2e emit`, sieht nur im Ergebnis genauso aus. Hübsch wäre auch `[char] . emit` aber das ist eine andere Baustelle...“

So, wie es aussieht, stünde es jedem Tutorial zu einem bestimmten Forthsystem gut, die *Vuolasaho–Aufgabe* einmal exemplarisch zu lösen. mk

[1] Quelle:
Von: vuokko@msn.com
Betreff: [Amforth] Future documentation ideas
Datum: 23. August 2012 05:06:20 MESZ
An: amforth-devel@lists.sourceforge.net
Antwort an: amforth-devel@lists.sourceforge.net

[2] `https://lists.sourceforge.net/lists/listinfo/amforth-devel`

Neuigkeiten bei amforth 4.9

Die im Sommer veröffentlichte Version 4.9 des kleinen Forth für kleine Controller ist eine typische Viele-Kleinigkeiten-Version. Es gibt keine revolutionären Änderungen (wie die Recognizer letztes Jahr). Dafür aber viele, die kleine Ärgernisse beseitigen und auch den einen oder anderen Fehler beheben.

Die wohl wichtigste Neuerung ist eine “richtige” Kommandoshell, die Keith Amidon geschrieben hat. Sie hat so ziemlich alle Features, die man sich so (als Kommandozeilennutzer) wünscht: Zeileneditor, editierbare Historie, Command Completion mit TAB usw. Für amforth ist noch wichtig, dass es eine elegante Möglichkeit gibt, die vielen controllerspezifischen Registernamen und auch einzelne Bits verfügbar zu haben. Die bisherige Methode, mittels `$dead constant registername` viele Definitionen zu haben, die eigentlich nur das Dictionary füllen, übernimmt jetzt die neue Shell. Sie analysiert den Quelltext und ersetzt solche Namen selbst durch den richtigen Zahlenwert. Damit sie das auch richtig macht, kommuniziert sie beim Start mit dem Controller und lädt die passende Definitionsdatei automatisch. Beim Prompt sieht man so auch gleich, mit welchem Controller man verbunden ist.

```
mt@ayla:~/amforth$ ./tools/amforth-shell
                              -p /dev/ttyUSB0
|I=Entering amforth interactive interpreter
|I=getting MCU name..
|I=successfully loaded register definitions
                              for atmega32
|I=getting filenames on the host
|I=  Reading ./lib
|I=  Reading ./misc
(ATmega32)> (TAB 2x gedrückt)
Display all 476 possibilities? (y or n)
                     SPCR_SPE             d2*
!                    SPCR_SPIE            d2/
!e                   SPCR_SPR             d<
!e[]                 SPDR
....
(ATmega32)>
```

Matthias Trute

Debugger für Amforth

Matthias Trute

Braucht man Debugger? Braucht man nicht. Brian Kernighams Aussage „The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.“ [Unix for Beginners, 1979] hat bekanntlich seine Forth–Entsprechung im Faktorisieren und strategisch platzierten `.S` *im Code.*

Warum doch?

Amforth hat viele Jahre mit dieser Philosophie gelebt und ist damit zu dem geworden, was es jetzt ist. Allerdings sind die Anwenderwünsche nach Debuggern nie verstummt. Es ist mitnichten trivial, Wünsche, die man selbst nicht hat, trotzdem zu erfüllen.

So haben Zeit und Zufall geholfen, einige Werkzeuge zu schaffen. Sie sind einfach, erfüllen ihren Zweck jedoch präzise. Hinzukommt, dass mit ihrer Verfügbarkeit der Wunsch, sie einzusetzen, doch merklich zugenommen hat...

Was gibt es?

Amforth hat nicht „den“ Debugger. Es gibt vielmehr eine Reihe von Tools, die nebeneinander existieren und sich ergänzen können. Die Werkzeuge greifen mitunter in Interna ein oder setzen solche voraus, so dass sie nicht unmittelbar portabel auf andere Systeme sein dürften.

Die Tools sind nur für Colon-Worte vorgesehen. Ebenso entfalten sie ihre Wirkung nur für Worte, die nachdem das jeweilige Tool geladen wurde, definiert werden. Worte, die bereits existieren, bleiben außen vor.

Profiler

Es gibt im Wesentlichen zwei Aufgaben für Profiler: Die Laufzeit messen und die Anzahl der Aufrufe bestimmen. Für die Laufzeit kann man auf das Benchmarkumfeld zurückgreifen: `benchme`. Es ruft ein Wort über dessen Execution Token n-mal auf. Dass das zu vermessende Wort keinen sichtbaren Stackeffekt haben darf, dürfte einleuchten. Als Ergebnis wird die Laufzeit in Millisekundengenauigkeit geliefert. Soll keiner sagen, Benchmarks wären unnütz.

```
(ATmega32)> ' noop 10000 benchme
10000 iterations in 86 ms
 ok
(ATmega32)>
```

Für die Ermittlung der Aufrufzahl werden zwei Dinge benötigt: ein Counter pro Wort und ein Mechanismus, der sicherstellt, dass der Counter auch hochzählt. Das wird erreicht, indem man in die Codegenerierung eingreift und zusätzlichen Code einschleust. Der eigentliche Code wird dadurch naturgemäß langsamer, eine Kombination mit dem obigen `benchme` ist nicht empfehlenswert (Jede Messung beeinflusst das zu messende Objekt, der Effekt ist sicher nicht nur Physikern bekannt).

```
variable profiling?
: profile:on -1 profiling? ! ;
: profile:off 0 profiling? ! ;

: profiler profiling? @
  if 1 swap +! else drop then ;
\ re-define colon
: : :
  here 1 cells allot
  postpone literal postpone profiler
;
```

Die drei : hintereinander mögen auf den ersten Blick verwirren. Damit wird mit dem bestehenden : ein neues Wort namens : definiert, das als erstes das alte : aufruft. Im Anschluss wird im RAM eine Zelle für den Counter alloziert. Deren Adresse wird zusammen mit dem Execution Token (XT) für den `profiler` als erstes in das neu definierte Wort compiliert. Damit wird erreicht, dass jedes Wort, das mit dem neuen : definiert wird, einen Counter hat, der bei jedem Aufruf des Wortes inkrementiert wird.

Um auf die so ermittelten Daten zugreifen zu können, gibt es ein weiteres Wort, das aus dem XT eines Wortes die Adresse des zugehörigen Counters bereitstellt. Mit dieser Adresse kann man ganz normal wie mit jeder anderen Variablen auch umgehen.

```
: xt>prf ( xt -- addr )
  cell+ @i \ tricky [1]
;
```

`cell+` heißt bei amforth `2 +`, angewandt auf Code im Flash wird um zwei Flashzellen, also 4 bytes, weiter verwiesen. Die erste Flashzelle, die somit übersprungen wird, ist das Execution Token des Wortes selbst, die zweite enthält das Execution Token von `literal` für die Counteradresse (interessieren hier beide nicht). Die dann folgende dritte Zelle enthält die Adresse des Counters, die sodann ausgelesen wird. Das ist ein direktes Abbild des Codes, der eingeschleust wird, und ist zugleich ein Beispiel, wie extreme Optimierung Gefahren produziert.

Will man nun wissen, wie oft ein Wort aufgerufen wird, setzt man den Counter zurück, lässt das Testszenario durchlaufen und liest den Counter aus.

```
(ATmega32)> : foo ;
 ok
(ATmega32)> 0 ' foo xt>prf !
ok
(ATmega32)> ' foo xt>prf @ u.
0  ok
(ATmega32)> foo
```

```
ok
(ATmega32)> ' foo xt>prf @ u.
1  ok
(ATmega32)>
```

Für bereits existierende Worte bietet sich eine einfache Neudefinition an:

```
: + + ;
```

Diese Neudefinition verändert nichts an der Wirkung der Worte, schafft jedoch den Rahmen für den Einsatz des Tools. Immediate kann und muss natürlich nachgezogen werden, es ist jedoch zu bezweifeln, dass man solche Worte profilen will.

Call Tracer

Ganz ähnlich gestrickt ist der Call-Tracer. Er liefert den Namen des aufgerufenen Wortes und den Inhalt des Datenstacks zu Beginn des entsprechenden Wortes.

```
variable tracing?
: trace:on -1 tracing? ! ;
: trace:off 0 tracing? ! ;
: tracer tracing? @
   if cr itype space .s else drop drop then ;

\ speichert zusätzlich den Wortnamen, redundant
: : >in @ >r : r> >in !
  parse-name
  postpone sliteral
  postpone tracer
;
```

Der Eingriff in die Codeerzeugung ist dem des Profilers sehr ähnlich. Diesmal wird jedoch keine Variable zusätzlich alloziert, sondern der Name des gerade zu definierenden Wortes aus dem Eingabestrom zweimal herausgelesen: Zuerst für den normalen Word-Header und danach als String-Literal im Dictionary abgespeichert. Der Tracer bekommt den Namen des Wortes somit als addr/len (Flashaddresse) geliefert.

Die Ausgabe kann sehr schnell sehr unübersichtlich werden. Das hängt aber maßgeblich vom Format des genutzten `.s` ab.

```
(ATmega32)> : foo 1 ;
 ok
(ATmega32)> : bar 2 foo ;
 ok
(ATmega32)> : baz 3 bar ;
 ok
(ATmega32)> trace:on
 ok
(ATmega32)> baz

baz
bar 3
foo 2 3  ok
(ATmega32)> .s
1 2 3  ok
(ATmega32)
```

Debugshell

Das dritte Modul, das hier vorgestellt werden soll, ist die Debugshell. Sie ist ein kompletter, unabhängiger Kommandoprompt, der jederzeit aktiviert werden kann. Mit ihm stehen alle Forthbefehle zur Verfügung, man sollte sie aber mit Bedacht einsetzen. Die Debugshell wird beendet, sobald man nur ENTER drückt. Damit wird das unterbrochene Wort fortgesetzt.

Der Code ist (natürlich) simpel:

```
82 buffer: debugbuf
: (?) cr ." debug> " debugbuf dup 80 accept ;
: ?? begin (?) dup while (evaluate) repeat 2drop ;
```

Das `(evaluate)` ist eine Variante des normalen `evaluate`, das Strings im RAM erwartet[1]. Die Debugshell kann durch verschiedene Trigger ausgelöst werden: explizite Breakpoints im Code, aber auch Hardware-Events kommen in Frage.

Breakpoints im Code sind sowas wie der `.S` in der klassischen Zeit:

```
  defer breakpoint
  ' ?? is breakpoint
  \ ' noop is breakpoint
...
  : foo bar breakpoint baz ;
```

Durch den Einsatz von defer kann man Breakpoints recht einfach deaktivieren.

Die Aktivierung der Debugshell auf „Knopfdruck“ ist etwas trickier. Hierfür braucht man zunächst erst mal einen passenden und freien Interrupt. Eine Taste an einem geeignet konfigurierten Port kann schon ausreichend sein.

```
' ?? EXTERNALINT0 int!
\ EXTERNALINT0 int-trap

debug> rp@ hex .
82D
debug>
```

Das klappt sogar bei Endlosschleifen.

Natürlich hat diese Shell einige Eigenheiten, die man kennen und beachten sollte. Zum einen wird zwar der Datenstack nicht verändert, der Returnstack jedoch gut gefüllt. Es ist also nicht trivial, diesen für das unterbrochene Programm zu bearbeiten, was inbesondere den Abbruch der Endlosschleife betreffen wird. Außerdem ist der Einsatz der Shell im Interruptmodus darauf angewiesen, dass die Tastendrücke für die Kommandos *ohne* Interrupts ins System kommen.

[1] Amforth kann Strings im RAM und im Flash haben. Nur anhand der Adresse ist jedoch nicht entscheidbar, welcher Speichertyp vorliegt. Aus diesem Grund haben viele string–orientierte Worte zwei Ausprägungen.

Wer hats erfunden?

Viele Anregungen zu den Tools, teilweise auch Code, stammen aus mitunter esoterischen Diskussionen der Usenetgroup `comp.lang.forth`.

Dem aufmerksamen Leser ist bestimmt die große Ähnlichkeit zwischen dem Profiler und dem Calltracer aufgefallen. Warum sind die nicht sauber faktorisiert? Und warum speichert der Calltracer den Namen des Wortes doppelt? Der Grund hierfür ist weder technischer Art noch der Unwille, es „richtig“ zu machen. Vielmehr soll auch gezeigt werden, was man alles machen *kann* und welche Methoden zur Verfügung stehen. Damit kann man sich Wünsche wie „Der Calltracer soll nur maximal 3 Level tief anzeigen“ oder „Der Profiler soll unterschiedliche Aufrufwege getrennt zählen“ selbst umsetzen.

Amforth sammelt Ideen und Konzepte in einem „Kochbuch“: `http://amforth.sf.net/recipes/`. Eine Kategorie dort beschäftigt sich mit Debuggingtools, der vorliegende Artikel ist „nur“ eine Zusammenfassung der dortigen Highlights. Neben den hier vorgestellten Tools gibt es weitere, wie einen Memory-Watcher, der Speicherbereiche überwacht und Eingriffe beim Zugriff auf bestimmte Adressen erlaubt.

Referenzen

1. `http://amforth.sourceforge.net/recipes`

net2o–Debugging

Bernd Paysan

Angeregt durch den Artikel von Matthias Trute*, stelle ich hier Debugging–Tools, die ich im Rahmen meiner net2o–Implementierung entwickelt habe, vor.*

Einleitung

Auch Gforth hat nicht „den“ Debugger, obwohl Jens Wilke durchaus einen Single–Step–Debugger entwickelt hat, der allerdings davon ausgeht, dass das Forth indirect threaded code (ITC) ist, und deshalb nur mit `gforth-itc` läuft. Diesen Debugger verwenden wir eigentlich nie; Anton Ertl sagt, ein Stepping Debugger ist „reine Zeitverschwendung.“ Der übliche Debugger in Gforth ist `~~`, das ist ein aufgepepptes `.s`, das neben dem Stack auch noch die Position des Sourcecodes ausdruckt.

Darüber hinaus hat Gforth noch Assertions, also Codeteile, die überprüfen, ob bestimmte Bedingungen eingehalten werden, und die man im Production–Code lassen kann, weil `assert(` einen Modus hat, in dem es lediglich als Kommentar funktioniert. Und natürlich liefert Gforth einen brauchbaren Backtrace, wenn es irgendeine Exception im Code gibt (`gforth-fast` allerdings nicht). Der net2o–Code läuft nach Tuning nur 10% schneller mit `gforth-fast` (sprich: Die meiste Zeit verbringe ich außerhalb von Gforth), weshalb das kein großes Problem ist.

Das hat sich bei der net2o–Entwicklung [1] als nur bedingt nützlich herausgestellt. Ein Programm, das über Netzwerk mit einem anderen Programm kommuniziert, ist eben doch etwas anderes — die Bugs ergeben sich erst aus dem Zusammenspiel zweier Programme, die Bugs entstehen durch die Kommunikation bzw. durch die Fehler in dieser. Zudem will ich natürlich auch die Performance untersuchen, und brauche dafür etwas Profiling, zumindest grob über die Programm–Abschnitte. Der generell funktionierende Gforth–Profiler ist noch immer nicht fertig, und ich weiß auch nicht, ob ein generelles Tool das richtige für mich ist...

Schaltbarer Debug–Code

Letztlich will ich eigentlich nur an gewissen Stellen irgendetwas ausgeben (Printf–Debugging eben). Nur: Nicht immer, und nicht immer das Gleiche. Am besten per Kommandozeilen–Option schaltbar. Der Debugging–Code soll teilweise und komplett ausschaltbar sein (komplett bedeutet: Als Kommentar ausmaskiert). Ähnlich wie `assert(` ist also der Debug–Code in runde Klammern gepackt, und im Kommentarfall (kein Debugging) ist der startende Teil dann einfach ein Alias für `(`. Die schließende Klammer muss aber etwas anderes tun als bei Assertions; ich habe mich dann damit beholfen, einfach den Code von `)` in Gforths Assertions so zu ändern, dass er ein auf dem Stack übergebenes xt ausführt — das ist hinreichend verallgemeinert.

Die Debug–Marker werden dann mit einem `create..does>`–Wort definiert:

```
: debug: ( -- ) Create immediate false ,
  DOES>
    state @ IF  ]] Literal @ IF [[
        ['] debug) assert-canary
    ELSE  @ IF ['] noop assert-canary
        ELSE postpone (
        THEN
    THEN  ;
```

Das Ende eines Debug–Markes compiliert natürlich ein einfaches `THEN`:

```
: debug)  ]] THEN [[ ;
```

Und hier noch die Definition der geschlossenen Klammer aus Gforths aktuellem `assert.fs`, damit bei der Erklärung nichts verloren geht:

```
: )  ( -- )
  assert-canary <> abort" unmatched assertion"
  execute ; immediate
```

Damit können wir uns Debugging–Marker definieren, z.B.

```
debug: msg(
```

und die Benutzung sieht dann so aus:

```
msg( ." Expect reply" cr )
```

Zum Einschalten der (per Default ausgeschalteten) Messages verwenden wir

```
: +db ( "word" -- ) ' >body on ;
```

Also reicht ein `+db msg(`, und alle Messages werden ausgegeben. Und wie war das mit der Kommandozeile? Als Switch für die Messages hätte ich gerne so etwas wie `+msg`, das dann in `+db msg(` expandiert wird. Kann man machen:

```
Variable debug-eval
: +debug ( -- )
    BEGIN  argc @ 1 > WHILE
        1 arg s" +" string-prefix?  WHILE
        1 arg debug-eval $!
        s" db " debug-eval 1 $ins
        s" (" debug-eval $+!
        debug-eval $@ evaluate
        shift-args
    REPEAT  THEN ;
```

Damit wäre das Werkzeug fertig, das es mir erlaubt, bequem Teile des Programms an- und wieder auszuschalten, ohne am Code etwas ändern zu müssen (auch beliebig zur Laufzeit). Dass die Teile des Programms Debug–Code enthalten, ist natürlich reine Konvention.

Abschnittsweise Profiling

Weil ich schon so ein schönes Tool zum bedingten Aktivieren habe, markiere ich den Profiling–Code natürlich mit `profile(`. Mich interessieren abschnittsweise Laufzeiten, also etwa, wie lange net2o damit verbringt, Daten zu ver- und entschlüsseln, wie lange es zum Senden und Empfangen braucht, und was der restliche Overhead ist.

Als Zeitgeber verwende ich die Nanosekunden–Uhr von Linux, als eine Zelle, weil das einfacher handhabbar ist:

```
: ticks-u ( -- u )  ntime drop ;
```

Für jede Zeitmessung werden die seit der letzten Messung angesammelten ticks auf die zugehörige Variable aufsummiert:

```
Variable last-tick
: !@ ( value addr -- old-value )
  dup @ >r ! r> ;
: +t ( addr -- )
  ticks-u dup last-tick !@ - swap +! ;
```

Jetzt nur noch ein `create..does>`–Wort als Wrapper für +t definieren, und fertig sind die Timer:

```
Variable timer-list
: timer: Create 0 , here timer-list !@ ,
  DOES> profile( +t EXIT ) drop ;
timer: +calc
timer: +enc
```

Dazu gibt's dann noch ein Wort zum Initialisieren der Timer und eins zum Ausgeben aller Ergebnisse, siehe Listing. Die Benutzung der beiden oben definierten Wörter sieht dann so aus:

```
: wurst-outbuf-encrypt ( flag -- ) +calc
  wurst-outbuf-init
  outbuf packet-data
  outbuf body-size mem-rounds# encrypt-buffer
  drop >r wurst-crc r> 128! +enc ;
```

Am Anfang unseres abschnittsweisen Profiling rechnen wir alles, was bisher passiert ist, auf `calc-time` auf (der generische Overhead), und alles, was bis zum Ende der Routine passiert, auf `enc-time`. An Ende eines Laufs mit `+profile` auf der Kommandozeile bekommen wir dann eine Ausgabe wie diese (Loopback–Interface):

```
wait    : 0.10334549
send    : 0.217490602
rec     : 0.002389491
enc     : 0.160893002
calc    : 0.165475241
calc1   : 0.007980409
```

Das ist — leicht zu sehen — der Sender. Er verbringt etwa die Hälfte der Zeit mit Warten und Senden, und die andere Hälfte mit Verschlüsseln und generellem Overhead. Für den Receiver habe ich den generellen Overhead etwas verfeinert, was sich dann auch im calc1–Ergebnis zeigt:

```
wait    : 0.242345874
send    : 0.019372181
rec     : 0.050428231
enc     : 0.168066293
calc    : 0.112463045
calc1   : 0.077648203
```

Mit diesen doch ziemlich einfachen Mitteln hat man dann schon ein brauchbares Werkzeug in der Hand, um seinen Code genauer zu untersuchen. Manchmal trügt der Schein: Linux scheint beim Empfangen eigentlich nur zu warten. Nun: Darin ist das eigentliche Empfangen der Daten schon enthalten, das anschließende `recvfrom` liest nur noch den Puffer aus. Das geht natürlich schnell.

Literaturverzeichnis

[1] Bernd Paysan, *net2o repository,* `http://fossil.net2o.de/net2o`

Listing

```
\ debugging aids

: debug)  ]] THEN [[ ;

true [IF]
    : debug: ( -- ) Create immediate false ,
      DOES>
        state @ IF  ]] Literal @ IF [[
                ['] debug) assert-canary
            ELSE  @ IF ['] noop assert-canary
                ELSE postpone (
                THEN
        THEN  ;
[ELSE]
    : debug: ( -- )  Create immediate false ,
      DOES>
        @ IF  ['] noop assert-canary
        ELSE  postpone (  THEN ;
[THEN]

: hex[ ]] base @ >r hex [[ ; immediate
: ]hex ]] r> base ! [[ ; immediate
: x~~ ]] hex[ ~~ ]hex [[ ; immediate

\ debugging switches

debug: timing(
debug: rate(
debug: ratex(
debug: deltat(
debug: slack(
debug: slk(
debug: bursts(
debug: resend(
debug: track(
debug: data(
debug: cmd(
debug: send(
debug: firstack(
debug: msg(
debug: profile(
debug: stat(
debug: timeout(
debug: ack(

: +db ( "word" -- ) ' >body on ;

Variable debug-eval

: +debug ( -- )
    BEGIN  argc @ 1 > WHILE
            1 arg s" +" string-prefix?  WHILE
                1 arg debug-eval $!
                s" db " debug-eval 1 $ins
                s" (" debug-eval $+!
                debug-eval $@ evaluate
                shift-args
        REPEAT  THEN ;

\ timing measurements

Variable last-tick

: ticks-u ( -- u )  ntime drop ;
: !@ ( value addr -- old-value )   dup @ >r ! r> ;

: +t ( addr -- )
   ticks-u dup last-tick !@ - swap +! ;

true [IF]
    Variable timer-list
    : timer: Create 0 , here timer-list !@ ,
      DOES> profile( +t EXIT ) drop ;
    : map-timer { xt -- }
        timer-list BEGIN  @ dup  WHILE dup >r
                cell - xt execute r> REPEAT drop ;

    : init-timer ( -- )
        ticks-u last-tick ! [: off ;] map-timer ;

    : .times ( -- ) profile(
        [: dup body> >name name>string 1 /string
           tuck type 8 swap - 0 max spaces ." : "
           @ s>f 1n f* f. cr ;] map-timer ) ;
[ELSE]
    ' noop alias init-timer
    ' noop alias .times
    : timer: ['] noop alias immediate ;
[THEN]

timer: +calc1
timer: +calc
timer: +enc
timer: +rec
timer: +send
timer: +wait

\ Emacs fontlock mode: Highlight more stuff

0 [IF]
Local Variables:
forth-local-words:
    (
     (("debug:" "timer:")
      non-immediate (font-lock-type-face . 2)
      "[ \t\n]" t name (font-lock-variable-name-face . 3))
     ("[a-z]+(" immediate (font-lock-comment-face . 1)
      ")" nil comment (font-lock-comment-face . 1))
    )
End:
[THEN]
```

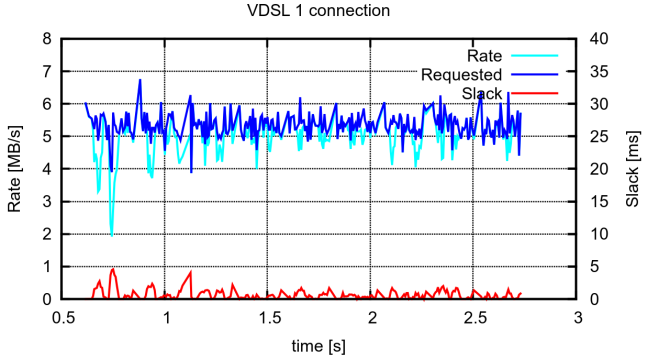


Abbildung 1: Graph einer Datenübertragung, aus den Debugging–Ausgaben gewonnen

Lokale Variablen in High–Level–Forth, ohne Immediacy und ohne Bracket–Tick

Fred Behringer

Es wird Wert darauf gelegt, ohne Immediate–Worte und ohne Einsatz von Assembler auszukommen. Returnstack–Manipulationen werden erlaubt. Für den lokalen Dienst werden global schon existierende Variablen herangezogen. Abgesehen vom Schwerpunkt der Betrachtung (Vermeidung von Immediate), liefert der Artikel nichts Neues.

Wie im Titel angedeutet, möchte ich zur Steuerung von lokalen Variablen nur ein paar allereinfachste Primitives verwenden (`r> >r @ ! swap dup`). Für alles andere soll auf High–Level–Forth zurückgegriffen werden. Vor allen Dingen soll aber auch von den leidigen Immediate–Worten weggeführt werden: Die Aufgabe eines Compilers besteht aus dem Compilieren des Quelltextes. Braucht man da wirklich Worte, die den Compiler mitten aus dem Compilier–Prozess heraus modifizieren — und bei denen man sich nicht immer merken kann, ob sie das im gegebenen Moment nun tun sollen oder nicht?

Es gibt Gründe, ein Forth–System aus einer minimalen Anzahl von Primitives heraus hochziehen zu wollen. Didaktische Gründe gehören ebenso dazu wie leichte Anpassbarkeit an eine Reihe diverser CPUs. Bernd Paysan hat das Thema „Minimal Set of Primitives“ schon vor Jahren zum Gegenstand seiner Untersuchungen gemacht [9], Willi Stricker hat sich in den letzten Jahren ausführlich mit dieser Frage beschäftigt [11]. Ein paar Ausführungen darüber gibt es auch in [3], [4], [5].

Von Hansen [7] stammt der Vorschlag, lokale Variablen in High–Level–Forth zu konstruieren und dazu Returnstack–Manipulationen zuzulassen. Als „Baumaterial“ werden von Hansen Variablen genommen, die global schon zur Verfügung stehen und die dann zwischendurch, innerhalb der betrachteten Colon–Definition, auch als lokale Variablen Dienst tun können. Nach Rückkehr vom (eventuellen) lokalen Einsatz erhalten sie dann wieder den bisherigen (globalen) Wert zugewiesen. Speicher- oder Zeitverschwendungen sollen dabei nicht berücksichtigt werden. Eine entsprechende Haltung wird auch in [3] und [5] vertreten.

Natürlich sind lokale Variablen nicht gerade das, was einem bei einer Diskussion über minimale Primitive–Sets als Erstes einfällt: Im Core–Word–Set von ANS–Forth wurden lokale Variablen nicht aufgenommen. Im ANS–Buch von Conklin/Rather [6] werden lokale Variablen nicht erwähnt, in den ANS–Forth–Empfehlungen [2] kommen lokale Variablen nur vage zur Sprache.

Andererseits kommt aber im Vorschlag von Hansen das Forth–Wort `[']` vor. Der implizite Tick `[']` (der Bracket–Tick) befindet sich Seite an Seite neben dem reinen Tick `'` im Core–Word–Set der ANS–Empfehlungen und gehört damit zu den unabdingbaren Worten eines jeden brauchbaren Forth–Systems.

Schon in Bezug auf den Bracket–Tick, aber auch bei der Frage nach einer eventuellen Vermeidung von Immediate–Worten, habe ich mit Interesse den Artikel „Semantics“ von Bernd Paysan im Heft 2/2012 [10] gelesen. Die Diskussion der Begriffe Immediacy, Ticks, Compile–only und Postpone hat mich gefesselt.

Ich hatte eigentlich vor, den High–Level–Forth–Charakter und die Vermeidung von sowohl Immediate–Worten wie auch Assembler (aber unter Zulassung von Manipulationen des Returnstacks) beizubehalten, scheiterte aber in meinen Überlegungen an Hansens Verwendung des Bracket–Ticks, den ich durch einfache Primitives, wie `>r` und `r>`, ersetzen wollte. Das müsste doch möglich sein! Ich bin noch am Nachdenken!

Was ich mir für den Moment in Erinnerung rief, war die Leichtigkeit, mit der Inline–Code in Colon–Definitionen in Turbo–Forth (und damit auch in ZF und allgemein in F83) in der Form `asm[ ... ]forth` eingebaut werden kann (eine bestechend einfache Form der Einbeziehung von Assembler in die betrachtete Hochsprache Forth). Ganz leicht zu handhaben, wenn man an die Umstände denkt, mit denen man noch bei Turbo–Pascal konfrontiert wurde. In Forth habe ich ja „normalerweise“ einen (gegebenen oder leicht hinzukonstruierbaren und leicht zu verwendenden) Assembler. Ich darf die Bezeichnungs–Analogie aus Turbo–Forth aufgreifen und führe im vorliegenden Artikel die Worte `local[` und `]global` zur Kennzeichnung von Bereichen ein, die innerhalb ein und derselben Colon–Definition liegen und in denen Variablen lokal eingesetzt werden (können). (Außerhalb der `[...]`–Klammern sollen sie dann wieder ihren bisherigen Wert zugeordnet bekommen.) Diese Form der Klammerung von lokalen Variablen ist nichts Ungewöhnliches. Hayes erinnert (schon 1990) in seiner viel zitierten Arbeit [8] an eine Analogie zum klammernden `if` in Paarung mit `then`. (Im Übrigen führt Hayes in [8] sieben Schriften über lokale Variablen aus einer Zeit schon wesentlich früher als 1990 an.)

Im Gegensatz zu dem Vorschlag von Hansen, bei welchem zur Einleitung des lokalen Bereichs nur ein einziges Wort, nämlich das Wort `global`, mit vorangesetztem Variablennamen (d.h., mit der die Variable kennzeichnenden Speicheradresse auf dem Datenstack), verwendet wird, und bei welchem die gesamte Colon–Definition als lokaler Bereich gilt, ist bei der im vorliegenden Artikel besprochenen Klammerung eine „beliebige“ Verschachtelung möglich. Natürlich muss dabei, ähnlich wie bei `if`

`...then`, auf eine paarweise Abstimmung geachtet werden. Und es darf mit den Ablagen auf dem Returnstack nichts „Ungehöriges“ vorgenommen werden. Zur Erläuterung habe ich im Beispiel 2 (siehe Listing) zwei nebengeordnete und eine übergeordnete lokale Klammerung konstruiert.

Dass bei der Handhabung von lokalen Variablen nicht von Colon–Definition zu Colon–Definition übergegriffen werden darf, ist klar. Und dass man bei Manipulationen des Returnstacks, die sich eventuell mit denjenigen bei den lokalen Variablen nicht vertragen, äußerst vorsichtig vorgehen sollte, versteht sich von selbst. Man beachte die Diskussionen über `do ...loops` bei Hayes [8].

Von den bei Stricker [11] verwendeten Primitives — und das ist das eigentliche Anliegen des vorliegenden Artikels — werden im unten stehenden Listing nur die folgenden verwendet: `r> >r @ ! swap dup`

Der Anzeige–Punkt (`.`) soll dabei nicht zählen. Genauso wenig wie der Begriff der Variablen selbst (die nach außen hin nichts weiter als eine Adresse im Speicher ist). Zählen sollen nur die eben genannten Primitives, soweit sie innerhalb von `local[` und `]global` geklammert erscheinen.

Referenzen

1. Allwright, Ray: From the Net: Minimal Word Sets. Forthwrite (FIGUK) 95, 3/1998.
2. ANS–Forth–Standard: The American National Standard for the Forth language (ANSI X3J14:1994).
3. Behringer, Fred: Drei Primitives weniger in Willi Strickers Forth–Minimalbasis. Vierte Dimension 4/2009.
4. Behringer, Fred: Über Flags in Forth. Vierte Dimension 1/2011, S.33–34.
5. Behringer, Fred: Kontrollstrukturen als Colon–Definitionen und ohne die Immediate–Eigenschaft?. Vierte Dimension 4/2011, S. 35–40.
6. Conklin, E.K., and Rather, E.D.: Forth Programmer's Handbook. Forth. Inc., California, USA.
7. Hansen, Henning: Leserbrief in der Vierten Dimension 1/1988, TU–Dänemark, Lynby.
8. Hayes, J.R.: Lokale Variablen – ein anderes Verfahren. Übersetzt von Gert–Ulrich Vack in: Vierte Dimension 2/1990.
9. Paysan, Bernd: Beitrag „Aus dem Netz,“ besprochen in [1].
10. Paysan, Bernd: Semantics. Vierte Dimension 2/2012.
11. Stricker, Willi: Minimaler Basis–Befehlssatz für ein Forth–System. Vierte Dimension 3/2009, S.15–17.

Listings

```
1   hex
2   variable xxx  \ Adresse der globalen Variablen xxx muss bereitstehen.
3   4711 xxx !    \ Zur Ueberpruefung mit (globalem) Wert belegen.
4
5                 \ local[ = Anfangsklammer fuer lokalen Bereich
6                 \ ------------------------------------------
7
8   : local[ ( ad -- )
9      r>         \ Zu local[ gehoerige Ruecksprungadresse
10     swap       \ mit ad vertauscht auf den Datenstack legen.
11     dup >r     \ ad auf den Returnstack legen.
12       @ >r     \ Wert von ad auf den Returnstack legen.
13         >r ;   \ Ruecksprungadresse von local[ wieder auf den Returnstack.
14
15                \ ]global = Endklammer fuer lokalen Bereich
16                \ ------------------------------------------
17
18  : ]global ( -- )
19     r>         \ Ruecksprungadresse von ]global auf den Datenstack legen.
20     r>         \ Bisherigen Wert vom zugehoerigen ad vom Returnstack holen.
21     r>         \ Adresse vom zugehoerigen ad vom Returnstack holen.
22      !         \ Urspruenglichen Wert der Variable ad wiederherstellen.
23     >r ;       \ Ruecksprungadresse von ]global wieder auf den Returnstack.
24
25
26  : zzz ( -- )  \ Beispiel 1: Einfache [...]-Klammer
```

```
27      xxx @ .     \ Bisherigen Wert der globalen Variablen xxx ausgeben.
28      xxx local[ \ xxx zur lokalen Variablen erklaeren.
29      0815 xxx ! \ Die momentan lokale Variable xxx mit 0815 belegen.
30      xxx @ .     \ Diesen (lokalen) Wert ueberpruefen.
31      ]global     \ xxx wieder zur globalen Variablen machen.
32      xxx @ . ;   \ Bisherigen (globalen) Wert ueberpruefen.
33
34    variable xx1  4711 xx1 !
35    variable xx2  4712 xx2 !
36    variable xx3  4713 xx3 !
37
38    : www ( -- )  \ Beispiel 2: Schachtelung mit drei lokalen Klammern
39                  \ -------------------------------------------------
40      xx3 @ .     \ Bisherigen Wert der globalen Variablen xx3 ausgeben.
41      xx3 local[ \ xx3 zur lokalen Variablen erklaeren.
42      0817 xx3 ! \ Die momentan lokale Variable xx3 mit 0817 belegen.
43      xx3 @ .     \ Diesen (lokalen) Wert ueberpruefen.
44                  \ =================================================
45      xx1 @ .     \ Bisherigen Wert der globalen Variablen xx1 ausgeben.
46      xx1 local[ \ xx1 zur lokalen Variablen erklaeren.
47      0815 xx1 ! \ Die momentan lokale Variable xx1 mit 0815 belegen.
48      xx1 @ .     \ Diesen (lokalen) Wert ueberpruefen.
49      ]global     \ xx1 wieder zur globalen Variablen machen.
50      xx1 @ .     \ Bisherigen (globalen) Wert ueberpruefen.
51                  \ -----------------------------------------
52      xx2 @ .     \ Bisherigen Wert der globalen Variablen xx2 ausgeben.
53      xx2 local[ \ xx2 zur lokalen Variablen erklaeren.
54      0816 xx2 ! \ Die momentan lokale Variable xx2 mit 0816 belegen.
55      xx2 @ .     \ Diesen (lokalen) Wert ueberpruefen.
56      ]global     \ xx2 wieder zur globalen Variablen machen.
57      xx2 @ .     \ Bisherigen (globalen) Wert ueberpruefen.
58                  \ =================================================
59      ]global     \ xx3 wieder zur globalen Variablen machen.
60      xx3 @ . ;   \ Bisherigen (globalen) Wert ueberpruefen.
61                  \ -------------------------------------------------
62
63    \ Der Aufruf von www ergibt, gleich geschachtelt zusammengefasst:
64    \ 4713 817   4711 815 4711
65    \            4712 816 4712   4713
66
67    \ Zur Ueberpruefung des hier Gesagten halte ich mich an Turbo-Forth in der
68    \ 16-Bit-Version (siehe taygeta.com oder Bremen mirror).
```

Kurtosis

Rafael Deliano

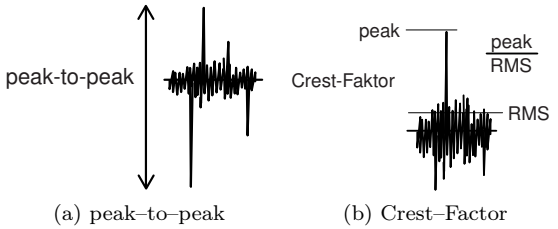
Eigentlich der Name eines Kennwerts aus der Statistik. In den 70er Jahren durch Anwendung in der Wälzlagerdiagnose bekannt geworden. Momente sind für die Beschreibung von Rauschsignalen und in der Mustererkennung nützlich.

Momente

Gaußkurve wurde schon in [1] behandelt und die Aufnahme entsprechender Histogramme in [2]. Hier wird angenommen, dass sich als Messwerte z.B. N=1024 Samples als Tabelle im RAM befinden. Das erste Moment ist der DC–Mittelwert, der für spätere Normierungen berechnet werden muss (Bild 2). Aus dem zweiten Moment kann man den Effektivwert berechnen, ein oft von der Anwendung benötigter Kennwert. Das zweite Moment ermöglicht auch die Berechnung der Standardabweichung, wenn man in einem weiteren Normierungsschritt den DC–Offset entfernt. Das dritte zentrale Moment zeigt die Asymmetrie an. Die formal bevorzugte Variante hat für Gaußkurve den Wert 0. Die quadrierte Variante ergibt immer eine vorzeichenlose Zahl, in vielen Anwendungen wird diese Form der Kennzahl bevorzugt. Das vierte zentrale Moment ist schließlich die Kurtosis, die die symmetrische Verteilung anzeigt. Ihr Wert ist 3,0, wenn die Form einer Gaußkurve entspricht. Die formale Variante normiert diesen Wert wieder auf 0. Jedoch wird in Anwendungen durchwegs die unnormierte Formel verwendet, die wieder eine vorzeichenlose Zahl ergibt. Generell zeigen ungerade Momente die Lage der Spitze der Verteilung relativ zum Mittelwert an. Während gerade Momente das Verhältnis Breite zu Höhe der Kurve angeben. Bei Wälzlagerschäden sollen besonders Spikes im Signal erkannt werden. In vielen Untersuchungen wurden deshalb weitere Kennwerte, basierend auf Spitzenwertdetektoren, ausgewertet. Peak–to–Peak (Bild 1a) variiert abhängig von der Signalstärke. Man bevorzugt aber Kennwerte, die unabhängig von der absoluten Signalstärke sind.

Crest–Faktor

Diese Eigenschaft hat auch das Verhältnis Spitzenwert zu Effektivwert (Bild 1b). Der Kennwert ist bekannt, weil er z.B. in der Spezifikation für RMS–Messgeräte zu finden ist. Er wäre zudem aufwandsärmer als Kurtosis berechenbar. Ein Vergleich wurde in [4] vorgenommen. Für Sinus ergeben beide einen konstanten Kennwert (Bild 4).



(a) peak–to–peak (b) Crest–Factor

Abbildung 1: Crest und P2P

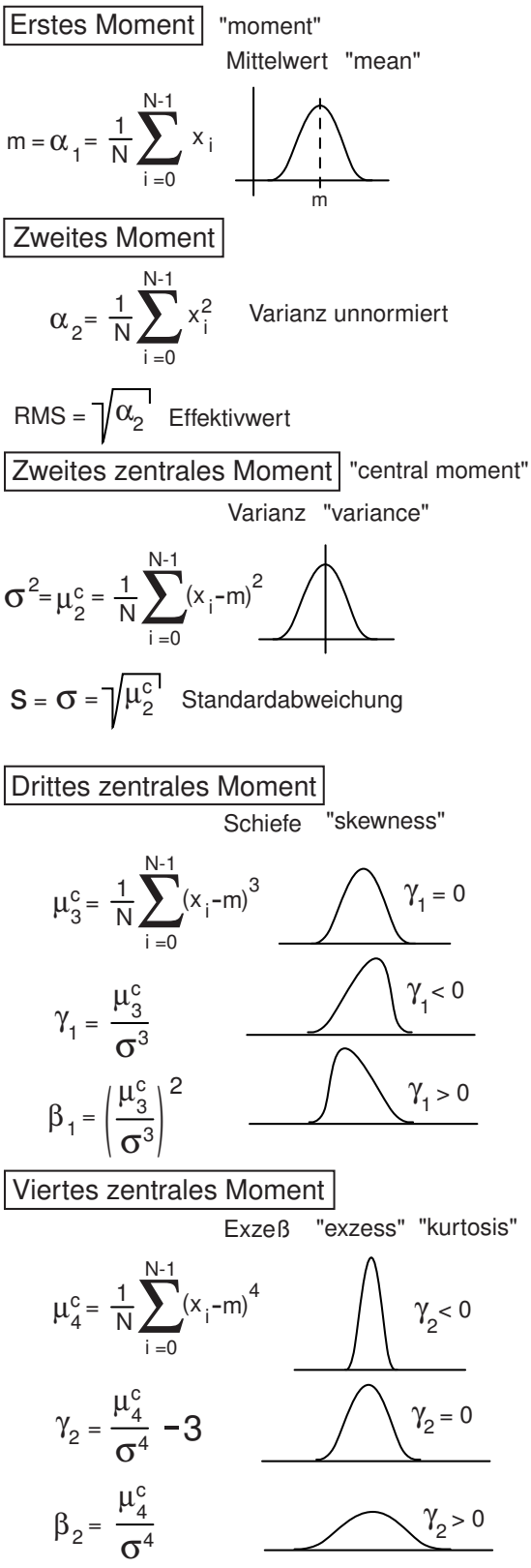


Abbildung 2: Momente

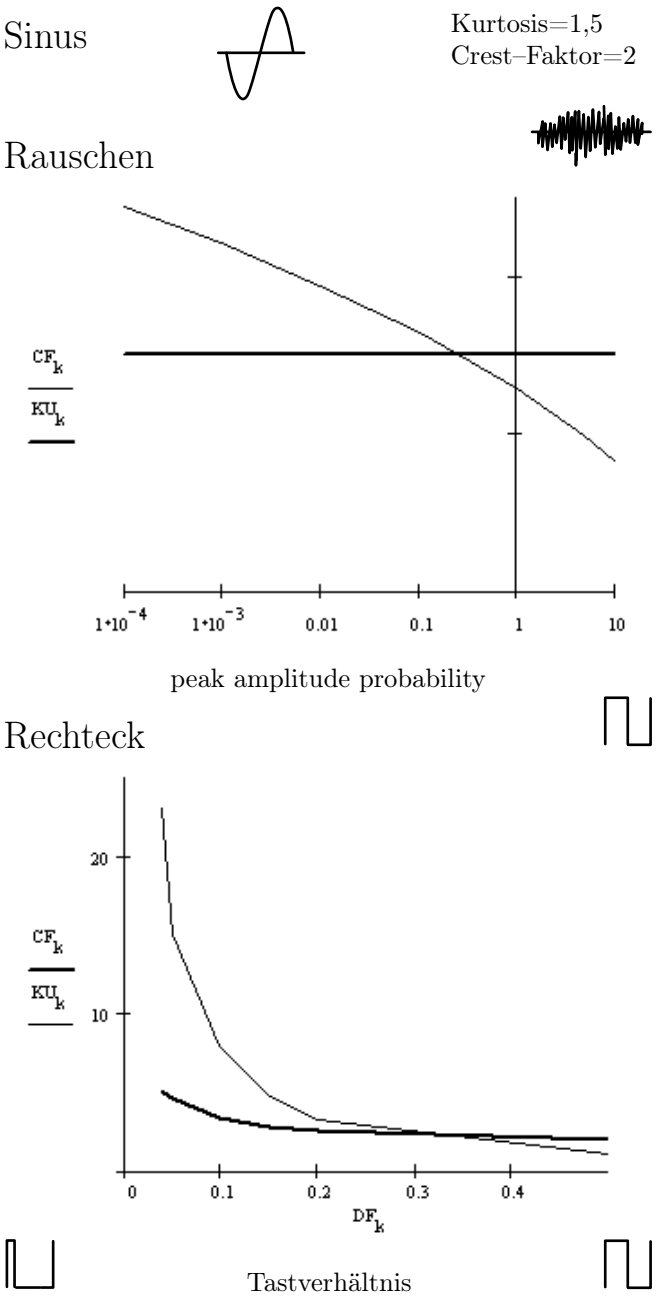


Abbildung 3: Vergleich Kurtosis und Crest–Faktor

Da Rauschen keinen festen Spitzenwert hat, definiert man dafür die Wahrscheinlichkeit, mit der es einen Schwellwert überschreitet. Der Crest–Faktor variiert hier, während Kurtosis völlig stabil ist. Rechteck mit veränderbarem Tastverhältnis ist ein einfaches Testsignal, das spike–förmige Charakteristik hat. Kurtosis reagiert darauf empfindlicher als Crest–Faktor.

Implementierung

Hier wird angenommen, dass 1024 Worte im 16–Bit–unsigned Format im Speicher vorliegen, wie es aus vielen 12–Bit–A/D–Wandlern ausgelesen wird. Die werden dann per 8000h - auf signed 2er–Komplement umgerechnet. Sie haben dann einen künstlichen, etwas idealisierten Nullpunkt. Als Testdaten eignet sich jedes Sammelsurium von Bits. Hier wurde ein Abschnitt des nanoFORTH–Programmspeichers verwendet. Gaußverteilung (Bild 5) ist nicht zu erkennen, aber die Anforderungen sind ja niedrig. Anhand von Referenzwerten, die mit Mathcad gerechnet wurden (Bild 6), kann man seine Software debuggen (Bild 7). Alternativ kann man auch mit dem Rauschgenerator und der Umwandlung auf Gaußverteilung aus [1] einen Datensatz erzeugen (Bild 8, 9). SK hat nun ca. den Sollwert 0 und KU ca. 3,0. Die Berechnung der Momente belegt auf dem GP32 zwar nur 1,2 kByte Programm, es müssen aber 1 kByte Arithmetikbefehle nachgerüstet werden. Abgesehen von 32–Bit–Shiftbefehlen sind für die Momente 3 und 4 weitere Multiplikationen nötig, wie in [1] beschrieben:

```
UD*  \( UD1 UN1 -- UD2 )
UD*/ \( UD1 UD2 UD3 -- UD4 )
```

Die Berechnung dieser beiden Momente dauert auf einem 2,45MHz–GP32 auch deutlich länger (Bild 10).

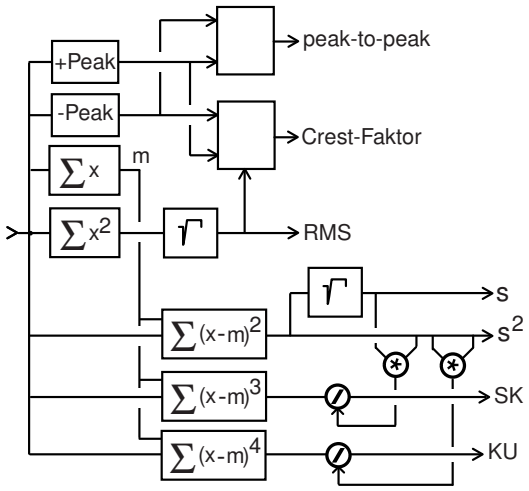


Abbildung 4: Ablauf Berechnung

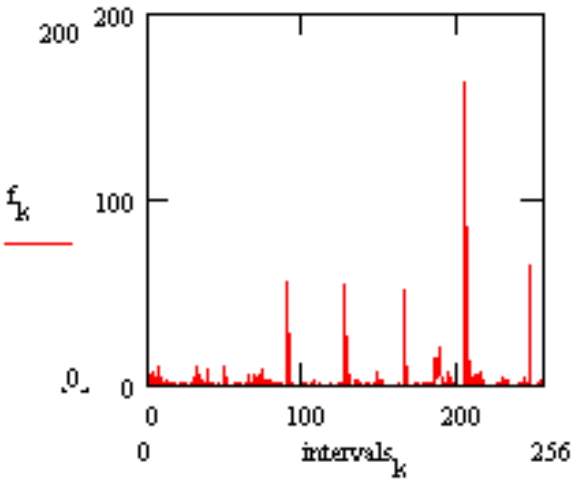


Abbildung 5: Histogramm Testdaten Speicher

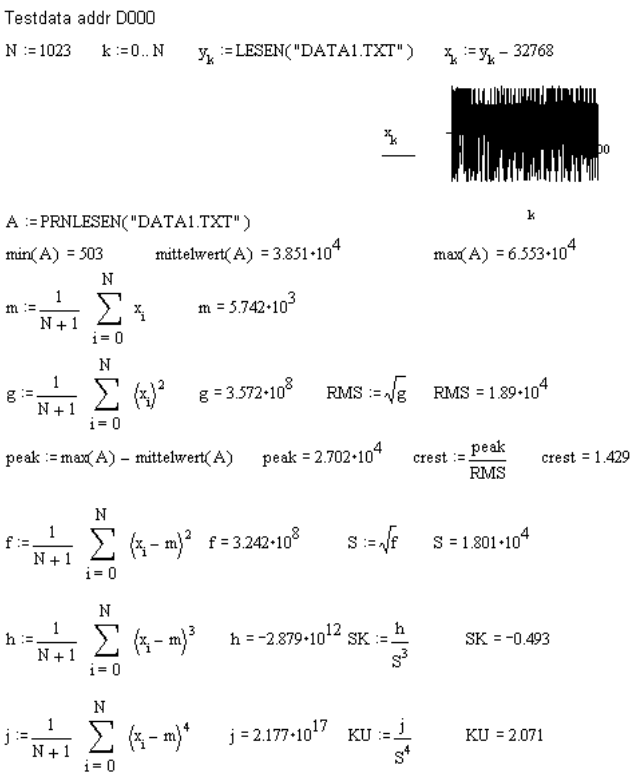


Abbildung 6: Testdaten Speicher mit Mathcad

```
---- ---- ---- | moment1 m1.
MEAN= 5741
P-PEAK-MAX=65527
N-PEAK-MAX= 503
PEAK-PEAK=65024
---- ---- ---- | moment2 crest!
---- ---- ---- | m2.
RMS=18899
PEAK-MAX=27018
CREST= 142
---- ---- ---- | nmoment2 nm2.
S^2=0324224976 S=18006
---- ---- ---- | nmoment3 nm3.
SK= -49
---- ---- ---- | nmoment4 nm4.
KU= 207
---- ---- ---- |
```

Abbildung 7: Testdaten Speicher mit GP32

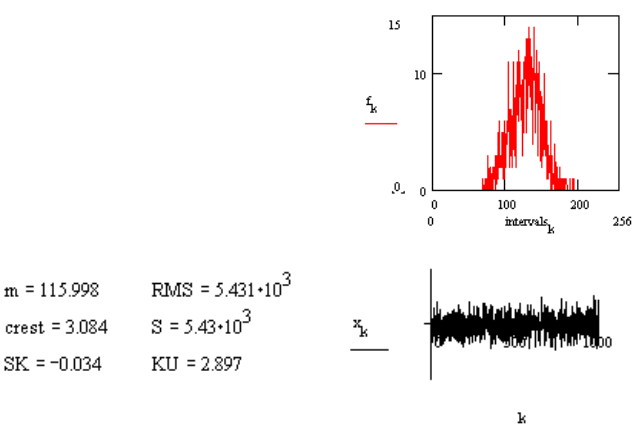


Abbildung 8: Testdaten Gauß Mathcad

```
---- ---- ---- | moment1 m1.
MEAN= 115
P-PEAK-MAX=49631
N-PEAK-MAX=17574
PEAK-PEAK=32057
---- ---- ---- | moment2 crest!
---- ---- ---- | m2.
RMS= 5430
PEAK-MAX=16748
CREST= 308
---- ---- ---- | nmoment2 nm2.
S^2=0029478026
S= 5429
---- ---- ---- | nmoment3 nm3.
SK= - 3
---- ---- ---- | nmoment4 nm4.
KU= 289
---- ---- ---- |
```

Abbildung 9: Testdaten Gauß mit GP32

moment1	0,3sec
moment2 crest!	0,3sec
moment2	0,45sec
moment3	2,5sec
nmoment4	2,5sec

Abbildung 10: Geschwindigkeit

SPI–RAM

Das intern verfügbare RAM des GP32–Controllers beträgt nur 256 Byte. Nötig ist deshalb für solche Anwendungen ein externes 32kByte–SPI–RAM. Der Zugriff darauf ist langsam, man will die Samples deshalb nicht öfter lesen als nötig. Es sind minimal zwei Schleifen nötig, die das RAM lesen (Bild 4). Erstmal müssen die ersten beiden Momente und die Spitzenwerte bestimmt werden. Erst wenn der Mittelwert m vorliegt, kann man die drei zentralen Momente berechnen. Hier wurde auf diese Optimierung verzichtet, weil sie den Programmfluss unübersichtlicher macht.

Stochastisches Samplen

Die Daten müssen nicht kontinuierlich mit voller Abtastrate gesampelt werden. Stochastisches Samplen [2] ist anwendbar und reduziert die Anforderungen an den Controller. Es ist jedoch hier vorgesehen, dass die Daten blockweise vorliegen. Längere Datensätze ergeben dabei wegen der Mittelung stabilere Kennwerte. Bei einem Kugellager wäre z.B. die Periode einer Umdrehung bei minimaler Drehzahl ein sinnvoller Wert.

Histogramm

Wenn am Gerät eine grafische Ausgabemöglichkeit vorhanden ist, kann es erwünscht sein, dass generell für jeden Datensatz das Histogramm berechnet und dargestellt wird. Das geschieht typisch mit reduzierter Quantisierung, also 256 Linien statt 65535 (Bild 5). Wenn man

die Momente dann mit Hilfe des Histogramms berechnet, sinkt der Rechenaufwand entsprechend. Allerdings natürlich auch die Genauigkeit der Werte.

Literaturverzeichnis

[1] emb (12) *Zentraler Grenzwertsatz* http://www.embeddedforth.de/emb12.pdf

[2] emb (14) *Histogramm–Rechner* http://www.embeddedforth.de/emp14.pdf

[3] emb (10) *Division, Multiplikation*http://www.embeddedforth.de/emb10.pdf

[4] Braun *„Mechanical Signature Analysis“* Academic Press 1986

Listings

```
<| \ Kurtosis

D000 CONSTANT BUFFER  \ 1024d 16 bit samples
2 ZVARIABLE   MEAN        \ signed
2 ZVARIABLE   P-PEAK-MAX  \ unsigned
2 ZVARIABLE   N-PEAK-MAX  \ unsigned
2 ZVARIABLE   PEAK-PEAK   \ unsigned
2 ZVARIABLE SIGN          \ local scratchpad

\ UD*/        \ ( UD1 UD2 UD3  -- UD4 )
              \ UD4 = ( UD1 * UD2 ) / UD3
\ UD*         \ ( UD1 UN1  -- UD2 )
\ 1DSHIFT>    \ ( UD1  -- UD2 ) 1x LSR
\ D2/         \ ( UD1  -- UD2 ) ASR
\ SQRT    \ ( UD1  -- UN1 ) UD1 =< 7FFF FFFF
\ ND.         \ ( UN1 -- ) print UN1 decimal
\ D.          \ ( UD1 -- ) print UD1 decimal
              \ UD1 =< 7FFF FFFF

: BUFFER@  \ ( I -- N1 ) I = 0 - 1023d
1<SHIFT BUFFER + @ 8000 - ;

: SIGN! DUP 8000 AND SIGN ! ; \ ( N1 -- N1 )

: SIGN@ SIGN @ ;  \ (   -- Flag )
                  \ negative = 8000h

: PEAK-MAX?             \ ( N1  -- N1 )
DUP 8000 +              \  -- N1 UN1 )
DUP P-PEAK-MAX @ U> IF DUP P-PEAK-MAX ! THEN
DUP N-PEAK-MAX @ U< IF DUP N-PEAK-MAX ! THEN
DROP ;

: MOMENT1         \ (  --  )
                  \ in: 1024 samples
                  \ out: MEAN
                  \      PEAK-PEAK
                  \      P-PEAK-MAX
                  \      N-PEAK-MAX
0 P-PEAK-MAX ! FFFF N-PEAK-MAX !
0 0 D% 1023 0 DO  \ ( sum --  )
  I BUFFER@
  PEAK-MAX?
  N->D D+
LOOP              \ (  -- D1  )
DUP 8000 AND SIGN !
DABS D% 1024 U/
SIGN @ IF NEGATE THEN MEAN !
P-PEAK-MAX @ N-PEAK-MAX @ - PEAK-PEAK ! ;

: M1.   ... ; \ (  --  ) print

2 ZVARIABLE RMS    \ unsigned

: MOMENT2          \ (  --  )
                  \ in: 1024 samples
                   \ out: RMS
0 0 D% 1023 0 DO   \ ( sum --  )
  I BUFFER@
  ABS DUP U*
  8DSHIFT>
  D+
LOOP               \ (  -- UD1  )
1DSHIFT> 1DSHIFT>
SQRT RMS ! ;

2 ZVARIABLE PEAK-MAX \ unsigned rel. to MEAN
2 ZVARIABLE CREST    \ unsigned rel. to MEAN

: CREST!             \ (  --  )
P-PEAK-MAX @ 8000 -  \ to signed
MEAN @ -            \ positive rel. to MEAN
N-PEAK-MAX @ 8000 -  \ to signed
ABS
MEAN @ -               \ positive rel. to mean
2DUP U< IF SWAP THEN DROP  \  -- UN1 )
DUP PEAK-MAX !
D% 100 U* RMS @ U/ CREST ! \ 100*peak/RMS
;

: M2.  ... ; \ (  --  ) print

2 ZVARIABLE S
4 ZVARIABLE S^2

: NMOMENT2        \ (  --  )
                 \ in: 1024 samples in BUFFER
                  \ out: S
                  \      S^2
0 0 D% 1023 0 DO          \ ( sum --  )
  I BUFFER@
  N->D MEAN @ N->D D- DABS 1DSHIFT> DROP
  DUP U*
  4DSHIFT> 1DSHIFT> 1DSHIFT>
  D+
LOOP                    \ (  -- UD1  )
1DSHIFT> 1DSHIFT>
2DUP S^2 D!    SQRT S ! ;

: NM2.  ... ; \ (  --  ) print

2 ZVARIABLE SK     \ unsigned,  1,00 = 100d

: NMOMENT3        \ (  --  )
                  \ in: 1024 samples in BUFFER
                  \ out:  SK
0 0 D% 1023  0 DO         \ ( sum --  )
  I BUFFER@
  N->D MEAN @ N->D D- D2/ DROP
  SIGN! ABS
  DUP DUP U* ROT 0
  0000 0020 UD*/           \ (  -- sum UD1  )
  SIGN@ IF DNEGATE THEN
  D+
LOOP                        \ (  -- D1  )
SIGN! DABS                  \ (  -- UD1 )
8DSHIFT> 8DSHIFT>           \ 1/2^16
D% 25600 UD*                \ 100*2^8
S^2 D@ S @ 0 2000 0000 UD*/ \
SWAP DROP                   \ 1/2^16
U/                         \ scaled to 1,00 = 100
1<SHIFT 8SHIFT> SK ! ;

: NM3.   ... ; \ (  --  ) print

2 ZVARIABLE KU  \ unsigned, scaled to 1,00 = 100

: NMOMENT4              \ (  --  )
```

```
                          \ in: 1024 samples
                           \ out:  SK
  0 0 D% 1023  0 DO            \ ( sum --  )
    I BUFFER@
    N->D MEAN @ N->D D- DABS 1DSHIFT> DROP
    DUP U* 2DUP 0000 8000 UD*/  \ (  -- sum UD1  )
                                \ 8000 0000 = 2^31
    4DSHIFT>                     \ 2^4
    D+
  LOOP                      \ (  -- UD1  ) Zähler
```

```
  8DSHIFT> 8DSHIFT>          \ 1/2^16
  D% 25600 UD*               \ 100*2^8
  S^2 D@ 2DUP 0000 1000 UD*/  \ 1000 0000 = 2^28
  SWAP DROP                   \ 1/2^16
  U/                         \ scaled to 1,00 = 100
  1<SHIFT 8SHIFT> KU ! ;

  : NM4.  ... ; \ (  --  ) print
  |>
```

Anwendung Wälzlager

Rafael Deliano

Für den Betreiber einer Maschine ist die Früherkennung des sich abzeichnenden Ausfalls, z.B. eines Kugellagers, von erheblicher wirtschaftlicher Bedeutung

Typisch kommen die Rohdaten aus einem Beschleunigungssensor. In einer von der British Steel Corporation veranlassten Untersuchung [1] wurden daraus skewness, Kurtosis, Crest–Faktor, peak–to–peak als mögliche Kennwerte berechnet. In [6] wurde neben RMS, peak–to–peak und Kurtosis als Kontrolle auch die Temperatur des Lagers mit einem Thermoelement gemessen (Bild 1). In der Realität sind die Messwerte aber selten so schön wie im Modellversuch. Kein Kennwert lieferte isoliert ein sicheres Ergebnis. In der weiteren Entwicklung konzentrierte sich das Interesse auf die Kombination von Kennwerten. Gern verwendet wurde Kurtosis [2].

Ein unbeschädigtes Lager (Bild 2) liefert weißes Rauschen als Signal und das kann über Kurtosis recht sicher erkannt werden. Der Wert 3,0 streut dabei oft nur um ±8% [1]. Geringe Beschädigung führt zu wenigen, aber ausgeprägten Spikes. Auch dieser Zustand ist über Kurtosis mit Werten 4 ... 11 gut erkennbar. Ein stark beschädigtes Lager liefert jedoch so viele Spikes, dass diese einen kontinuierlichen Rauschpegel bilden, bei dem der Kurtosis–Wert wieder absinkt.

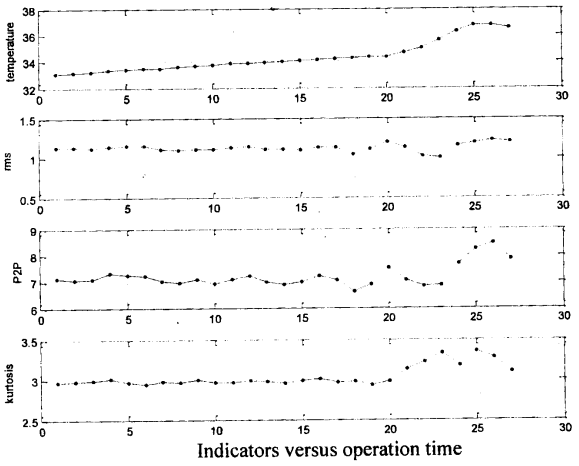


Abbildung 1: Daten aus Testaufbau [6]

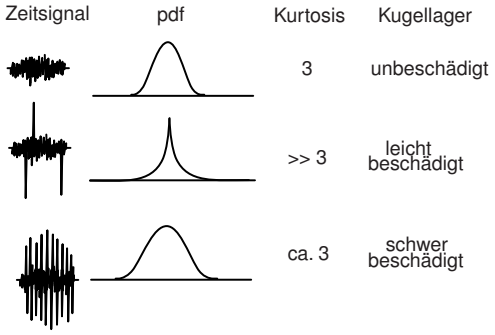


Abbildung 2: schematische und stark vereinfachte Signale eines Wälzlagers

Analyzer

In kommerziell verfügbaren Geräten wurde das Signal deshalb in Frequenzbänder zerlegt [3] (Bild 3), bei denen neben dem Kurtosis–Koeffizienten K auch noch ein Wert G für die absolute Signalstärke ausgewertet wird. Steigt die Zahl der Spikes, steigt der Kurtosis–Wert in den höheren Frequenzbändern an. Ein stark beschädigtes Lager ist zwar nicht mehr sicher über Kurtosis detektierbar, kann aber über den hohen absoluten Signalpegel erkannt werden.

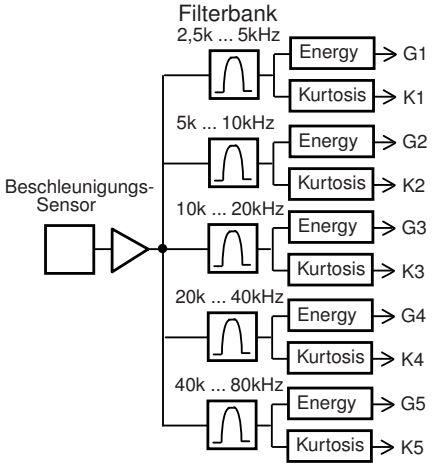


Abbildung 3: Analyse–Gerät

Literaturverzeichnis

[1] Dyer, Stewart *„Detection of rolling element bearing damage by statistical vibration analysis“* Trans. ASME J. Mech. Design April 1978

[2] Rush *„Kurtosis — A Crystal Ball for Maintenance Engineers“* Iron and Steel International 2/1979

[3] Nojack *„Lagerschaden–Früherkennung mit der Kurtosis–Methode“* Elektronik 17/1981

[4] Mathew, Alfredson *„The condition monitoring of rolling element bearings using vibration analysis“* ASME Trans. J. Vib Acoust. Stress. Reliab. Design 106 1984

[5] Thorpe, Martin, Connolly *„The Choice of Parameters for Bearing Monitoring and Diagnosis“* Proceedings Conference Condition Monitoring 1991

[6] Zhang et al *„Asset health reliability estimation based on condition data“* Queensland University of Technology

Kontinuierliche RMS–Berechnung

Rafael Deliano

Der Ablauf ist prinzipiell auch für andere Kennwerte, die auf Momenten beruhen, anwendbar.

Die Formel (Bild 1a) für die blockweise Berechnung kann man auch als Flussdiagramm darstellen (Bild 1b), das nun stark überlappte Datenblöcke sample-by-sample berechnet. Der Averager [1] ist nur eine von mehreren möglichen Varianten für ein Tiefpass–Filter (Bild 1c). In Anlehnung an analoge RMS–Schaltungen wird meist ein 1pol IIR bevorzugt [2] (Bild 1d). Bei genügender Glättung und stationärem Eingangssignal werden alle RMS–Varianten identisches Ergebnis liefern. Während veränderlichem Signal ist das Verhalten wegen der typisch nicht exakt definierten Filter unterschiedlich.

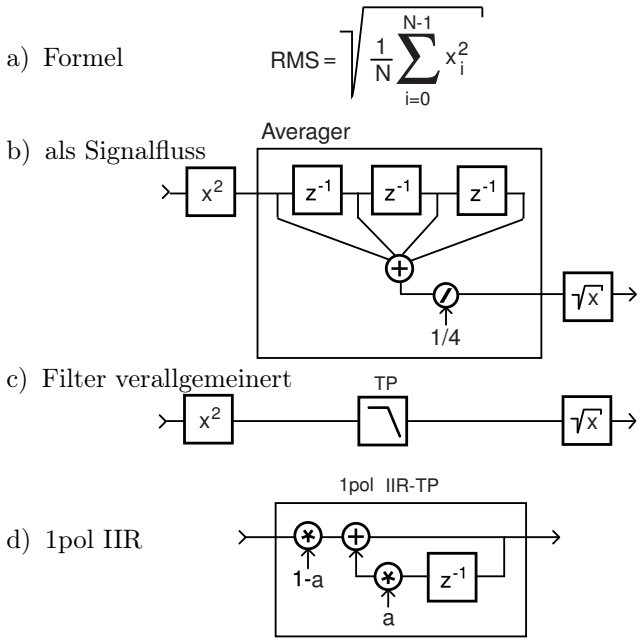


Abbildung 1: Filter

Implementierung

Verwendet wird eine für lange Zeitkonstanten günstigere Struktur des IIR (Bild 2). Für ein vorzeichenbehaftetes Eingangssignal mit Mittelwert m=0 kann man das Signal einfach durch Absolutwertbildung auf das vorzeichenlose Format bringen. Dann ist die Multiplikation auf dem GP32 günstiger. Die Wortlänge verdoppelt sich und verkürzt sich dann erst wieder durch Ziehen der Wurzel. Ein typischer SQRT–Algorithmus [3] verwendet sukzessive Approximation und benötigt bei der hier verwendeten Wortlänge 16 Schritte mit entsprechender Rechenzeit.

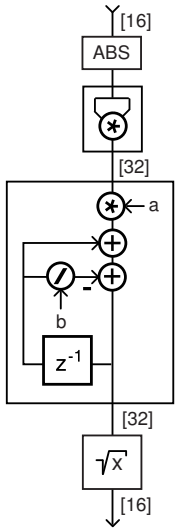


Abbildung 2: Flussdiagramm

tracking SQRT

Abhängig von seiner Zeitkonstante ergibt sich am Ausgang des Filters ein Signal mit begrenzter Anstiegszeit. Für vereinfachte Simulation kann man die Sprungantwort durch eine Rampe darstellen (Bild 3), die zudem ja auch die Zeitkonstante des 1pol–Filters abbildet. Für schnelle Anstiegszeit (Bild 4a) ist der übliche SQRT–Algorithmus weiterhin am besten geeignet. Ein simpler Tracking–Algorithmus ist verwendbar, wenn die Zeitkonstante sehr lang ist (Bild 4c). Der Fall hat aber keine praktische Bedeutung, weil bei fester Schrittweite ±1 die Folgegeschwindigkeit viel zu langsam ist. Im Übergangsbereich (Bild 4b) ist ein Tracking–Algorithmus mit variabler Schrittweite möglich.

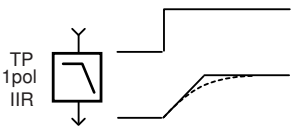


Abbildung 3: Sprungantwort 1pol TP

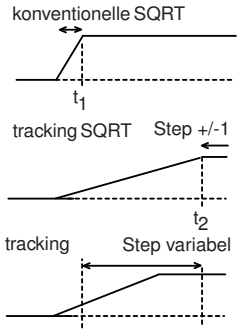


Abbildung 4: Zeitkonstanten a/b/c

Schrittweiten

Für das Testsignal, d.h. die Rampe, sind die erforderlichen Schrittweiten der SQRT in Bild 5 dargestellt. Es sind mehrere Schrittweiten der Rampe, also Zeitkonstanten des Tiefpassfilters geplottet. Da bei der SQRT–Funktion die Steilheit bei kleinen Werten am höchsten ist, beginnt die Kurve hier mit der maximalen Schrittweite, fällt danach aber steil ab, vgl. die logarithmische Ansicht.

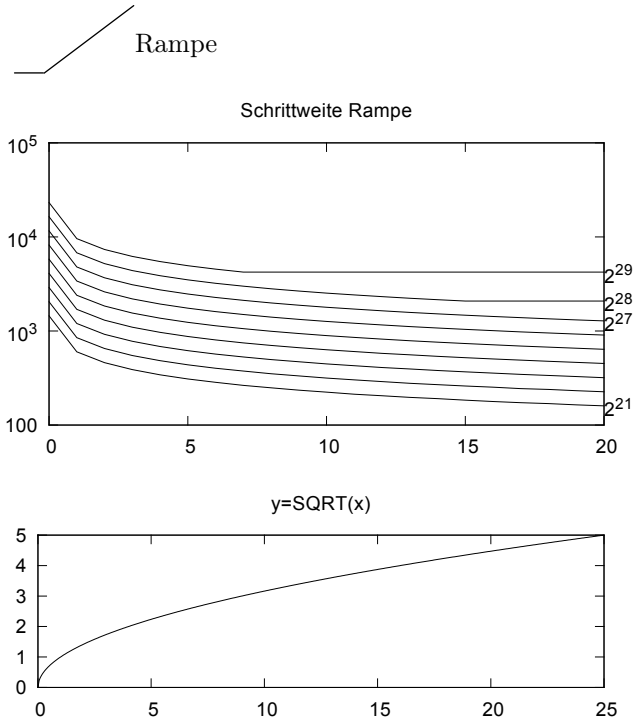


Abbildung 5: Maximale Schrittweite

Algorithmus

Um dem folgen zu können, wird im Programm die Schrittweite um den Faktor $\times 2$ oder $\times 0,5$ verändert (Bild 6). IN", oIN, OUT" und STEP sind interne Variablen. STEP muss auf die kleinste Schrittweite 1 initialisiert werden. Die größte Schrittweite MAX–STEP legt man passend zur Zeitkonstante des Filters fest. Hier wurde mit oIN" noch ein Patch eingebaut, der, wenn das Eingangssignal konstant ist, die Schrittweite auf 1 setzt, damit in diesem Fall der Fehler minimiert wird.

Test

Als Eingangssignal wurden neben Rampe auch Bereiche, in denen das Signal statisch ist, verwendet (Bild 7). Ausgangssignal ist die normale SQRT und die Tracking–SQRT. Durch Differenzbildung kann man dann das Fehlersignal bestimmen. Der Fehler im Ausgangssignal wird aber in vielen Anwendungen durch die deutlich geringere Rechenzeit aufgewogen. Tabelle 1 zeigt die Werte für einen MC68HC908GP32 mit 2,54 MHz Busfrequenz.

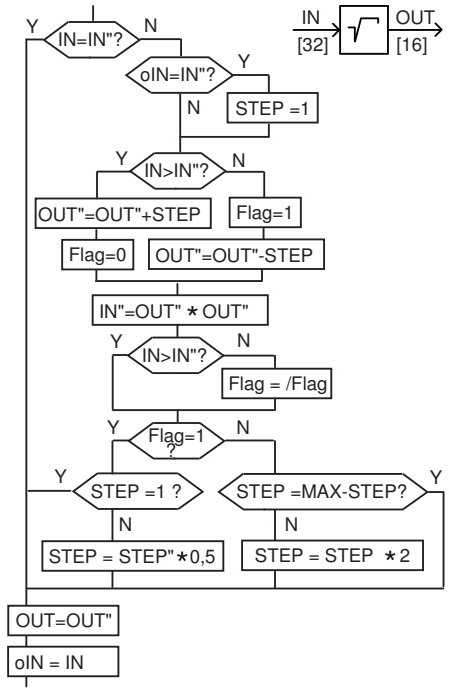


Abbildung 6: Flussdiagramm

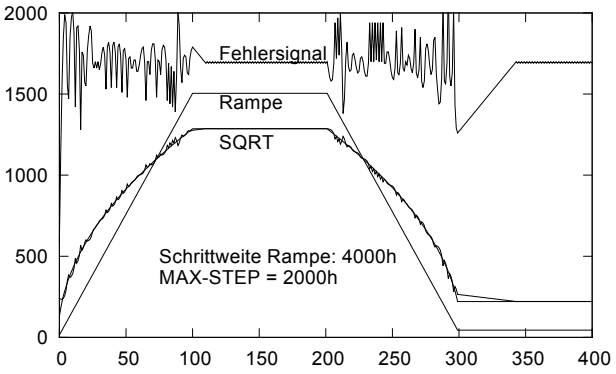


Abbildung 7: Testlauf

SQRT	*Byte*	*msec*
FORTH	248	13
Assembler	275	2,5
Tracking–SQRT		
FORTH	379	0,6
Assembler	364	0,06

Tabelle 1: Timing, Speicher

Literaturverzeichnis

[1] emb(10) Mittelwertfilter `http://www.embeddedforth.de/emb10.pdf`

[2] emb(11) 1pol IIR-Filter `http://www.embeddedforth.de/emb11.pdf`

[3] emb(4) Quadratwurzel `http://www.embeddedforth.de/emb4.pdf`

Listings

```
1   <| \ tracking SQRT
2
3   4 ZVARIABLE oIN   \ unsigned
4   4 ZVARIABLE IN"   \ unsigned
5   2 ZVARIABLE OUT"  \ unsigned
6   2 ZVARIABLE STEP  \ unsigned
7   1 ZVARIABLE FLAG  \ flag
8
9   : INIT   1 STEP ! ;
10
11  : (SQRT) \ ( Flag --- )
12           \ Flag 1 -> Step*0,5 ; 0 -> Step*2
13  STEP @ DUP ROT
14  IF      1 = LNOT IF 1SHIFT> THEN  \*0,5
15  ELSE 2000 = LNOT IF 1<SHIFT THEN  \*2
16  THEN STEP ! ;
17
18  : SQRT  \ ( UD1  --- UN1 ) UD1 =< 7FFF FFFF
19  2DUP IN" D@ D- OR IF.     \ UD1 >< IN ?
20    2DUP oIN D@ D- OR LNOT IF 1 STEP ! THEN
21    2DUP oIN D!
22    2DUP IN" D@ UD>
23      IF                          \ new > old
24      OUT" @ STEP @ + OUT !   0 FLAG C!
25      ELSE                        \ old > new
26      OUT" @ STEP @ - OUT !   1 FLAG C!
27      THEN
28    OUT" @ DUP
29    U*          \( UN1 UN2 --- UD1 )
30    2DUP IN" D! \ --- UD1 new )
31    UD>
32      IF   FLAG C@        \ old > new
33      ELSE FLAG C@ LNOT   \ old <= new
34      THEN (SQRT)
35  ELSE. oIN D!
36  THEN OUT" @ ;
37  |>
```

```
1   <| \ SQRT
2
3   \ sukz. Approx. ohne Multiplikation
4
5   HEX
6   4 ZVARIABLE O#        4 ZVARIABLE ARG
7   TABLE BIT-TAB
8   8000 , 4000 , 2000 , 1000 , 0800 , 0400 , 0200 , 0100 ,
9   0080 , 0040 , 0020 , 0010 , 0008 , 0004 , 0002 , 0001 ,
10
11  : SQRT       \ ( UD1  --- UN1 ) UD1 =< 7FFF FFFF
12  ARG D!    0 0 O# D!    0 0
13  F 0 DO                              \ ( wz ---  )
14  2DUP 1<DSHIFT                       \ ( --- wz 2*A )
15  I 1<SHIFT BIT-TAB + @ 0 D+          \ ( --- wz 2*A+B )
16  F I - 0 DO 1<DSHIFT LOOP 1DSHIFT>   \ ( --- wz B*(2*A+B) )
17  O# D@ D+                            \ ( --- wz A^2+B*(2*A+B) )
18  2DUP ARG D@ UD> LNOT                \ ( --- wz N# )
19  IF O# D!                            \ O# <- N#
20  I 1<SHIFT BIT-TAB + @ 0 D+          \ wz = wz + B
21  ELSE 2DROP
22  THEN
23  LOOP DROP ;
24  |>
```

Gforth auf Android

Bernd Paysan

Gforth auf Android zu portieren, war zunächst leicht, und wurde dann schwer... Zunächst habe ich Gforth als Konsole–Programm unter Kevin Boones KBOX[1] portiert — da hat man weitgehend ein Linux–System zur Verfügung, und lediglich Gforths Unterstützung für C–Crosscompilation musste verbessert werden.

Leider kann man auf der Konsole eben wirklich nur Konsole machen, kein OpenGL, keine Abfrage von Touch–Screen und Sensoren, also keine Chance, irgendwas App–ähnliches zu entwickeln. Ab Android 2.3 gibt es aber „native activities“, bei der man kein Java schreiben muss, sondern einfach nur eine C–Library aufgerufen wird. Um Gforth in dieser Umgebung zum Laufen zu bringen, musste „nur noch“ die Möglichkeit her, Gforth als shared library zu compilieren. Das stand durchaus auch auf dem Plan, denn damit kann man Gforth leicht in andere Programme einbinden.

Gerald Wodnis SWIG–Anbindung ist auch inzwischen einigermaßen gediehen, damit man die nötigen C–Interfaces automatisch einbinden kann. Auch hier gibt es leichte Probleme deshalb, weil man bei der Android–Compilation einen Cross–Compiler verwendet, und nicht einfach mal schnell eben das Generierte ausführen kann — man muss den generierten Zwischencode erst auf das Zielsystem kopieren. Das Ergebnis dieser Bemühungen ist ein apk, das man noch nicht von Google Play herunterladen kann, aber von meiner Homepage: `http://bernd-paysan.de/Gforth.apk`.

Das Gforth packt nach dem ersten Start erst mal seine Sourcen auf die SD–Karte, was etwas dauern kann — SD–Schreibzugriffe sind nicht sonderlich schnell. In dem Zustand kann es noch keine Bildschirmausgaben machen, also einfach Geduld. Wenn es dann fertig ist, startet es ein Terminal (in OpenGL implementiert), und die Menü–Taste blendet die Bildschirmtastatur ein und aus. Mit

```
include gl-sample.fs
```

lädt man eine kleine OpenGL–Demo, aber eigentlich ist das Terminal selbst das kompliziertere OpenGL–Programm. Denn der Text wird über einen Fragment Shader gerendert. Da leider die Qualitätskontrolle bei Android recht schlecht ist, funktioniert je nach Gerät dieser oder jener Befehl der GL Shader Language nicht — wer nur weiß sieht, sollte sich mal mit mir kurzschließen, damit man das debuggen kann.

[1] `http://kevinboone.net/kbox.html`

Es kommt auf die Notation an.

Stephen Pelc

Auf der Forthtagung 2012 im Beukenhof sprach ich aus dem Stegreif zu dem Thema. Michael Kalus drängelte mich dann, das doch als Beitrag für das Forth–Magazin Vierte Dimension *aufzuschreiben. In bester Stegreiftradition gab es natürlich keine Notizen. Aber Michael fertigte eine Abschrift meiner Rede an. So konnte ich den Text überarbeiten, Beispiele und noch etwas mehr Text einfügen. Und da ist er nun.*

Notation: Schreibweise. Die spezielle Art, etwas aufzuschreiben

In Forth benutzen wir inzwischen eine Schreibweise (notation), die sich mit der Zeit entwickelt hat und Konsens ist. Aber sollten wir uns jemals auf eine Notation für OOP einigen können, wäre das ein Wunder; und ich würde halleluja! sagen. (Wörtliche Übersetzung: Lobpreiset Jah!) [Anmerkung der Redaktion: Es ging auf der Forthtagung auch um eine Notation für Objektorientierte Programmierung OOP; Pelcs Rede bezog sich darauf.] Um das hinzukriegen, sollten wir uns einiges über Forth klarmachen, das nichts zu tun hat mit `DUP` oder `VARIABLE`, sondern mit der Art und Weise, wie wir Forth handhaben. Was wir da machen, was Forth ja wirklich ausmacht, ist seine interaktive Erweiterbarkeit. Und das bedeutet, wir können die Notation anpassen auf das, was wir tun. Und das ist die bedeutsamste Eigenschaft in diesem (Programmier–)Sprachmodell.

ForthScript

Würde ich einen Webserver in Forth schreiben, bräuchte ich gar keine eigene Skriptsprache dafür zu erfinden, serverseitig gäbe es dann eine Mixtur aus HTML und Skriptbefehlen. So könnten wir einstellen, dass Forth selbst als seine Skriptsprache benutzt wird:

```
<% language=forthscript %>
```

Die `<%` und `%>` schließen Skriptbefehle ein. Dann könnte ich Forth–Stückchen in die Webseite einfügen, um meine Ausgabe zu erzeugen.

```
<% VAL @ . %>
```

Da ist mein Skript, wir alle wissen, wie das anzuwenden ist, und es kostet mich praktisch nichts, um das zu machen, und es geschieht zur Laufzeit.

Oder eine Definition, um einen COM-Port zu öffnen, könnte so aussehen:

```
COM1  9600 BAUD ... 8N1
```

Und das ist Forth. Das könnten sogar meine Anwender verstehen. Es geht also immer darum, eine Notation passend für die Aufgabe zu erschaffen.

In beiden obigen Beispielen wurden bloß die interaktiven Fähigkeiten zur Laufzeit von Forth benutzt, noch nicht einmal die zur Compilezeit. Oft machen wir ja so etwas: Wir stellen einen Satz an Befehlen bereit, die ein Endbenutzer verstehen muss. Zum Beispiel müssten die Leute, die Webseiten erstellen, die ForthScript enthalten, nur wissen, wie das ForthScript benutzt wird, nicht das ganze Forthsystem darunter. Wir müssen die Notation dafür schaffen (design the notation). Heutzutage haben selbst eingebettete Systeme (embedded systems) genug Speicher, um das zu können.

Die Forth–Gemeinschaft (Forth community) neigt dazu, sich über unendliche Details bei einer Implementation des Systems selbst zu streiten, aber für unsere Anwender ist die Notation wichtig. Denn das ist es, was dann auf dem Papier stehen wird, und auf dem Bildschirm. Wenn wir also über einen Forth–Code reden, lasst uns die Notation zuerst machen, und dann die Implementation.

Code unterhalten und erweitern (maintain and extend)

Wenn die Notation einwandfrei lesbar ist, ist sie auch klar brauchbar, und wir wissen, dass damit dann Anwendungen möglich sind. Nun, ich bin ja extrem befangen, ich bin ein Verkäufer (vendor), heißt, ich mag Geld. Und das Gute daran wiederum ist, dass ich die großen Anwendungen mag, weil sie viel kosten. Es wird sehr viel Code dafür geschrieben, und das bedeutet, dass ich auch noch nach sechs Monaten, oder sechs Jahren, in der Lage sein muss, zu verstehen, was der macht.

Und ihr wisst alle, dass ihr schon Kommentare zu jeder Zeile gemacht habt, und trotzdem verstand man den Code nicht. Also muss der Code für sich selbst sprechen. Wenn man etwas sagt, sollte man verstehen, was du meinst. Du hoffst, dass der, den du anredest, dich auch versteht. Der Instandhaltungsprogrammierer (maintenance programmer) ist diese Person, zu der du sprichst, wenn du den Code schreibst. Und du kennst ihn/sie nicht, weißt nicht, wie clever er/sie sein wird.

Ich habe mit einigen Anwendungen zu tun, die schon über 25 Jahre alt sind. Der Crosscompiler, den MPE verkauft, basiert auf Code, den wir zum ersten Mal 1981 sahen. Nur wenige Zeilen davon blieben unverändert, aber der aktuelle Code ist ein direkter Nachfahre des Originalcodes von 1981. Wir haben sogar Kunden mit Code, der noch älter ist. Diese Leute haben es mit Code zu tun, der nun 30 Jahre alt ist, und der auf einem HP–Desktop–Computer zur Welt kam, als das noch die sexysten Dinger an Computern waren. Und wir unterhalten und erweitern diese Codebasis noch immer.

Was also zählt, ist, dass Code überlebt, und zwar nicht in der Form des Objektcodes. Der Objektcode wechselte vom 68000 zu einem 8086 zu einem 80386 zu einem ARM und zu was noch allem. So alle 10 Jahre wechselte er die CPU und das Betriebssystem. Es ist Standard für so einen Prozess, dass das ganze Zeug als Quellcode auf Disk oder Papier steht, als die beste Form, es aufzubewahren.

Notation am Beispiel Morse–Code

Kehren wir also zurück zu unserem OOP–Paket, oder weil Carsten heute Morgen über Anwendungsfelder für den Morse–Code sprach, zu der Frage: "Wie hast du das denn definiert, dass ich jetzt Morse–Code eingeben kann?"Du könntest Punkte und Striche genommen haben, wobei das Minuszeichen der Strich ist, und dann wüssten wir, dass es Morse–Code darstellen soll. Und das wären druckbare Zeichen aus dem ASCII–Zeichensatz. Also so vielleicht:

```
(Beispiel A)
morse-char A . -
morse-char B - . . .
morse-char C - . - .
```

oder

```
(Beispiel B)
morse-char A .-
morse-char B -...
morse-char C -.-.
```

Das übersetzen wir dann in eine Form, die den Morsecode erzeugt, also Punkte und Striche schreibt oder anderweitig überträgt (Töne). Die Übertragungsseite des Codes enthält dann auch das Wissen um Pausen zwischen den Zeichen und Worten. Aber jetzt sind wir erstmal nur interessiert an der Notation, der Rest sind Details der Implementation. Als Erstes müssen wir daher nur das Wort MORSE-CHAR definieren, damit es ein Morsezeichen in unsere Tabelle einträgt.

Aber das wäre nicht wirklich gut, denn dann müsste man die Morsezeichen in den Quellcode übertragen. Und das ist fehleranfällig und ermüdend. Was wir können möchten, ist doch, existierende Morsezeichentabellen einzulesen. Hier ist so eine Tabelle dafür:

```
http://encyclopedia2.thefreedictionary.com/
Morse+Code+%28table%29

International Morse Code

  A   .-       U   ..-
  B   -...     V   ...-
  C   -.-.     W   .--
  D   -..      X   -..-
  E   .        Y   -.--
               Z   --..
  F   ..-.
  G   --.      0   -----
  H   ....     1   .----
  I   ..       2   ..---
  J   .---     3   ...--
               4   ....-
  K   -.-      5   .....
  L   .-..     6   -....
  M   --       7   --...
  N   -.       8   ---..
  O   ---      9   ----.

  P   .--.     Period      .-.-.-
  Q   --.-     Comma       --..--
  R   .-.      ? Mark      ..--..
  S   ...      Hyphen      -....-
  T   -        Apostrophe  .----.
               Colon       ---...
  U   ..-      Quotation   .-..-.
  V   ...-     Slash       -..-.
  W   .--      @ sign      .--.-.
  X   -..-
  Y   -.--
  Z   --..
```

Um es also besser zu machen als im Beispiel A) oder B), sagen wir, hier ist die Morse–Code–Tabelle. Also benutze ich einige magische Zeichen für eine Notation dafür. So wissen wir, dass dieser einzelne Buchstabe A das da werden soll (er zeigt auf den Morsecode dahinter: .-) Das ist schon eine Notation, und sobald die gefunden ist, wird es eine einfach zu verstehende Tabelle. Wir sehen Paare von Zeichengruppen (pairs of token). Das erste Zeichen ist der Buchstabe, das zweite ist seine Darstellung in Punkten und Strichen. Grad so, wie es das Wort MORSE-CHAR im Beispiel B) handhabte. Und es ist kein Aufwand, die Tabelle anzupassen, indem man ein „Period“ als „ . “ schreibt, und den Fehler der doppelten Angaben von U...Z rausmacht. Mit etwas syntaktischem Zucker kommt man dann also zu einer Morsecodetabelle im Quellcode, die auch später noch ganz einfach gewartet werden kann. Da kann man erproben, ob der Morsecode auch stimmt, und die Tabelle bearbeiten, falls das nicht der Fall sein sollte.

```
create MorseTable  \ -- addr
[morse
  A   .-       U   ..-
  B   -...     V   ...-
  C   -.-.     W   .--
  D   -..      X   -..-
  E   .        Y   -.--
               Z   --..
  F   ..-.
  G   --.      0   -----
  H   ....     1   .----
  I   ..       2   ..---
  J   .---     3   ...--
               4   ....-
  K   -.-      5   .....
  L   .-..     6   -....
  M   --       7   --...
  N   -.       8   ---..
  O   ---      9   ----.
```

```
  P   .--.      .   .-.-.-
  Q   --.-      ,   --..--
  R   .-.       ?   ..--..
  S   ...       -   -....-
  T   -         '   .----.
                :   ---...
                "   .-..-.
                /   -..-.
                @   .--.-.
morse]
```

Zuverlässigen Code schreiben

Eine gute Notation erzeugt Zuverlässigkeit. Stell dir vor, du machst eine Tabelle für 256 Fonts (Schriftsatz). Dann wirst du ein ASCII–Zeichen sehen und reihenweise Bit–an/aus–Definitionen. Und wenn du glaubst, du könntest das fehlerfrei in Hex-Bytes übersetzen, denk lieber noch mal nach. Du siehst das und siehst schon, dass du Fehler machen wirst, wenn das manuell zu Code werden müsste. Die Frage ist also, verbringst du eine halbe Stunde damit, eine Notation dafür zu finden, oder später Stunden, um den Code zu korrigieren?

Das ist die Moral von der Geschichte. Es kommt auf die Notation an. Wir haben es mit Forth zu tun, einer Sprache, in der wir unsere Notation manipulieren können. Wir sollten uns daran erinnern, um genau daraus den Vorteil zu ziehen. (Übersetzung aus dem Englischen: mk)

Links

`http://www.mpeforth.com/`

Morse 5: eine deklarative Version

Erich Wälde

Stephen Pelc hat uns auf der Forth–Jahrestagung 2012 nahegelegt, eine weitere Version des Morseprogramms zu schreiben, welche die Tabelle mit den Morsecodes in einer ganz simplen, deklarativen Schreibweise erzeugt. Forth verfügt über die dazu nötigen Mittel. Hier ein Versuch mit `amforth` *[1].*

Motivation

Um zu verstehen, warum diese deklarative Version der Tabelle erstrebenswert sein könnte, schauen wir uns an, wie die Zeichen in den in [2] und [3] vorgestellten Versionen generiert bzw. aufgehoben werden.

Morse 1

Jedes Morsezeichen wird durch ein separates Wort ausgegeben. Es gibt keine Tabelle und dafür jede Menge Wiederholungen. Das ist einfach zu verstehen, auch wenn die Funktion `morseemit` ein Monster ist und keine kleine, gar elegante Funktion. Fairerweise soll hier erwähnt werden, dass das auch nicht das Ziel dieser Version ist.

```
: _A  kurz lang Zend ;
: _B  lang kurz kurz kurz Zend ;
```

Morse 2

Die *execution token* der Worte aus 1. werden in einer Tabelle gehalten. Das vereinfacht die Funktion `morseemit` enorm. Die Tabelle liegt im RAM und ist nach einem Stromausfall nicht mehr korrekt gefüllt.

```
\ Tabelle zur Kompilierzeit fuellen
' _A char a >mtable
' _B char b >mtable
```

Morse 3

In dieser Version wird ein Binär-Code aus dem Lang(1)-Kurz(0)-Muster zusammen mit der Länge des Zeichens (z.B. a: Kurz Lang, Länge 2) in ein Byte gepackt und dieser in einer Tabelle abgelegt. `morseemit` holt den Code aus der Tabelle und entpackt ihn, `domorse` gibt das Morsezeichen entsprechend der entpackten Information (Länge, Lang-Kurz-Muster) aus. Der Platzbedarf ist mit 1 Byte/Morsezeichen ziemlich unschlagbar. Die Tabelle liegt im RAM, wie bei 2. Ein Nachteil ist, dass die Binärcodes *verkehrt herum* aussehen, weil die Zeichenausgabe mit dem niedrigstwertigen Bit beginnt. Kurz Lang ergibt eben nicht 01, sondern 10. Alles funktioniert richtig, aber es *sieht verkehrt aus.*

```
\ Tabelle zur Kompilierzeit fuellen
binary 00010 decimal 2 pack char a >mtable
binary 00001 decimal 4 pack char b >mtable
```

Morse 4

Wie bei 3 wird die Länge und der Zeichencode in eine Tabelle kopiert, diese existiert jetzt aber im Flash. Die Tabelle im Flash wird zellenweise (nicht byteweise) adressiert. Daher belegt jedes Zeichen 2 Byte in der Tabelle. Die Arbeitsweise ist identisch zu Version 3, jetzt bleibt die Tabelle bei einem Stromausfall erhalten.

```
create mtable
0 ,                 \    $60 offset
```

```
%00010  &2 pack ,  \ a  $61
%00001  &4 pack ,  \ b
```

Wenn man die Programmlistings lange genug liest und verdaut, dann wird man es auch schaffen, zusätzliche Zeichen einzubauen. Vielleicht findet man auch heraus, dass in Version 3 die Zeichen maximal 5 Bit belegen dürfen, und in Version 4 8 Bit (wobei man das noch auf 12 Bit aufbohren könnte). Aber wir wurden zu Recht darauf hingewiesen, dass sich das doch besser machen lässt.

Morse 5

Die folgende Schreibweise wäre hoffentlich um Längen einfacher zu verstehen:

```
morsetable:
    . _      |   a
    _ . . .  |   b
...
;morsetable
```

Lang wird hier direkt mit einem Unterstrich (`_`) und Kurz mit einem Punkt (`.`) notiert. Der senkrechte Strich (`|`) trennt die beiden Tabellenspalten, kann aber wahrscheinlich durch eine andere Implementierung auch noch wegrationalisiert werden. Hier ein neues Zeichen hinzuzufügen, ist in 10 Sekunden erledigt — jedenfalls für den von-links-nach-rechts-lesenden Teil des Planeten, und solange zwischen den Zeichen immer schön ein Leerzeichen steht.

Die nachfolgende Version Morse 5 benutzt folgende Zutaten zur Lösung dieser Aufgabe: 1. Eine spezielle *Wortliste* beinhaltet die Worte `.` und `_` sowie den Spaltentrenner `|`, sowie die notwendigen Hilfsworte und –variablen. Diese Wortliste wird nur aktiviert, um die Tabelle zur compile-Zeit zu verarbeiten. 2. Wir verwenden den in *finite state machine* [4] vorgestellten Trick, ein Stück normalen Programmtext zur compile-Zeit zu verarbeiten, indem man im `create`-Teil den compiler explizit aufruft. Funktioniert so bekanntlich nur mit *indirect threaded* Forths.

Implementierung

Zunächst reservieren wir den Platz für die Tabelle. Die benötigt zwei Byte für jedes Morsezeichen (Länge, Zeichenmuster).

```
variable mtable $ff cells allot
: m.clear  mtable $ff cells 0 fill ;
```

Die Funktion `m.clear` dient der besseren Lesbarkeit; sie wird lediglich einmal beim Erstellen der Tabelle benötigt. Wenn der Eintrag in der Tabelle 0 ist, dann bedeutet das, dass kein Zeichen definiert wurde.

```
include lib/vocabulary.frt
\ new wordlist, used only within
\ morsetable: ... ;morsetable
vocabulary <morse>
also <morse> definitions

variable morse.tmp.length
variable morse.tmp.code
: /morse.tmp
  0 morse.tmp.code !
  0 morse.tmp.length !
;
: length++
  1 morse.tmp.length +!
;
```

Beim Bearbeiten der Zeichentabelle führen wir Buch über die Werte und die Anzahl der eingelesenen Zeichen (Punkt oder Strich). Dazu definieren wir zwei Variablen `morse.tmp.length` und `morse.tmp.code`. Die Funktion `/morse.tmp` löscht die Inhalte dieser Variablen.

Die Interpretation der Zeichen Punkt und Strich bedeutet, dass ich die Funktionen `.` und `_` (oder auch `-`) mit ganz neuen Inhalten versehen (überladen) muss. Daher rührt die Entscheidung, diese neuen Definitionen in einer separaten Wortliste `<morse>` zu verwalten. Die Funktionen `.` und `_` sind simpel: schiebe eine 0 oder 1 in das aktuelle Zeichenmuster (`morse.tmp.code`) und erhöhe die Länge um 1.

Hier muss man sich noch Gedanken über die Bitfolge im Zeichenmuster machen. In der Funktion `morseemit` aus Morse 4 wird zuerst das Zeichen ausgegeben, welches im niedrigsten Bit kodiert ist. Verwendet man zum Aufsammeln der Zeichenfolge unbekümmert ein `1 lshift`, dann dreht sich die Reihenfolge der Bits um: das im höchstwertigen Bit kodierte Zeichen muss jetzt zuerst ausgegeben werden. Mindestens folgende Möglichkeiten stehen zur Verfügung:

1. die Funktion `morseemit` so ändern, dass die Ausgabe mit dem höchsten Bit (entsprechend der Länge) beginnt.
2. die aufgesammelten Bits an ihre Position kodieren, also etwa `code @ 1 length @ lshift or code !`, um einen Strich an der Position `length` zu kodieren.
3. die Zeichen mit `rshift` statt `lshift` zusammensammeln und beim Packen auf die korrekte Position verschieben.

Ich habe probehalber alle drei Varianten programmiert und mich dann für Variante 2 entschieden, weil der resultierende Programmtext der einfachste ist. Für einen Strich muss das Bit an Position `morse.tmp.length` auf 1 gesetzt werden. Für einen Punkt muss man im Muster gar nichts mehr tun, weil an der Bitposition bereits eine 0 steht.

```
: _
  morse.tmp.code @
  \ place 1 at position length
  1 morse.tmp.length @ lshift  or
  morse.tmp.code !
  length++ ;
: .
  length++
;
: |
  \ produce packed value
  morse.tmp.length @ >< morse.tmp.code @ or
  /morse.tmp
```

```
  char \ eat next char in table definition
  cells mtable + !
;
\ end vocabulary <morse>
previous definitions
```

Um das Ende des Morsezeichens in der Definition zu sehen, habe ich mich entschieden, einen Spaltentrenner zu benutzen. Das Wort `|` packt die aktuellen Werte der Variablen `morse.tmp.code` und `morse.tmp.length` zusammen und löscht deren Inhalt anschließend. Dann konsumiert es das Zeichen in der zweiten Spalte und berechnet daraus den Index in der Tabelle. Der gepackte Wert wird an diesem Index in der Tabelle gespeichert. Jetzt ist man in der Lage, die Tabelle selbst zu definieren. Am Anfang müssen Tabelle und Hilfsvariablen gelöscht und die Wortliste `<morse>` aktiviert werden. Der unauffällige Aufruf von `[` sorgt dafür, dass der Text zwischen `morsetable:` und `;morsetable` sofort (und nicht erst zur Laufzeit) ausgewertet wird. Am Ende der Tabelle wird die zusätzliche Wortliste wieder deaktiviert.

```
: morsetable:
  [ also <morse> ]
  m.clear
  also <morse>
  /morse.tmp
  [ previous ]
  [         \ NOT PORTABLE!
            \ indirect threaded forth only!
;
: ;morsetable
  previous \ drop <morse>
;
```

Solchermaßen gewappnet wird die Tabelle tatsächlich erzeugt. Dabei ist es völlig unerheblich, in welcher Reihenfolge die Zeichen generiert werden, es ist auch möglich, mehr als eine Definition in eine Zeile zu schreiben.

```
morsetable:
    . _     |  a        _ .     |  n
    _ . . .  |  b        _ _ _   |  o
    _ . _ .  |  c        . _ _ .  |  p
    _ . .    |  d        _ _ . _  |  q
    .        |  e        . _ .   |  r
    . . _ .  |  f        . . .   |  s
    _ _ .    |  g        _       |  t
    . . . .  |  h        . . _   |  u
    . .      |  i        . . . _  |  v
    . _ _ _  |  j        . _ _   |  w
    _ . _    |  k        _ . . _  |  x
    . _ . .  |  l        _ . _ _  |  y
    _ _      |  m        _ _ . .  |  z
;morsetable
```

In dieser Form neue Zeichen hinzuzufügen, ist jetzt wahrscheinlich ein Kinderspiel. Allerdings haben wir die Arbeit trickreich in andere Worte gelegt, deren Lesbarkeit für einen Anfänger höchstwahrscheinlich zweifelhaft ist.

Die restlichen Funktionen sind identisch mit denen von Morse 4 und sind hier nur der Vollständigkeit halber angegeben.

```
: unpack  dup $00ff and swap >< $00ff and ;
\ shift code --- lsb first
: domorse ( code length -- )
  0 ?do  dup  1 and  if
      lang
    else
      kurz
    then
    2/
  loop
  drop
  Zend
;
variable o-emit   ' emit defer@ o-emit !
: morseemit
  $00ff and
  dup o-emit @ execute
  dup bl = if
    Wend
    drop
    exit
  then
  \ tr [A-Z] [a-z]
  dup $40 > over $5b < and if
    $20 +
  then
  cells mtable + @
  unpack domorse
;
: morse
  ['] emit defer@ o-emit !
  ['] morseemit is emit
;
: endmorse
  o-emit @ is emit
;
```

Verfügt man über einen Arduino Duemilanove und ein Danger Shield, oder verändert man `lang` und `kurz` so, dass man die Ausgabe der Zeichen anderweitig sehen kann, dann lässt sich die Morsetabelle auf folgende Arten benutzen:

```
init
piepser
char s morseemit
morse ." bla" type endmorse
: msg s" Geblubber" itype ;
morse msg endmorse
```

Tabelle in Flash–Speicher ablegen

Die im RAM abgelegte Tabelle ist natürlich nicht persistent und macht das ganze Projekt unbefriedigend. Nachdem die Tabelle im RAM erstellt wurde, könnte man sie in den Flash–Speicher übertragen. Extrabonus gibt es, wenn man anschließend den Platz im RAM wieder freigibt. Also vielleicht gleich die Tabelle ins flash schreiben? Die Eigenheiten vom Flash–Speicher legen allerdings nahe, dass man die Tabelle mit `$ffff` (`-1`) initialisiert, und nicht mit Nullen. Das erfordert eine zusätzliche Abfrage in `morseemit`.

Statt der Tabelle `mtable` im RAM reservieren wir Platz im Flash durch Verschieben des Zeigers `dp`. Anschließend wird der reservierte Platz initialisiert.

```
: fl.erase ( addr n -- )
  0 ?do $ffff over i + !i loop drop
;
: fl.allot ( length -- )
  dp swap       \ -- dp L
  over over     \ -- dp L dp L
  + to dp       \ -- dp L
  fl.erase
;
create mtable $ff fl.allot
```

Dabei ist zu bedenken, dass der Flash–Speicher vermittelst `!i` und `@i` schon zellenweise (nicht byteweise) bearbeitet wird. Entsprechend ist das Wort `cells` in den Worten `|` und `morseemit` zu löschen.

```
: |
...
  \ use as offset and store
  mtable + !i
;
```

Aus `morsetable:` verschwindet der Aufruf von `m.clear`.

Zuletzt ändert sich die Anweisung in `morseemit`, welche das gepackte Zeichen aus der Tabelle holt. Hier kommt auch die zusätzliche Abfrage, ob der Wert aus der Tabelle nicht `$ffff` ist, was einem nicht definierten Zeichen entspricht.

```
: morseemit
...
  mtable + @i
  dup $ffff <> if
    unpack domorse
  else
    drop
  then
;
```

Alles andere funktioniert, wie gehabt. Eigentlich erstaunlich, dass eine recht grundsätzliche Änderung des Programms nur kleine Unterschiede benötigt. Und weil wir so fleißig waren, bekommen wir noch Ziffern und Satzzeichen in die Tabelle. Es gäbe zwar noch Umlaute und die *Ligatur* `ch` zu vergeben, aber mit solchen Seltenheiten will ich mich hier nicht plagen.

```
morsetable:
...
    . _ _ _ _  | 1
    . . _ _ _  | 2
    . . . _ _  | 3
    . . . . _  | 4
    . . . . .  | 5
    _ . . . .  | 6
    _ _ . . .  | 7
    _ _ _ . .  | 8
    _ _ _ _ .  | 9
    _ _ _ _ _  | 0

    . _ . _ . _ | .   \ Punkt
    _ _ . . _ _ | ,   \ Komma
    _ _ _ . . . | :   \ Doppelpunkt
    _ . . . . _ | -   \ Bindestrich
    . _ _ _ _ . | '   \ Apostroph
    . . _ _ . . | ?   \ Fragezeichen
    _ . _ _ . _ | (   \ Klammer
    _ . _ _ . _ | )   \ Klammer

;morsetable
```

So Sachen

1. Und nochmal zur Wiederholung: die hier gezeigte Lösung funktioniert nur mit einem *indirect–threaded* Forth.

2. Das vorgestellte Programm wurde auf amforth in den Versionen 4.6 und 4.9 entwickelt und getestet.

3. Wenn man die Leerzeichen in der Definition der Punkt-Strich-Zeichen wegrationalisieren will, dann kann man einen Recognizer schreiben, der beim Einlesen der Tabelle zusätzlich aktiviert wird. Vielleicht gibt's also eine weitere Fortsetzung.

4. Mir scheint, dass das Morseprogramm hinreichend komplex ist, um allerhand Sprachkonzepte zu bemühen und dem Forth–Anfänger hoffentlich auch schmackhaft zu machen. Mein Dank geht, wie immer, an die Teilnehmer des mittwöchlichen IRC–Chats für Ideen, Hinweise und die Aufmunterung, dass das doch auch besser geht. Wer ernstlich Morsen lernen will, der sollte sich die Seite zu Kochs Methode ansehen [5].

Referenzen

1. `http://amforth.sourceforge.net/`
2. Wälde et al., Fort*h*bildung, VD 2011/04 S. 6ff
3. Wälde, Nachtrag: Morse 4 repariert, VD 2012/01, S. 30ff
4. Wälde, Adventures 11: Finite State Machine, VD 2012/02, S. 6ff
5. `http://www.df2ok.privat.t-online.de/afu01a.htm`

Listings

base.fs

```
1   \ --- morse/base.fs -------------------------
2   \ 2011-10-26  EW
3   \ 2011-11-02  CS
4   \ arduino duemilanove + danger shield
5   \ morse code stuff
6   \ (Potsdam/Augsburg/Oberhausen)
7
8   \ make marker loads:
9   \   lib/misc.frt
```

```
\   lib/bitnames.frt
\   lib/ans94/marker.frt
\   .../devices/atmega328p/atmega328p.frt

marker --base--
decimal

PORTB 2 portpin: sw1
PORTB 3 portpin: sw2
PORTB 4 portpin: sw3

PORTD 5 portpin: led1
PORTD 6 portpin: led2

PORTD 3 portpin: bz

PORTC 2 portpin: sl1
PORTC 1 portpin: sl2
PORTC 0 portpin: sl3

PORTC 3 portpin: photocell
PORTC 4 portpin: thermometer
PORTC 5 portpin: knocksensor

PORTD 4 portpin: sr_in
PORTD 7 portpin: sr_oe \ output enable
PORTB 0 portpin: sr_cl

\ Piepser
\ 2 ms T_period =^= 500 Hz
: buzz ( cycles -- )
  0 ?do bz low 1ms bz high 1ms loop
;
: gap  ( cycles -- )
  0 ?do bz high 1ms bz high 1ms loop
;
: blink ( cycles -- )
  led1 high
  0 ?do  1ms 1ms 1ms loop
  led1 low
;

Edefer transmit
: piepser   ['] buzz  is transmit ;
: blinker   ['] blink is transmit ;

decimal
: kurz    50 transmit 50 gap ;
: lang   150 transmit 50 gap ;
: Zend   100 gap ;   \ Pause zwischen Zeichen
: Wend   300 gap ;   \ Pause zwischen Worten

: init
  led1 pin_output led1 low
  led2 pin_output led2 low
  bz   pin_output

  piepser
;
```

Morse 5.1a: Tabelle im RAM

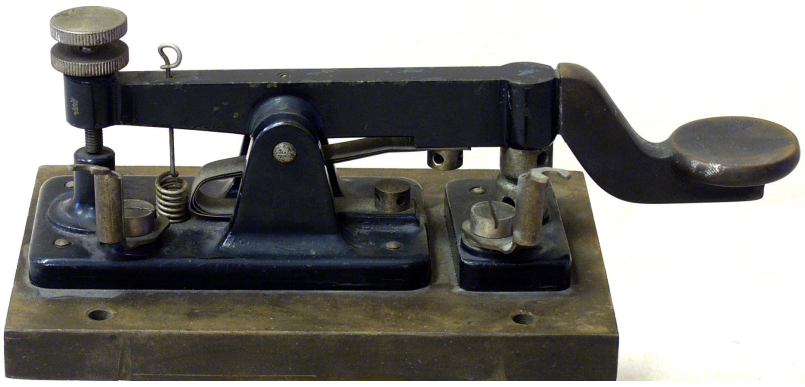
```
\ 2012-08-21  ew  morse5_1a.fs
\
\ transmitting morse codes
\ table of codes in declarative format

include base.fs

marker --morse--
include lib/vocabulary.frt

variable mtable $ff cells allot
: m.clear  mtable $ff cells 0 fill ;

\ new wordlist, to be used only within
\ morsetable: ... ;morsetable
vocabulary <morse>
also <morse> definitions

variable morse.tmp.length
variable morse.tmp.code
: /morse.tmp
  0 morse.tmp.code !
  0 morse.tmp.length !
;
: length++
  1 morse.tmp.length +!
;
: _
  morse.tmp.code @
  \ place 1 at position length
  1 morse.tmp.length @ lshift  or
  morse.tmp.code !
  length++ ;
: .
  length++
;
: |
  \ produce packed value
  morse.tmp.length @ >< morse.tmp.code @ or
  \ reset tmp vars
  /morse.tmp
  \ eat next char in table definition
  char
  \ use as offset and store
  cells mtable + !
;
\ end vocabulary <morse>
previous definitions

: morsetable:
  [ also <morse> ]
  m.clear
  also <morse>
  /morse.tmp
  [ previous ]
  [        \ NOT PORTABLE!
           \ indirect threaded forth only!
;
: ;morsetable
  previous \ drop <morse>
;

morsetable:
    . _      |  a        _ .      |  n
    _ . . .  |  b        _ _ _    |  o
    _ . _ .  |  c        . _ _ .  |  p
    _ . .    |  d        _ _ . _  |  q
    .        |  e        . _ .    |  r
    . . _ .  |  f        . . .    |  s
    _ _ .    |  g        _        |  t
    . . . .  |  h        . . _    |  u
```

```
72        . .       |   i         . . . _   |   v
73        . _ _ _   |   j         . _ _     |   w
74        _ . _     |   k         _ . . _   |   x
75        . _ . .   |   l         _ . _ _   |   y
76        _ _       |   m         _ _ . .   |   z
77    ;morsetable
78
79    : unpack  dup $00ff and swap >< $00ff and ;
80    : domorse ( code length -- )
81      0 ?do  dup  1 and  if
82          lang
83        else
84          kurz
85        then
86        2/
87      loop
88      drop
89      Zend
90    ;
91    variable o-emit
92    ' emit defer@ o-emit !
93    : morseemit
94      $00ff and
95      dup o-emit @ execute
96      dup bl = if
97        Wend
98        drop
99        exit
100     then
101     \ tr [A-Z] [a-z]
102     dup $40 $5b within if
103       $20 +
104     then
105     cells mtable + @
106     unpack domorse
107   ;
108   : morse
109     ['] emit defer@ o-emit !
110     ['] morseemit is emit
111   ;
112   : endmorse
113     o-emit @ is emit
114   ;
115
116   \ init
117   \ piepser
118   \ char s morseemit
119   \ morse ." bla" type endmorse
120   : msg s" ab ab ab" itype ;
121   \ morse msg endmorse
122
123   : test
124     init piepser lang &500 ms
125     morse msg endmorse
126   ;
127   \ fin
```

Morse 5.1b: Tabelle im flash

```
1   \ 2012-08-21  ew  morse5_1d.fs
2   \
3   \ transmitting morse codes
4   \ table of codes in declarative format
5   \ create table directly in flash
6   \ use $ffff as "undef" default
7   \
8   \ \ included by "make marker":
9   \ include lib/misc.frt
10  \ include lib/bitnames.frt
11  \ include lib/ans94/marker.frt
12  \ include .../atmega328p.frt
13
14  include base.fs
15
16  marker --morse--
17  include lib/vocabulary.frt
18
19  : fl.erase ( addr n -- )
20    0 ?do $ffff over i + !i loop drop
21  ;
22  : fl.allot ( length -- )
23    dp + to dp
24  ;
25
26  create mtable $ff fl.allot
27  mtable $ff fl.erase
28
29  \ new wordlist, to be used only within
30  \ morsetable: ... ;morsetable
31  vocabulary <morse>
32  also <morse> definitions
33
34  variable morse.tmp.length
35  variable morse.tmp.code
36  : /m
37    0 morse.tmp.code !
38    0 morse.tmp.length !
39  ;
40  : length++
41    1 morse.tmp.length +!
42  ;
43  : _
44    morse.tmp.code @
45    \ place 1 at position length
46    1 morse.tmp.length @ lshift  or
47    morse.tmp.code !
48    length++ ;
49  : .
50    length++
51  ;
52  : |
53    \ produce packed value
54    morse.tmp.length @ >< morse.tmp.code @ or
55    \ reset tmp vars
56    /m
57    \ eat next char in table definition
58    char
59    \ use as offset and store
60    mtable + !i
61
62  ;
63  \ end vocabulary <morse>
64  previous definitions
65
66  : morsetable:
67    [ also <morse> ]
68    also <morse>
69    /m
70    [ previous ]
71    [         \ NOT PORTABLE!
72              \ indirect threaded forth only!
73  ;
74  : ;morsetable
```

```
  previous \ drop <morse>
;

morsetable:
    . _      |   a        _ .      |   n
    _ . . .  |   b        _ _ _    |   o
    _ . _ .  |   c        . _ _ .  |   p
    _ . .    |   d        _ _ . _  |   q
    .        |   e        . _ .    |   r
    . . _ .  |   f        . . .    |   s
    _ _ .    |   g        _        |   t
    . . . .  |   h        . . _    |   u
    . .      |   i        . . . _  |   v
    . _ _ _  |   j        . _ _    |   w
    _ . _    |   k        _ . . _  |   x
    . _ . .  |   l        _ . _ _  |   y
    _ _      |   m        _ _ . .  |   z

    . _ _ _ _  | 1
    . . _ _ _  | 2
    . . . _ _  | 3
    . . . . _  | 4
    . . . . .  | 5
    _ . . . .  | 6
    _ _ . . .  | 7
    _ _ _ . .  | 8
    _ _ _ _ .  | 9
    _ _ _ _ _  | 0

    . _ . _ . _ | .   \ Punkt
    _ _ . . _ _ | ,   \ Komma
    _ _ _ . . . | :   \ Doppelpunkt
    _ . . . . _ | -   \ Bindestrich
    . _ _ _ _ . | '   \ Apostroph
    . . _ _ . . | ?   \ Fragezeichen
    _ . _ _ . _ | (   \ Klammer
    _ . _ _ . _ | )   \ Klammer
;morsetable

: unpack  dup $00ff and swap >< $00ff and ;
: domorse ( code length -- )
  0 ?do  dup  1 and  if
      lang
    else
      kurz
    then
    2/
  loop
  drop
  Zend
;

variable o-emit
' emit defer@ o-emit !
: morseemit
  $00ff and
  dup o-emit @ execute
  dup bl = if
    Wend
    drop
    exit
  then
  \ tr [A-Z] [a-z]
\  dup $40 > over $5b < and if
  dup $40 $5a within if
    $20 +
  then
  mtable + @i
  dup $ffff <> if
    unpack domorse
  else
    drop
  then
;
: morse
  ['] emit defer@ o-emit !
  ['] morseemit is emit
;
: endmorse
  o-emit @ is emit
;

\ init
\ piepser
\ char s morseemit
\ morse ." bla" type endmorse
: msg s" ab ab ab" itype ;
\ morse msg endmorse

: test
  init piepser lang &500 ms
  morse msg endmorse
  morse s" 1 2 3 schnell herbei" itype endmorse cr
  morse s" a: b. c," itype endmorse cr
;
\ fin
```



Morsetaste (Wikimedia Commons: Hans Grobe)

Geburtstagsfragen (2)

Hannes Teich

Wie feiern eigentlich Leute, die am 29. Februar geboren wurden, Geburtstag? Kein Problem: Man feiert einen Tag nach dem 28. Februar. Aber ist das auch spitzfindig-korrekt? Und wie korrekt feiert der Rest der Menschheit?

Die Frage lässt sich im Prinzip recht einfach beantworten: Man geht von der Geburtsstunde (besser: -minute) aus und addiert eine Anzahl tropischer Jahre. Ein tropisches Jahr, auch „Sonnenjahr“ genannt, ist die Zeitspanne von einer Frühlings-Tagundnachtgleiche bis zur nächsten, und damit ist sie das rechte Maß für den Fortgang der Jahreszeiten.

Das (nicht ganz konstante) Sonnenjahr–Jahr dauert in etwa 365,2423 Tage, während die gregorianische Schalttag-Regel – über 400 Jahre gemittelt – auf (365*400+25*4-3)/400 = 365,2425 Tage kommt. Wenn wir die beiden mit 86400 [Sekunden pro Tag] multiplizieren, ergibt sich für Gregor ein Überhang von rund 17 Sekunden pro Tag. Irgendwann in ferner Zukunft ist ein zusätzlicher Schalttag fällig.

Um zu ermitteln, auf welches bürgerliche Datum – von der Geburtsstunde an gezählt – die Zeitspanne einer bestimmten Anzahl Sonnenjahre trifft, brauchen wir eine Umrechnung des bürgerlichen Datums in eine fortlaufende Tageszählung und auch wieder zurück. Eine unter Astronomen gängige Tageszählung, „Julian Day“ (JD) genannt, hat Joseph Justus Scaliger im Jahre 1583 vorgeschlagen. Diese Zählung gibt die Zeit in Tagen und Tagesbruchteilen an, die seit dem 1. Januar 4713 v. Chr. vergangen ist, und daraus lassen sich Datum, Uhrzeit und Wochentag rückgewinnen.

Um leichter rechnen zu können, ist eine Zählung ab 1. Januar des Jahres Eins vorgeschlagen worden. Ich finde allerdings den Start am 3. März des Jahres 1 v. Chr. praktischer, denn: Mit diesem Tag, der nach Gregorianischem Kalender auf den 1. März des Jahres Null fällt, beginnt ein 400-Jahres-Zyklus, nach dem sich die Schalttag-Regel wiederholt. Und mit einem virtuellen Jahr, das mit März beginnt, kommt niemals ein Schalttag in die Quere, denn der ergibt sich am Jahresende durch den errechneten Start des neuen Jahres.

Für die Beantwortung der oben aufgeworfenen Frage eignet sich das folgende Gforth-Programm. Zusätzlich kann man die Zeitspanne zwischen zwei Datums-Eingaben ermitteln. Und man kann sogar erfahren, wann die Tochter oder Enkelin tausend Wochen alt sein wird.

Ein konkretes Beispiel

Welche Geburtstage (in einer Zeitspanne von 10 Jahren) ergeben sich für einen am 29. Februar 1984 um 12 Uhr mittags Geborenen? **Abb. 1** zeigt das Bildschirm-Protokoll. Man sieht: Die jährliche Wiederkehr der hier angenommenen Geburtsminute trifft auf drei verschiedene Kalenderdaten. Es ist leicht einzusehen, dass da andere Geburtsdaten auch nicht unbehelligt bleiben.

```
Eingabe Datum mit Uhrzeit:
        Jahr = [2012] 1984
       Monat = [7] 2
         Tag = [26] 29
      Stunde = [11] 12
      Minute = [15] 0
   Eingegeben: 1984-02-29 12:00
    Tageszahl: TZ=724640.5  JD=2445760  Mi

          "a" Eingabe eines zweiten Datums
          "b" Eingabe einer Anzahl Tage
          "c" Eingabe einer Anzahl Sonnenjahre
          "q" quit to prompt
  Wahl [a]: c

 Sonnenjahre = [1] 10
Schrittgröße = [10] 1
            0  1984-02-29 12:00  Mi
            1  1985-02-28 17:49  Do
            2  1986-02-28 23:38  Fr
            3  1987-03-01 05:27  So
            4  1988-02-29 11:16  Mo
            5  1989-02-28 17:05  Di
            6  1990-02-28 22:53  Mi
            7  1991-03-01 04:42  Fr
            8  1992-02-29 10:31  Sa
            9  1993-02-28 16:20  So
           10  1994-02-28 22:09  Mo
```

Abbildung 1: Bildschirmprotokoll für 10 Geburtstage (Ausschnitt).

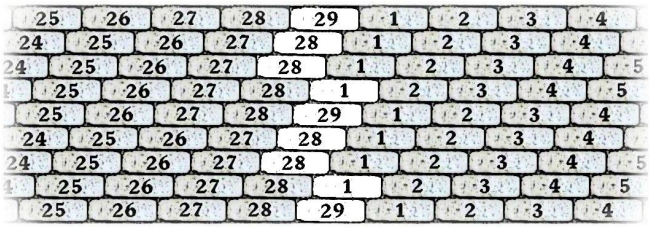


Abbildung 2: Versuch einer Visualisierung für 8 Geburtstage (Ausschnitt).

Wenn der Vorschlag für die Schrittgröße (= Anzahl der Sonnenjahre) mittels Return übernommen wird, gibt es nur zwei Zeilen: Anfang und Ende. Und wenn 10 Sonnenjahre mit einer Schrittweite von 3 gewählt werden, gibt es 5 Zeilen: 0, 3, 6, 9, 12. Das funktioniert auch rückwärts.

Handhabung des Programms

Gibt man nach dem ersten Datum über „a“ ein zweites ein, so wird (gleiche Zeitzone vorausgesetzt) die dazwischenliegende Zeitspanne in Tagen, Stunden, Minuten und zusätzlich in Sonnenjahren angezeigt. Ein Tages-Offset über „b“ liefert ein neues Datum; mit 7000 sind

das die erwähnten tausend Wochen. Und „q“ bringt ein „ok“, worauf im Programm experimentiert werden kann.

Zum Programm selbst ist nicht viel zu sagen. Für die Differenz zwischen zwei Daten konnte auf Floating Point verzichtet werden, nicht aber für die Wandlung in Tagesbruchteile und zurück. (Im Prinzip wäre auch das möglich, versteht sich.) Zur Vermeidung von Rundungsfehlern steckt die Floskel `.5e f+ f>s` in der Routine `TZ>Date`.

In `TN>Jhr&Tage` sind mit `IF ... THEN` zwei Korrekturen enthalten, die durch unterdrückte Schalttage (in jedem Normaljahr und in manchen Hundertern) erforderlich werden. Dazu folgende Überlegung: 1461 Tage sind 4 Jahre. Eine Division mit `1461 /mod` ergibt `0 1`, das heißt: 1 mal 4 Jahre, Rest 0. Nun ein Tag weniger: 1460 ergibt 0 mal 4 Jahre, Rest 1460. Dieser Rest mit `365 /mod` bearbeitet ergibt `0 4`, das heißt: 4 Jahre ohne Rest, also dasselbe wie 1461 und somit ein Problem. Die Lösung lautet: 3 Jahre mit einem Rest von 365 Tagen.

Routine `getIdem` verwendet lokale Values nach Gforth-Art, die einfache für eine Variablen-Adresse, die doppelte für eine Zeichenkette.

Beim Programmstart hat man die Wahl zwischen drei Modi: Alles gregorianisch; alles julianisch; oder hybrid (historisch), mit automatischer Umschaltung zwischen julianisch und gregorianisch anno 1582.

Erwähnenswert ist noch die Erkennung der Monatslängen in den Routinen `Mon&Tag>Tage` und `Tage>Mon&Tag`. Da wird mit 306 multipliziert bzw. dividiert, und das ist ein hübscher Trick (nicht von mir): Addiert man zu 31 mehrmals 30,6 als Monatslänge und verwertet man nur den Integer-Teil, so ergibt sich die Reihe: 31, 61, 92, 122, 153, 184, 214, 245, 275, 306, 337 – und das sind die Monatsanfänge von April bis Februar, wenn der 1. März als Tag Null gilt.

```
Berechnung der Differenz zweier Kalenderdaten oder
eines zweiten Kalenderdatums aus der Differenz. Die
Differenz kann auch als Vielfaches des astronomischen
Sonnenjahres (tropischen Jahres) eingegeben werden.

Datum & Uhrzeit bei Programmstart: 2012-07-26 11:06

           "j" Julianischer Kalender
           "g" Gregorianischer Kalender
           "h" Historisch: Gregorianisch ab 1582-10-15
           "q" quit to prompt
  Wahl [g]: j
               Julianischer Kalender gewählt

Eingabe Datum mit Uhrzeit:
         Jahr = [2012]
        Monat = [7]
          Tag = [13]
       Stunde = [11]
       Minute = [6]
   Eingegeben: 2012-07-13 11:06
    Tageszahl: TZ=735015.4625  JD=2456134.9625  Do
```

Abbildung 3: Programmstart und Wahl des Julianischen Kalenders. Der Unterschied zwischen beiden Kalendern beträgt hier 13 Tage.

Listing

```
1   \ ------------------------------------------------------------------------
2   \  K a l e n d e r - R o u t i n e n  (Gforth 0.7.0)   30jul12 17:30 -jgt
3   \ ------------------------------------------------------------------------
4
5     cr cr ."  Berechnung der Differenz zweier Kalenderdaten oder"
6        cr ."  eines zweiten Kalenderdatums aus der Differenz. Die"
7        cr ."  Differenz kann auch als Vielfaches des astronomischen"
8        cr ."  Sonnenjahres (tropischen Jahres) eingegeben werden."
9
10  \ Zur Differenzbildung wird eine fortlaufende Tageszählung verwendet,
11  \ die nach Julianischem Kalender am 3. März im Jahr 1 unserer
12  \ Zeitrechnung mit Tag Null beginnt. Das ist nach proleptischem
13  \ (= vorgezogenem) Gregorianischem Kalender (der vor 1582 keine
14  \ historische Bedeutung hat) der 1. März des Jahres Null.
15  \
16  \ Mit diesem Tag beginnt ein 400-Jahres-Zyklus. Durch Beginn des
17  \ (virtuellen) Jahres mit März ergibt sich ein Schalttag am Jahresende
18  \ durch den Start des neuen Jahres. Die Monate März bis Dezember
19  \ behalten ihre Nummerierung, Januar und Februar gelten als 13. und
20  \ 14. Monat des Vorjahres.
21  \
22  \ Neben der intern verwendeten Tageszahl (TZ) wird der gebräuchliche
23  \ "Julian Day" (JD, nach Scalinger) angezeigt. Er beginnt mit dem
24  \ 1. Januar 4713 v.Chr., in seiner historischen Variante um 0 Uhr,
25  \ astronomisch jedoch um 12 Uhr mittags, was nach Gregorianischem
26  \ Kalender -4713-11-24 12:00 (UT) entspricht. Um aus TZ den astrono-
27  \ mischen JD zu erhalten, wird 1721119,5 addiert.
28  \
29  \ Bei Programmstart wird der Kalender gewählt: gregorianisch, julianisch
30  \ oder hybrid (historisch). Letzteres soll heißen, dass ein Eingabedatum
31  \ vor 1582-10-15 automatisch julianisch interpretiert wird. Historisch
32  \ folgte auf 1582-10-04 (jul.) unmittelbar 1582-10-15 (greg.).
33  \
34  \ Sodann wird in der Hauptschleife ein Eingabedatum angefordert und
35  \ die gewünschte Funktion gewählt: (a) Eingabe eines zweiten Datums
```

```
\ und Anzeige der dazwischenliegenden Tage, Stunden und Minuten;
\ (b) Eingabe eines Offsets in Tagen und Ausgabe des resultierenden
\ Datums; (c) Eingabe einer Anzahl Sonnenjahre und tabellarische
\ Ausgabe der dazwischenliegenden Daten zur jeweils gleichen Jahres-
\ zeit ("Wahrer Geburtstag"). Zu beachten: Bei (b) und (c) wird der
\ Kalender des Ausgangstages beibehalten, auch wenn der historische
\ Kalendersprung in die Quere kommen sollte.
\
\ Die Länge des Sonnenjahres ist leichten Veränderungen unterworfen.
\ Zu Beginn des Jahres 2000 betrug sie 365,2421905 Tage, gegenwärtig
\ liegt sie bei 365,242375 Tagen. 365,2423 dürfte ein für unsere Zwecke
\ brauchbarer Wert sein. (Das Gregorianische Jahr dauert - über 400
\ Jahre gemittelt - 365,2425 Tage.)

\ ------------------------------------------------------------------------
\       K O N S T A N T E N   U N D   V A R I A B L E N
\ ------------------------------------------------------------------------
        fvariable  TZ1      fvariable  TZ2      \ Tageszähler
        variable   wahl                         \ Funktionswahl
        variable   offs     1 offs !            \ Datums-Offset
        variable   trops    1 trops !           \ Anzahl Sonnenjahre
        variable   stepWid                      \ Tabellen-Schrittweite
        variable   mode                         \ Kalenderwahl
        variable   jflag                        \ julianisches Flag
        variable   func                         \ Funktionsvariable
        variable   temp                         \ Kurzzeitvariable

        365.2423e  fconstant  ftrop             \ tropisches Jahr

        s"          Jahr = "  2constant  Jhr$
        s"         Monat = "  2constant  Mon$
        s"           Tag = "  2constant  Tag$
        s"        Stunde = "  2constant  Std$
        s"        Minute = "  2constant  Min$

\ Datumsvariablen aktuell vorbesetzen. (Bei Verwendung des Julianischen
\ Kalenders bis in die Neuzeit wird das aktuelle Datum umgerechnet.)
   time&date  ( sec min hour day month year)
       dup  variable Jhr1  Jhr1 !   variable Jhr2  Jhr2 !
       dup  variable Mon1  Mon1 !   variable Mon2  Mon2 !
       dup  variable Tag1  Tag1 !   variable Tag2  Tag2 !
       dup  variable Std1  Std1 !   variable Std2  Std2 !
       dup  variable Min1  Min1 !   variable Min2  Min2 !  ( sec) drop

\ ------------------------------------------------------------------------
\       H I L F S F U N K T I O N E N
\ ------------------------------------------------------------------------
\ In einem Feld mit p Positionen wird die doppelt-genaue Zahl d
\ mindestens #-stellig (ggf. mit führenden Nullen) rechtsbündig ausgeben.
: d.r#  ( d # p --)  >r 1- >r tuck dabs
        <<# r> 0 do # loop #s rot sign #>
        r> over - spaces type #>> ;

\ Die einfach-genaue Zahl n wird mindestens #-stellig (ggf. mit
\ führenden Nullen) ausgeben, ohne Blank am Ende.
: .#  ( n # --)  swap s>d rot  0 d.r# ;

\ Division mit Aufrunden.
: /up  ( n1 n2 -- n3)  negate / negate ;

\ Integer-Funktion: r2 ist der ganzzahlige Anteil von r1.
: fint  ( r1 -- r2)  f>d d>f ;

\ Wie »f.«, aber mit unterdrücktem Dezimalpunkt im Integer-Fall.
: f.'  ( r --)  fdup fdup fint f= IF f>d d. ELSE f. THEN ;

\ Ausgabe einer Zeichenkette, eingeschlossen in Anführungszeichen.
\ Beispiel für einen Aufruf: s" Hallo!" type"
: type"  ( stringvar --)  [char] " emit  type  [char] " emit ;

\ Mehrfach-Drop vom Daten-Stack.
: drops  ( n xx --)  0 DO drop LOOP ;

\ Kürzel für Kalenderwahl.
: juln  ( --)  -1 jflag ! ;             \ julianischer Kalender
: greg  ( --)   0 jflag ! ;             \ gregorianischer Kalender
```

```
\ ------------------------------------------------------------------------
\       T A G E S Z A H L   A U S   D A T U M
\ ------------------------------------------------------------------------
\ Januar und Februar zum 13. und 14. Monat des Vorjahres deklarieren.
: Justage  ( Mon Jhr -- Mon' Jhr')  over 3 < IF 1- swap 12 + swap THEN ;

\ Aus der Jahreszahl die (ganzzahlige) Tagesnummer für 1. März berechnen.
: gJhr>TN  ( n1 -- n2)       \ für Gregorianischn Kalender
    400 /mod 146097 *       swap    \ 146097 = 400*365 + 97
    100 /mod  36524 * rot + swap    \  36524 = 100*365 + 24
      4 /mod   1461 * rot + swap    \   1461 =   4*365 + 1
                365 *     + ;       \    365 =   1*365

: jJhr>TN  ( n1 - n2)        \ für Julianischen Kalender
      4 /mod   1461 * swap          \   1461 =   4*365 + 1
                365 * +             \    365 =   1*365
                  2 - ;      \ Kalenderdifferenz zum Zeitpunkt TZ=0

\ Aus der Jahreszahl die (ganzzahlige) Tagesnummer für 1. März berechnen.
: Jhr>TN  ( n1 -- n2)  jflag @  IF  jJhr>TN  ELSE  gJhr>TN  THEN ;

\ Aus Monatstag (n1) und Monat (n2) die Tage seit 1. März (n3) ermitteln.
: Mon&Tag>Tage  ( n1 n2 -- n3)  1+ 306 * 10 / 123 - + ;

\ Aus Minute (n1) und Stunde (n2) den Tagesbruchteil (r) berechnen.
: Std&Min>Tag  ( n1 n2 -- r)  s>f 24e f/  s>f 1440e f/ f+ ;

\ Aus einem Kalenderdatum die (ganzzahlige) Tagesnummer berechnen.
: Jhr&Mon&Tag>TN  ( Tag Mon Jhr -- n)  Jhr>TN -rot Mon&Tag>Tage + ;

\ Aus einem Kalenderdatum die Tageszahl berechnen.
: Date>TZ  ( Min Std Tag Mon Jhr -- r)
       Justage Jhr&Mon&Tag>TN s>f Std&Min>Tag f+ ;

\ Den »Julian Day« (JD, astron.) berechnen. Nicht verwendet im Programm.
: Date>JD  ( Min Std Tag Mon Jhr -- MJD)  Date>TZ  1721119.5e f+ ;

\ ------------------------------------------------------------------------
\       D A T U M   A U S   T A G E S Z A H L
\ ------------------------------------------------------------------------
\ Aus der (ganzzahligen) Tagesnummer Jahreszahl und Jahrestage berechnen.
: gTN>Jhr&Tage  ( n -- Tage Jahr)         \ für Gregorianischen Kalender
      146097 /mod >r    36524 /mod dup 4 =
            IF  1- nip 36524 swap  THEN  >r
        1461 /mod >r      365 /mod dup 4 =
            IF  1- nip   365 swap  THEN
            r> 4 * +  r> 100 * +  r> 400 * + ;

: jTN>Jhr&Tage  ( n -- Tage Jahr)  2 +    \ für Julianischen Kalender
       1461 /mod >r      365 /mod dup 4 =
            IF  1- nip   365 swap  THEN
            r> 4 * + ;

\ Aus der (ganzzahligen) Tagesnummer Jahreszahl und Jahrestage berechnen.
\ Schaltjahre werden erkannt (4*365=1460 -> 365 3; 4*365+1=1461 -> 0 4).
: TN>Jhr&Tage  ( n -- Tage Jahr)
       jflag @  IF  jTN>Jhr&Tage  ELSE  gTN>Jhr&Tage  THEN ;

\ Aus dem Jahrestag (0...365) Monat und Monatstag berechnen.
\ Januar und Februar gelten als 13. und 14. Monat des virtuellen Jahres.
: Tage>Mon&Tag  ( Tage -- Tag Monat)
       31 + 10 * 306 /mod >r  10 / 1+ r> 2 + ;

\ Aus der Tageszahl das echte Datum bilden.
\ Beispiele: 305.5e -> 0 12 31 12 0; 306.5e -> 0 12 1 1 1
: TZ>Date  ( TZ -- Min Std Tag Mon Jhr)
       fdup  1e fmod  1440e f* .5e f+  f>s  60 /mod  ( Min Std)
             f>s TN>Jhr&Tage swap Tage>Mon&Tag  ( Jhr Tag Mon)
             dup 12 > IF   12 - rot 1+  ( Jan & Feb zum neuen Jahr)
                      ELSE      rot  THEN ;

\ Datum aus TZ berechnen und Variablen laden.
: setDate  ( TZ --)  TZ>Date
       dup Jhr1 ! Jhr2 !   dup Mon1 ! Mon2 !   dup Tag1 ! Tag2 !
       dup Std1 ! Std2 !   dup Min1 ! Min2 ! ;
```

```
\ Aus der (ganzzahligen) Tagesnummer Wochentag berechnen und ausgeben.
: DayOfTheWeek  ( n --)
   2 + 7 mod  CASE
                0  OF  ."  Mo"  ENDOF      1  OF  ."  Di"  ENDOF
                2  OF  ."  Mi"  ENDOF      3  OF  ."  Do"  ENDOF
                4  OF  ."  Fr"  ENDOF      5  OF  ."  Sa"  ENDOF
                6  OF  ."  So"  ENDOF
              ENDCASE ;

\ Ausgabe Datum und Uhrzeit (Format: yyyy-mm-dd hh:mm).
: displayDate  ( min hour day month year --)
   ( Jhr) 4 .#  ." -"  ( Mon) 0 max 2 .#  ." -"  ( Tag) 0 max 2 .#
   space  ( Std) -1 max 2 .# ." :"  ( Min) -1 max 2 .# ;

\ Aus einer Tageszahl das echte Datum errechnen und ausgeben.
: tellDate  ( TZ --)  fdup TZ>Date  displayDate  f>s space DayOfTheWeek ;

\ ------------------------------------------------------------------------
\       T A S T A T U R E I N G A B E
\ ------------------------------------------------------------------------
\ Tastatur-Eingabe für einen Zahlenwert. (Nicht-Zahlen werden abgewiesen.)
: getItem  { W: var-addr D: strg -- }
   BEGIN  cr  strg ( c-addr u) type ." [" var-addr @ 1 .r ." ] "
                          pad 10 accept
          dup IF          pad swap s>number?
                  IF      d>s  var-addr !  true
                  ELSE    2drop            false
                  THEN
              ELSE        0=  ( null string)
              THEN
   UNTIL ;

\ Aktuellen Kalender (gregorianisch oder julianisch) anzeigen.
: tellMode  ( --)
        mode @ 0=  IF  jflag @  IF    ."  (julianisch)"
                                ELSE  ."  (gregorianisch)"  THEN
                                THEN ;

\ Bereitstellung der Datums-Parameter.
: Date1>  ( -- Min Std Tag Mon Jhr)
        Min1 @  Std1 @  Tag1 @  Mon1 @  Jhr1 @  ;

: Date2>  ( -- Min Std Tag Mon Jhr)
        Min2 @  Std2 @  Tag2 @  Mon2 @  Jhr2 @  ;

\ Flag-Variable wird negiert (0 zu -1 bzw. <>0 zu 0).
: toggle  ( a --)  dup @ 0= swap ! ;

\ Automatische Kalenderwahl: TZ<578051 --> Julianischer Kalender.
\ Auf Tag 1582-10-04 (julianisch) folgt Tag 1582-10-15 (gregorianisch).
\ Eine Eingabe bis 1582-10-14 wird hier noch als julianisch gewertet.
\ ------------------------------------------------------------------------
\ Wochentag:             Di Mi Do | Fr Sa So Mo Di Mi Do Fr Sa So | Mo Di
\ Julianisch:    1582-10-02 03 04 | 05 06 07 08 09 10 11 12 13 14 | 15 16
\ Gregorianisch: 1582-10-12 13 14 | 15 16 17 18 19 20 21 22 23 24 | 25 26
\ Tageszähler (TZ):  578038 39 40 | 41 42 43 44 45 46 47 48 49 50 | 51 52
\ ------------------------------------------------------------------------
: decide  ( date --)  mode @ 0=  ( nur hybrid)
        IF     greg Date>TZ  578041e f<  IF juln THEN
        ELSE   5 drops  THEN ;

\ Erste Eingabe von Datum und Uhrzeit.
: getDate1  ( --)
   cr ."  Eingabe Datum mit Uhrzeit:"
   Jhr1 Jhr$  getItem   Mon1 Mon$  getItem   Tag1 Tag$  getItem
   Std1 Std$  getItem   Min1 Min$  getItem
   cr  ."     Eingegeben: "    Date1> displayDate
                               Date1> decide tellMode
                               Date1> Date>TZ  fdup TZ1 f!
               mode @ 0=  IF          jflag toggle
   cr  ."     Entspricht: "    fdup  TZ>Date displayDate tellMode
                                     jflag toggle
                          THEN
   cr ."      Tageszahl: TZ="  fdup  f.'
                      ."  JD="  fdup  1721119.5e f+  f.'
                                  f>s DayOfTheWeek ;
```

```
264
265  \ Zweite Eingabe von Datum und Uhrzeit.
266  : getDate2  ( --)
267      cr ."  Eingabe zweites Datum mit Uhrzeit:"
268      Jhr2 Jhr$  getItem   Mon2 Mon$  getItem   Tag2 Tag$  getItem
269      Std2 Std$  getItem   Min2 Min$  getItem
270      cr  ."     Eingegeben: "    Date2> displayDate
271                                  Date2> decide tellMode
272                                  Date2> Date>TZ  fdup TZ2 f!
273                 mode @ 0=  IF          jflag toggle
274      cr  ."     Entspricht: "    fdup  TZ>Date displayDate tellMode
275                                        jflag toggle
276                           THEN
277      cr ."      Tageszahl: TZ="  fdup  f.'
278                        ."  JD="  fdup  1721119.5e f+  f.'
279                                        f>s DayOfTheWeek ;
280
281  \ ------------------------------------------------------------------------
282  \       T A G E   B Z W .  S O N N E N J A H R E   A D D I E R E N
283  \ ------------------------------------------------------------------------
284  \ Eingabe eines Offsets in Tagen, Ausgabe des resultierenden Datums.
285  : newDate  ( --)  offs s"          Tage = "  getItem
286                    TZ1 f@  offs @ s>f f+
287                    cr ."    Neues Datum: " tellDate cr ;
288
289  \ Aus Vielfachen von Sonnenjahren eine Datentabelle erzeugen.
290  : makeTable  ( --)           trops s"   Sonnenjahre = " getItem
291      trops @ abs stepWid !  stepWid s"  Schrittgröße = " getItem
292      stepWid @  trops @  abs min  abs 1 max  stepWid !
293      trops @  dup 0>=  ( tropjahre f)
294              IF    stepWid @ /up 1+ 0        \ vorwärts
295              ELSE  stepWid @ /   1 swap      \ rückwärts
296              THEN  cr
297          DO      i stepWid @ *  TZ1 f@
298                    i stepWid @ *  s>f ftrop f* f+
299                  14 .r  2 spaces  tellDate  cr
300          LOOP  ;
301
302  \ ------------------------------------------------------------------------
303  \       Z E I T D I F F E R E N Z
304  \ ------------------------------------------------------------------------
305  \ Die Datumsdifferenz in Minuten berechnen.
306  : MinDiff  ( -- n)
307      Tag2 @  Mon2 @  Jhr2 @  Jhr&Mon&Tag>TN  1440 *
308                              Std2 @  60 * +  Min2 @ +
309      Tag1 @  Mon1 @  Jhr1 @  Jhr&Mon&Tag>TN  1440 *
310                              Std1 @  60 * +  Min1 @ + - abs ;
311
312  \ Aus Datumsdifferenz Tage, Stunden und Minuten berechnen.
313  : makeDiff  ( -- Min Std Tage)
314          getDate2  MinDiff  1440 /mod >r  60 /mod r> ;
315
316  \ Aus Datumsdifferenz die Anzahl Sonnenjahre berechnen.
317  : yDiff  ( --)  TZ2 f@  TZ1 f@  f- fabs ftrop f/ ;
318
319  \ Aus Datumsdifferenz Tage, Stunden, Minuten und Sonnenjahre ausgeben.
320  : tellDiff  ( --)  makeDiff  cr ."      Differenz: "
321          . ." Tag[e] "  . ." Stunde[n] "  . ." Minute[n] "  cr
322               yDiff 16 spaces  ." (" f.' ." Sonnenjahre) "  cr ;
323
324  \ Korrektur des aktuellen Datums für den Julianischen Kalender.
325  : actual  ( --)
326      jflag  IF     greg Date1> Date>TZ  fdup TZ1 f!  juln setDate
327             THEN ;
328
329
330  \ ------------------------------------------------------------------------
331  \       H A U P T S C H L E I F E
332  \ ------------------------------------------------------------------------
333  : mainloop  ( --)
334          [char] a  wahl !      \ Voreinstellung des Modus
335    BEGIN
336      getDate1
337   cr cr 12 spaces s" a" type" ."  Eingabe eines zweiten Datums"
338      cr 12 spaces s" b" type" ."  Eingabe einer Anzahl Tage"
339      cr 12 spaces s" c" type" ."  Eingabe einer Anzahl Sonnenjahre"
```

```
340       cr 12 spaces s" q" type" ."  quit to prompt"
341
342                         wahl @  temp !
343       cr ."    Wahl [" wahl @ emit ." ]: " key dup emit cr
344       dup 13 =  IF  drop wahl @  ELSE  dup wahl !  THEN
345           CASE  [char] a  OF  tellDiff       ENDOF
346                 [char] b  OF  newDate        ENDOF
347                 [char] c  OF  makeTable      ENDOF
348                 [char] q  OF  ."   ok" quit  ENDOF
349                       temp @  wahl !
350           ENDCASE
351
352       cr ." --------------- " .s f.s ." --------------- " cr  7 emit ( beep)
353     AGAIN ;              ( Stacks prüfen )
354
355   \ ------------------------------------------------------------------------
356   \       P R O G R A M M S T A R T
357   \ ------------------------------------------------------------------------
358   : hybrid     ( --)  0 mode !
359           cr  16 spaces  ." Hybrider Kalender gewählt" ;
360   : julian     ( --)  1 mode !  juln  actual
361           cr  16 spaces  ." Julianischer Kalender gewählt" ;
362   : gregorian  ( --)  2 mode !  greg
363           cr  16 spaces  ." Gregorianischer Kalender gewählt" ;
364
365   : intro  ( --)
366    cr cr  ."  Datum & Uhrzeit bei Programmstart: "
367           Min1 @  Std1 @  Tag1 @  Mon1 @  Jhr1 @  displayDate
368
369   \ Kalenderwahl: julianisch, gregorianisch, hybrid.
370    cr cr 12 spaces s" g" type" ."  Gregorianischer Kalender"
371       cr 12 spaces s" j" type" ."  Julianischer Kalender"
372       cr 12 spaces s" h" type" ."  Hybrid: Julianisch bis 1582-10-14"
373       cr 12 spaces s" q" type" ."  quit to prompt"
374           [char] g  wahl !
375       BEGIN         wahl @  func !
376           cr ."    Wahl [" wahl @ emit ." ]: " key dup emit
377           dup 13 =  IF  drop wahl @  ELSE  dup  wahl !  THEN
378               CASE  [char] j  OF  julian     -1 temp !  ENDOF
379                     [char] g  OF  gregorian  -1 temp !  ENDOF
380                     [char] h  OF  hybrid     -1 temp !  ENDOF
381                     [char] q  OF  ."   ok" quit         ENDOF
382                     func @  wahl !
383                     0 temp !
384               ENDCASE  temp @
385       UNTIL
386   cr ;
387
388   \ ------------------------------------------------------------------------
389           12 set-precision  ( statt 15 oder mehr)
390           intro
391           mainloop
```

Schlussbemerkung

Nach Fertigstellung dieses Artikels bin ich über die Routinen unter dem Titel „Datumsberechnung in Forth“ gestolpert, die ebenfalls mit dem 1. März 0 die Tage zu zählen beginnen, allerdings nach Julianischem Kalender. Siehe:
`http://forth-ev.de/wiki/doku.php/examples:daymonthyear`

Weitere Links

Online-Datums-Umrechner mit chronologischem JD (und Tageszählung ab 29. 12. 0000):
`http://www.pfeff-net.de/kalend.html`
Online-Datums-Umrechner mit chronologischem JD und 19 verschiedenen Kalendern:
`http://www.nabkal.de/kalrech1.html`
Online-Datums-Umrechner mit astronomischem JD und MJD:
`http://www.nabkal.de/kalrechJD.html`
Listing `http://www.forth-ev.de/filemgmt/singlefile.php?lid=439`

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

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

Gruppengründungen, Kontakte

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

µP–Controller Verleih

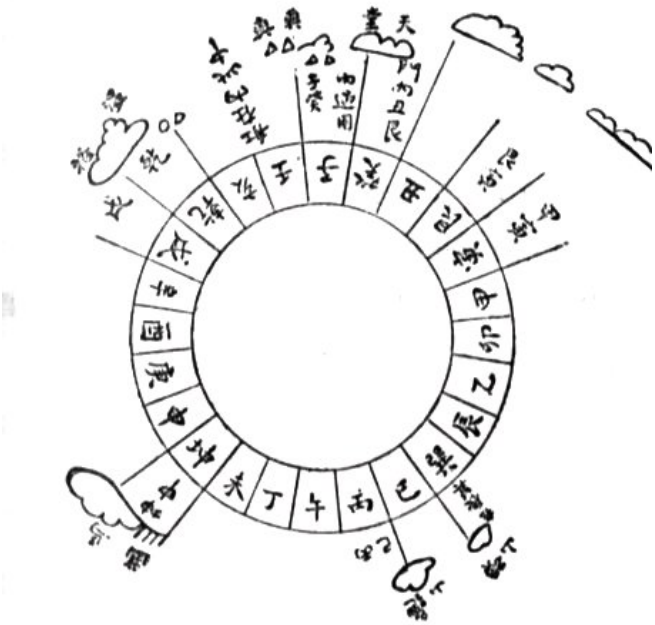
Carsten Strotmann
`microcontrollerverleih@forth-ev.de`
`mcv@forth-ev.de`

Spezielle Fachgebiete

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

Termine

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



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.

So war's auf der EuroForth 2012

Die EuroForth 2012 fand im bezaubernden Exeter College statt, ein Ort, an dem (Swap-)Drachen und bärtige Programmierer nicht viel ungewöhnlicher erscheinen als in Hogwarts. Die 8 Sessions fanden im Salon des Rektors statt, die Kaffeepausen in seinem Garten.

Programm

Bill Stoddard Forth semantics for Computer verification

Ian van Breda Building an LR Parser Using Forth

Ulrich Hoffmann Applying Model checking techniques to Forth

Willi Stricker Strip Forth processor presentation

Anton Ertl Objects 2

Klaus Schleisiek Explaining simpleOOP

Andrew Haley Standardise Forth OOP

Dirk Brühl Forth for Education — 4€4th and 4€4th IDE

Stephen Pelc Educational Forth (workshop)

Andrew Read The N.I.G.E. Machine: an FPGA based micro computer

Stephen Pelc Notation matters

Gerald Wodni Forth to .NET interface

Peter Knaggs Java Forth

Bernd Paysan Recognizers (shifted evening talk)

Bernd Paysan Forth meta object protocol (workshop)

Bernd Paysan net2o — from Fiction to Reality

Bill Stoddart Forth Local Variable Semantics

Ian van Breda Some comments on Forth Standard 200x

Awards Gold, Silver & Bronze awards for Papers

Die Papers und Slides liegen auf `http://www.complang.tuwien.ac.at/anton/euroforth/ef12/papers/`.