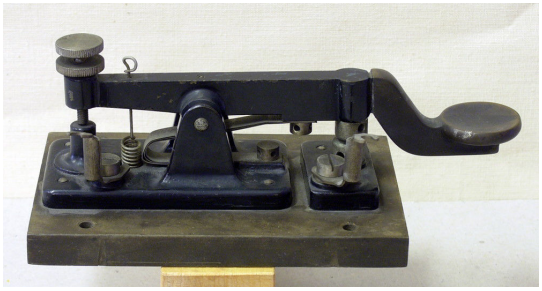
Vierte Dimension

1/2012

Das Forth–Magazin

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

In dieser Ausgabe:



F O R T H

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,

In *Vierte Dimension 2011-04* schrieb Erich Wälde

„Ich gebe hiermit den Staffelstab an Ulli Hoffmann weiter.“

Nun denn — hier ist sie nun — die erste Ausgabe des 28. Jahrgangs unseres Forth-Magazins. „Forth — gibt's das denn überhaupt noch?“ Diese Frage habe ich gerade im vergangenen Jahr wiederholt gehört und erwidert: „Ja!“ das gäbe es noch und eine kleine Gemeinde von Forth–Begeisterten macht immer noch spannendste Dinge mit Forth. Dass unser Elan ungebrochen ist, kann man an dieser Ausgabe sehen:

Johannes Teich widmet sich dem Geburtstags–Paradoxon und führt seine Erläuterungen zu seiner Wave-Engine fort.

Fred Behringer zeigt uns, wie man weitere Kontrollstrukturen, dieses Mal `IF`, `ELSE` und `THEN` in high-level-Forth definieren kann, wenn man passende Annahmen über das Forth-System macht (Manipulation der Adressen auf dem Return–Stack ist zulässig und die zugehörige Arithmetik ist bekannt).

Erich Wälde diskutiert einen Fehler der im Morsecode-Programm des vergangenen Hefts zu beobachten war. Liegt das Problem in der Anwendung oder im zu Grunde liegenden amForth? Urteilt selbst.

Michael Kalus berichtet uns über CamelForth für den MSP430FR5739 von Texas Instruments, wo das Forth im FRAM (nicht flüchtiges RAM) abgelegt ist. Außerdem stellt er uns das 4€–billige TI-LaunchPad mit MSP430 und 4e4th vor. Dieses kleine Board mit interaktivem Forth war der Renner auf der diesjährigen Forth–Tagung. 4e4th ist derzeit in der Entwicklung und so werden wir sicherlich in Zukunft mehr über dieses interessante System erfahren.

Seit vielen Jahren produzieren wir die Vierte Dimension in einem verteilten Arbeitsprozess, dessen Dreh- und Angelpunkt das Subversion-Repostory unter `www.forth-ev.de/repos/vd` ist. Carsten Strotmann weist uns auf Fossil hin, ein verteiltes Versionskontrollsystem der dritten Generation (1G: SCCS/RCS, 2G: VCS, SVN, 3G: git, mercurial, fossil). Dieses System macht Versions-Verwaltung so einfach, dass jeder(!) sie benutzen sollte. Ein Workshop auf der Forth–Tagung beleuchtete die Thematik genauer.

Durch unseren SWAP-Preisträger Stephen Pelc auf der EuroForth 2011 in Wien angefacht, ist in den letzten Monaten eine heftige Diskussion um die beste Art der objekt-orientierten Programmierung in Forth entbrannt. Soll man auf early- oder late-binding setzen. Muss Polymorphismus und Kapselung unterstützt werden und wenn ja, wie? Sind Virtual Method Tables essentiell oder entbehrlich? Kann das Forth-Dictonary für die Namensverwaltung verwendet werden? Welche Syntax haben Klassen- und Methoden-Definitionen oder -Aufrufe? In Forth ist das Implementieren objekt-orientierter Konzepte so einfach, dass jeder seine eigene OOP-Erweiterung haben kann. Austauschbar sind die objekt-orientierten Programme dieser unterschiedlichen Dialekte dann aber nicht. Einige Vorschläge liegen vor, aber so richtig hat keine Seite die anderen bisher überzeugen können. Vielleicht können wir in den kommenden Ausgaben unseres Forth-Magazins ja die unterschiedlichen Ansätze genauer beleuchten.

Ich wünsche uns allen viel Vergnügen beim Lesen,

Ulrich Hoffmann

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
`http://www.forth-ev.de/repos/vd/2012-01/`

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

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

3 MSP-EXP430FR5739 sind da.

```
Date: Mon, 23 Jan 2012 14:34:00 +0100
From: Carsten Strotmann <carsten@strotmann.de>
Subject: Re: 3 MSP-EXP430FR5739 sind da.


Hallo,

ich habe heute die Boards offiziell in den Verleih
aufgenommen und auf die Webseite gestellt.

Das letzte bei mir verbliebene Board wurde heute an
Rolf Lauer ausgeliefert, der eine Ausleih-Anfrage
gestellt hat.

Bis Anfang Maerz bin ich jetzt in Down-Under und
kann daher keine Boards versenden oder annehmen.

Beste Gruesse

Carsten

On 8/17/11 2:01 PM, Kalus Michael wrote:
> Moin zusammen.
>
> Hab nun 3x MSP-EXP430FR5739 hier -
> Experimentierbords mit dem neuen FRAM-Chip
> MSP430FR5739 von Texas Instruments drauf.
> Würde die gerne in den Verleih abgeben.
>
> Michael
```

FET-Pro430 Lite und CamelForth auf dem Experimenter Board

Übrigens gibt es bei Elprotronic [1] ein gut gelungenes Tool, um die MCUs von TI, auch unser FRAM im MSP430FR5739, zu laden, zu verifizieren, auszulesen, das kostenlose FET-Pro430 Lite. Damit ist es ganz leicht, das CF430FRexp in das Experimenter Board von TI zu schreiben, benutze einfach AUTO PROG dazu.

Das Image dieser CamelForth-Version heißt CF430FRexp.a43 und liegt im Repository der Forth-Gesellschaft. Diese Datei enthält das Image im Intel-Hex-Format. Viel Vergnügen. mk

[1] `http://www.elprotronic.com/` [2] `http://www.forth-ev.de/repos/CF430FR/CF430FRexp.a43`

Wenn der MSP430FR5739 sich nicht mehr laden lassen will...

ist der BSL–Scripter von Texas Instruments die Rettung. Nicht verschwiegen werden soll hier auch, dass es beim CF430FR–Projekt vorübergehend ein Problem mit der Verständigung zwischen der IAR-Workbench und dem MSP430FR5739 gab. Hartnäckig erschien auf einmal die Meldung „Could not find device..“ und das CF430FR–Projekt war nicht mehr in die MCU zu laden. Dabei lief das Forth darin ganz munter, und DUMP konnte benutzt werden, um über den Zustand der Speicherbereiche Auskunft zu geben. Aber ich hab nicht herausfinden können, was da passiert ist. Selbst eine Analyse der Signale, die über die Programmierschnittstelle kamen, brachte keinen Aufschluss, eigentlich alles richtig, aber der Chip wollte einfach nicht.

Abhilfe kam von Dirk Brühl. Er hat sich den BSL–Scripter angesehen, mit dem TI seine Chips lädt, wenn sie nicht auf so Experimentierplatinen mit USB–Anschluss sitzen. Das Verfahren benutzt ganz klassisch den *COM-Port* eines PCs, und über ein Interface zur Pegelanpassung lassen sich damit die Kommandos an den *boot strap loader (BSL)* schicken, der in jeder dieser TI–MCUs ROM–resident gegeben ist. Allerdings erschien das von TI angegebene Interface doch zu aufwändig, und er hat eine einfachere Schaltung angegeben und gebaut.

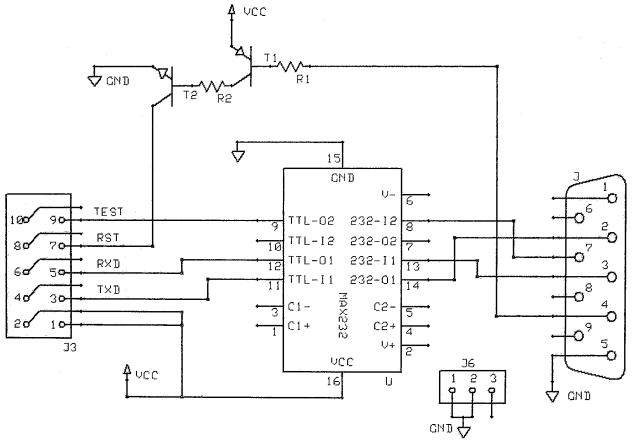


Abbildung 1: BSL–Interface–Schaltplan

Das rettende Platinchen kam kurz darauf als Lufpost angeflogen, und über die angegebenen Kommandos konnte damit der Chip tatsächlich schon im ersten Versuch zurückgesetzt werden, und ließ sich fortan wieder brav auch vom IAR aus laden, so dass das CF430FR–Projekt doch noch abgeschlossen werden konnte. So schön die Experimentierplatine auch geworden ist, die TI dort vertreibt, so ganz ohne die BSL–Hilfe kommt sie auch nicht immer aus, und man ist gut beraten, den klassischen Zugang per BSL–Scripter im Hinterkopf zu behalten — für alle Fälle. mk

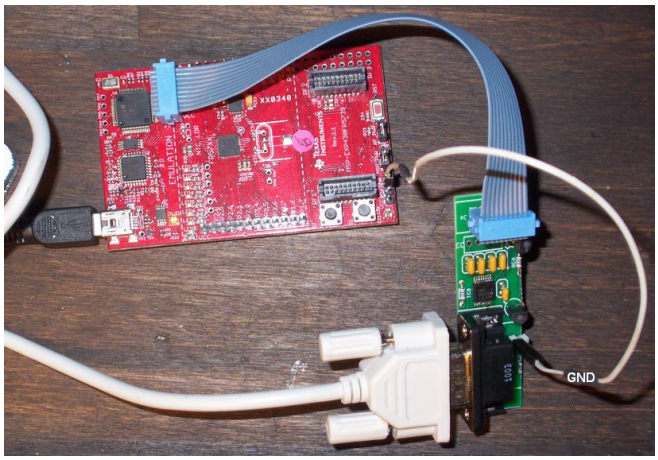


Abbildung 2: BSL-Interface

Kommentar zum Artikel „Kontrollstrukturen als Colon-Definitionen und ohne die Immediate-Eigenschaft“ in der VD 4/2011 von Fred Behringer

Zunächst möchte ich auf den Versuch eingehen, die drei Forth-Basisbefehle `LIT`, `BRANCH` und `?BRANCH` als High-Level Befehle zu definieren:

Sie stellen im Forth-System eine Besonderheit dar, da sie einen einkompilierten direkten Parameter benötigen, anders als fast alle weiteren Forth-Befehle. Außerdem werden sie vom Compiler benutzt, um Literale und Strukturen zu kompilieren. Dabei wird der Befehl `LIT` vom Befehl `LITERAL`, der Befehl `?BRANCH` (bedingter Sprung) von den Strukturbefehlen `IF`, `WHILE` und `UNTIL` und der Befehl `BRANCH` (unbedingter Sprung) von den Strukturbefehlen `ELSE`, `AGAIN` und `REPEAT` benutzt. Die Strukturbefehle `BEGIN` und `THEN` dienen dem Compiler lediglich als Sprungmarken. Die genannten Literal- und Strukturbefehle sind deswegen als „Immediate-Befehle“ programmiert. Ob es möglich ist, wie von Fred Behringer versucht, sie als „Nicht-Immediate-Befehle“ zu definieren, kann ich nicht beurteilen, bin aber sehr skeptisch.

Der Artikel zeigt aber die zwangsläufig auftretenden Schwierigkeiten beim Versuch, den benötigten Basis- (oder Minimal-) Befehlssatz für ein Forth-System immer weiter zu reduzieren:

Die Definition:

```
: BRANCH R> @ >R ;
```

funktioniert, wäre also einsetzbar! Anders verhält es sich mit der Definition:

```
: LIT R@ @ R> 2 + >R ;
```

sie funktioniert zwar im Prinzip, benutzt jedoch für die Literale `2` sich selbst. Das geht also nicht! Das Gleiche passiert, wenn man versuchen würde, den Befehl `?BRANCH` zu definieren:

```
: ?BRANCH IF R> 2 + >R ELSE R> @ >R THEN ;
```

Auch hierbei wäre der `?BRANCH` im `IF`-Befehl als bereits vorhanden vorausgesetzt! Man dreht sich damit gewissermaßen im Kreis, da eine Rekursion in diesem Fall nicht möglich ist.

Die Aussage, dass der (in Forth definierte) Befehl `D+` ebenso als Basisbefehl dienen kann, wie der von mir definierte (in Forth unbekannte) Befehl `+C`, ist prinzipiell richtig. Für die Benutzung von `+C` gibt es jedoch formale und insbesondere auch einen technischen Grund:

Bei der Addition zweier n-Bit-Zahlen kann das Ergebnis maximal n+1 Bits betragen. Das überzählige Bit wird bei Mikroprozessoren üblicherweise als Carry-Bit in einem Flag-Register gespeichert. Da Forth, wie alle anderen Programmiersprachen auch, kein Flag-Register kennt, ist es sinnvoll, das Bit in einem zusätzlichem n-Bit-Wort abzulegen. Damit wird aus der Addition zweier n-Bit Worte ein (n+n)-Bit Ergebnis. Das Gleiche wird bei der Multiplikation praktiziert, siehe Forth-Befehl `UM*` (n Bits * n Bits => 2n Bits). Damit ist das Ergebnis immer eindeutig! Außerdem besteht dann die Möglichkeit, die Addition auf Zahlen beliebiger Bitbreiten zu erweitern und der Anwender kann selbst entscheiden, ob er das überzählige Bit braucht oder nicht, z. B.:

```
: + +C DROP ;
```

Formal ist der Befehlssatz ohnehin nur für Single-Befehle gedacht, Double-Befehle betrachte ich lediglich als Erweiterung des Single-Befehlssatzes.

Der technische Grund für den Befehl `+C` ist folgender: Der STRIP-Forth-Prozessor kann (wie alle mir bekannten Stack-Prozessoren) pro Befehl auf maximal 2 Eingangs-Parameter zugreifen und maximal 2 Ausgangs-Parameter ausgeben (jeweils die obersten 2 Stackworte). Der Befehl `D+` braucht aber 4 Eingangs-Parameter (2 für jeden Summanden). Anmerkung: Aus dem gleichen Grund sind auch Befehle wie `ROT` oder `OVER` dort nicht als Hardware-Befehle realisierbar.

Willi Stricker

The programming language formally known as '5'

In der VD 1/2010 hat Bernd Ulmann über die Programmiersprache **5** berichtet, eine Sprache, welche die Programmierkonzepte von Forth mit denen von APL verbindet.

Seit Sommer 2011 heißt diese Sprache nun **lang5** [1], was sich positiv auf die Suchmöglichkeiten in Internet Suchmaschinen auswirkt.

Vector, die Publikation der britischen APL–Vereinigung (British APL Association) hat im Dezember 2011 einen neuen Artikel von Bernd Ulmann über lang5 veröffentlicht [2].

Im Oktober 2011 hat es auch ein neues Release der Sprache gegeben. Das aktuelle Release, sowie weiterführende Informationen über lang5 findet sich auf der Webseite des Projektes [3].

C.S.

[1] `http://lang5.sourceforge.net/tiki-view_blog_post.php?postId=7`
[2] `http://archive.vector.org.uk/art10500710`
[3] `http://lang5.sourceforge.net/tiki-index.php?page=HomePage`

Forth - ein etwas ausgefallenes Programmiersystem

„Ein etwas ausgefallenes, aber sehr effizientes Programmier- und Testsystem (für den MSP430 von Texas Instruments) ist FORTH. Neben einigen OpenSource-FORTH-Projekten gibt es ein sehr gutes professionelles FORTH-Entwicklungssystem, Swift-X. Es verwendet die Umgekehrte Polnische Notation (UPN).“

Aus dem Kreis der Forth–Gesellschaft kann diese Bemerkung in Wikipedia sicher nicht stammen. Uns wäre nicht eingefallen, die Bezeichnung „etwas ausgefallen“ zu verwenden.

Aber die Bemerkung an sich geht in die richtige Richtung. Könnte es nicht sein, dass diesmal gelingt, was beim R8C vor einigen Jahren auf der Strecke blieb, nämlich beim experimentier- und bastelfreudigen Volk stärkeres Interesse für Forth zu wecken? Auf der Forth-Tagung im Kloster Beukenhof (in der Nähe von Tilburg und s'Hertogenbosch in den Niederlanden) hat jedenfalls dieses Fast-Geschenk namens LaunchPad von Texas Instruments, nicht zuletzt dank der emsigen Bemühungen von Michael Kalus, großen Zuspruch erhalten: Im E-Mail-Verteiler der LaunchPad-Erwerber (aus Deutschland, Österreich, den Niederlanden und Großbritannien) sind 22 Adressaten aufgeführt. (Weiteres im Artikel von Michael im vorliegenden Heft.)

Fred Behringer

Geburtstagsfragen

Hannes Teich

Irgendjemand hat mir vor längerer Zeit den Floh ins Ohr gesetzt, dass 23 Personen genügen sollen, damit mindestens zwei von ihnen mit über 50 Prozent Wahrscheinlichkeit am selben Jahrestag Geburtstag haben. Besonders populär scheint die Aufgabenstellung nicht zu sein, denn bei Google wurde ich nicht fündig. „Hilfe, wie viele Spätzle für 23 Personen???“ ist da zu lesen und vieles mehr, was die Geburtstagsfrage nicht beantwortet.

Darum habe ich Gforth bemüht, der Sache auf den Grund zu gehen. Ein Zufallsgenerator verteilt die Geburtstage von 23 Personen auf ein Array von 365 oder 366 Elementen, und dann wird geprüft, ob in mindestens einem dieser Elemente mehr als eine Eintragung stattfand. Das ergibt schließlich ein Ja oder ein Nein. Um einen statistischen Wert in Prozenten zu erhalten, wird eine größere Menge von Menschengruppen zu je 23 Personen untersucht und die Anzahl Treffer durch die Anzahl Gruppen dividiert. Um Floating-Zahlen zu vermeiden, werden die Treffer vorher mit Zehntausend multipliziert.

Tja, die Behauptung mit den 23 Personen scheint zu stimmen, doch der geheimnisvolle Grund für diese Tatsache bleibt dabei verborgen. Bemerkenswert auch, dass für ein kurzes Jahr von sieben Tagen (also eine Woche) ganze vier Leute aufgewendet werden müssen.

Mit BASIC-256 komme ich zu gleichen Ergebnissen, aber dennoch bleibt ein Quäntchen schlechtes Gewissen: Der verwendete Zufallsgenerator produziert Zahlen von 2^0 bis $2^{32}-1$, das heißt, die Null fehlt. Um die dadurch entstehende Ungerechtigkeit zu messen, wären allerdings weit mehr als hunderttausend Gruppen nötig, und die fressen schon ein paar Sekunden.

Das Programm ist soweit handlich, man kann geschwind die Eingangswerte ändern oder belassen. An Verbesserungsvorschlägen (more elegance) bin ich (ht@pop.ms) stets interessiert. Für Leute ohne Gforth ist in Abbildung 1 ein Bildschirm–Schnappschuss beigefügt.

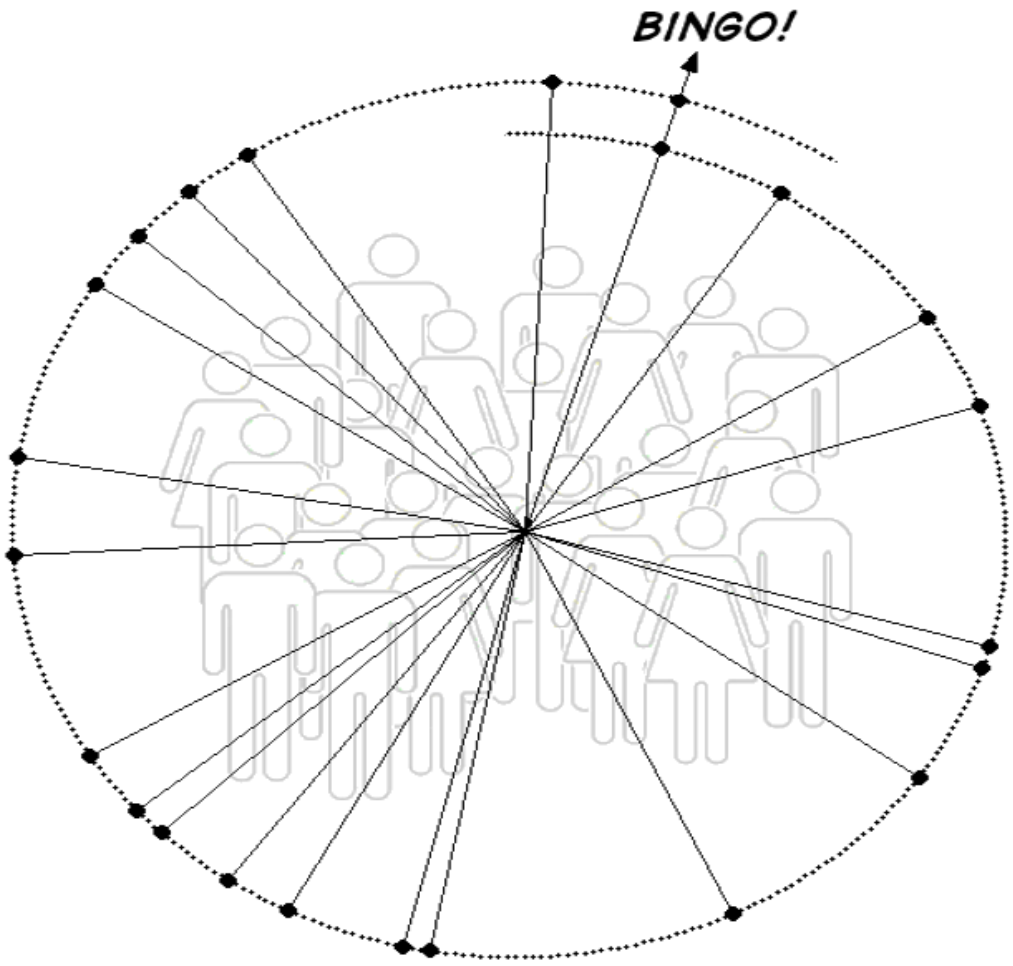
```
Wahrscheinlichkeit gemeinsamer Geburtstage
in einer Gruppe zufälliger Personen.
Dazu müssen viele Gruppen getestet werden.

Geben Sie Anzahl der Gruppen und Personen
sowie die Jahreslänge ein (oder return):

Gruppen:   [ 100000 ]
Personen:  [ 23 ]
Tage/Jahr: [ 365 ]   ...

100000 Gruppen zu 23 Personen ergaben 5034 Treffer.
Damit ist die Wahrscheinlichkeit für mindestens
einen gemeinsamen Geburtstag ungefähr 50.34 %
```

Abbildung 1: Bildschirm-Schnappschuss

Listing

```
\ Gforth 0.7.0                              jgt 02jan12
\ ask (mit s>number?) kann eliminiert werden.

: head  ( --)  cr cr
  cr ."   Wahrscheinlichkeit gemeinsamer Geburtstage"
  cr ."   in einer Gruppe zufälliger Personen."
  cr ."   Dazu müssen viele Gruppen getestet werden." ;
: request  ( --)  cr
  cr ."   Geben Sie Anzahl der Gruppen und Personen"
  cr ."   sowie die Jahreslänge ein (oder return):" cr ;

   100000  value groups
       23  value persons
      365  value days

: ask  ( --)
    begin  cr ."   Gruppen:   [ " groups . ." ] "
                           pad 7 accept
           dup if          pad swap s>number?
                    if     d>s abs to groups  true
                    else   2drop              false
                    then
               else        0=  ( null string)
               then
    until
    begin  cr ."   Personen:  [ " persons . ." ] "
                           pad 3 accept
           dup if          pad swap s>number?
                    if     d>s abs to persons true
                    else   2drop              false
                    then
               else        0=  ( null string)
               then
    until
    begin  cr ."   Tage/Jahr: [ " days . ." ] "
                           pad 3 accept
           dup if          pad swap s>number?
                    if     d>s abs to days    true
                    else   2drop              false
                    then
               else        0=  ( null string)
               then
    until ;

   create date-array days allot
          variable  accu
          variable  flag

\ Random generator (1...2^32-1)
  1234567890 value (rnd) \ seed <> 0
: rnd32  ( -- n)
    (rnd) dup 13 lshift xor
    dup 17 rshift xor
    dup dup 5 lshift xor to (rnd) ;

\ Random generator (0...2^16-1)
: rnd16  ( -- n)  rnd32 65535 and ;

\ Random generator (0...days-1)
: rnd  ( -- n)  rnd16 days * 65536 / ;
```

```
61
62  : clear-array  ( --)
63      days 0 do  0 date-array i + c!   loop ;
64
65  : fill-array  ( --)
66      persons 0 do  date-array rnd +
67                    dup c@ 1+ swap c!  loop ;
68
69  : doubles?  ( -- f)
70      days 0 do  date-array i + c@
71                 1 > if -1 flag ! then  loop
72                 flag @ ;
73
74  : workout  ( -- n)  0 accu !
75      groups 0 do  0 flag !  clear-array fill-array
76                   doubles? if accu dup @ 1+ swap !
77                            then
78                 loop         accu @ 10000 * groups / ;
79
80  : percent  ( n --)  s>d swap over dabs
81                  <# # # [char] . hold
82                  # #s rot sign #> type ."  % " ;
83
84  : tell  ( n --)  cr
85      cr ."   " groups . ." Gruppen zu " persons .
86              ." Personen ergaben " dup . ." Treffer."
87      cr ."   Damit ist die Wahrscheinlichkeit für "
88              ." mindestens"
89      cr ."   einen gemeinsamen Geburtstag ungefähr "
90              percent cr ;
91
92  : nomore?  ( -- f)  cr cr ."   nochmal? [ j/n ] "
93                      key [char] n = ;
94
95  : wayout  ( --)
96      cr ."   QUIT - Neustart mit »go«.  ok" quit ;
97
98  : go  ( --)  head begin  request ask ."  ..."
99                           workout tell nomore?
100                          if cr wayout then
101                  again ;    go
102
```



Wave Engine (2)

Hannes Teich

Weiter geht's mit der virtuellen „viermanualigen“ PC-Orgel, die mit Hilfe eines Programms in Gforth eine ASCII-Partitur in eine Wave-Datei wandelt. In der letzten Folge wurden vorgestellt: die Grundfunktion des Sinus-Generators, der Aufbau der Wave-Datei sowie die Interpretation von Notenzeilen. Diesmal wenden wir uns der Registrierung und dem Datentransport zu. Zunächst eine grobe Übersicht „über alles“.

Die Beschickung des Sinus-Generators ist, wie **Abb. 3** zeigt, ziemlich üppig. Ursprünglich hatte ich mit der *Wave Engine* (Kosename: *Waver*) nur ein Werkzeug zum Experimentieren mit reiner Stimmung im Sinn. Der Riesen-Aufwand wäre hierfür nicht nötig gewesen, aber ich konnte ihn mir einfach nicht verkneifen. :-)

Es sind zwar noch nicht alle Funktionen am Laufen, aber das Konzept steht soweit und wird sich – durch diese Artikelserie gedopt – hoffentlich bald akustisch in voller Pracht präsentieren können. Die Regeln für die Partitur können jedenfalls jetzt schon beschrieben werden. Was bislang akustisch geht, zeigen die Links am Ende.

In der Partitur stecken die Vorgaben für alle Aktionen, als da sind: Feste und veränderbare Voreinstellungen, die Registrierung, die Musiknoten sowie Kommentare, die teilweise in die Wave-Datei übernommen werden. Die Registrierung kann direkt erfolgen oder in Records abgelegt werden, von wo die Inhalte bei Bedarf durch *Habitus*-Marken abgeholt werden, die ins Notenbild eingestreut sind, ohne dieses unnötig aufzublähen. Hinter solchen Marken kann sich sehr viel Registrierung verbergen.

Die Notenzeilen für die vier virtuellen *Manuale* werden nacheinander gelesen, sollen aber zugleich erklingen. Das macht einen *FIFO*-Pufferspeicher nötig, aus dem der *Loader* jeweils das zum *Generator* schickt, was dieser gerade

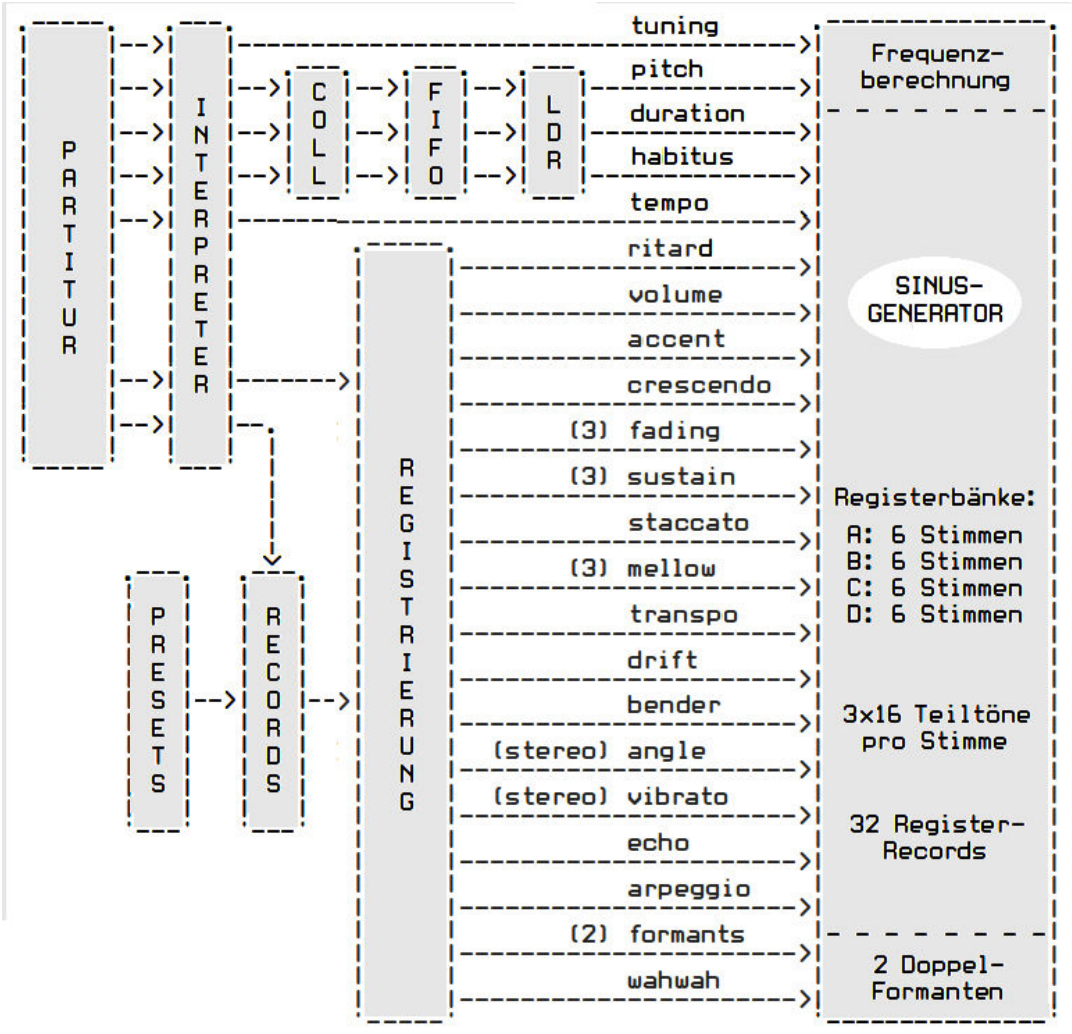


Abbildung 1: Vereinfachtes Schema der Wave Engine

benötigt. Der dem Interpreter nachgeschaltete Collector sorgt dafür, dass die Daten im rechten Format an den Pufferspeicher übergeben werden.

Die Funktionen im Einzelnen, grob beleuchtet

- **tuning** – die Stimmungshöhe, normalerweise 440 Hertz für das „eingestrichene A“.
- **pitch** und **duration** – Höhe und Dauer eines Tons.
- **habitus** – Marken, durch welche die „Records“ angezapft werden.
- **tempo** – Anzahl Viertelnoten pro Minute.
- **ritard** – langsamer werden (ritardando). Negativ: schneller.
- **volume** – Partialtöne einzeln und alle gemeinsam.
- **accent** – Lautstärke für einen oder mehrere Töne erhöhen/erniedrigen.
- **crescendo** – anschwellende Lautstärke. Negativ: abschwellen.
- **fading**, **sustain**, **staccato** – Abklingen, Nachklingen, Zwischenpausen.
- **mellow** – weicher Toneinsatz.
- **transpo** – Transponierung in andere Tonarten.
- **drift** – Verstimmung, z. B. zur Erzeugung der Naturseptime.
- **bender** – Ziehen der Tonhöhe.
- **angle** – Richtung der Tonquelle (stereo).
- **vibrato**, **tremolo**, **beat** – Frequenz-, Amplituden-, Phasenvibrato.
- **echo** – leiser werdendes Flatterecho. (Etwas dürftiger Ersatz für Hall; braucht viel Speicher.)
- **arpeggio** – gebrochene Akkorde.
- **formants** – von Tonhöhe unabhängige Vokale. Durch Berechnung der Obertöne erzeugt. Manche Vokale haben zwei Maxima, deshalb „doppelt“.
- **wahwah** – wechselnde Klangfarbe, etwa wie Trompete mit vorgehaltenem Hütchen. (Durch Überblendung zwischen zwei Vokalen erzeugt.)

Was fehlt noch? Außer Hall wüsste ich nichts mehr. Hall zu berechnen übersteigt meine Fähigkeiten. Immerhin kann man das Wave-Ergebnis nachträglich durch eine Hallmaschine laufen lassen. Oder man begnügt sich mit Flatterecho. (*Les Paul* hat davon viel Gebrauch gemacht.) **vibrato** und **tremolo** unterstütze ich nur halbherzig, die Funktion **beat** reicht mir eigentlich. Sie klingt recht gut, etwa wie beim Vibraphon von *Milt Jackson*.

Bei einer Pfeifenorgel werden die Registerknöpfe gezogen, ehe Manuale und Pedal mani- und pedipuliert werden. Beim *Waver* wird die Registrierung durch die Partitur vorgenommen. In beiden Fällen muss eine „Apparatur“ hinter den Kulissen für die Verwirklichung sorgen.

Folgende Funktionen aus obiger Palette waren für das Tonbeispiel *Top One* im Einsatz: **tuning, pitch, duration, tempo, volume, fading, sustain, staccato**. Schmerzlich vermisst wurden die Funktionen **mellow** und **bender**.

Weicher Toneinsatz im Prinzip

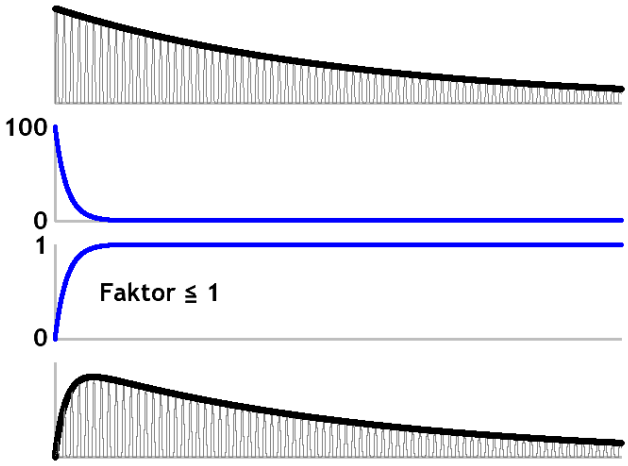


Abbildung 2: Bildung des weichen Toneinsatzes

Die **Abb.2** zeigt, wie ich mir die Generierung des weichen Toneinsatzes (**mellow**) vorstelle. Oben ist die Hüllkurve eines „gezupften“, abklingenden Tons zu sehen, wie er beim Thema *Generator* beschrieben worden ist: Die Amplitude wird laufend mit einem Faktor „kleiner Eins“ multipliziert, was zu konstanten Halbwertszeiten führt. Die Kurve darunter entsteht auf gleiche Weise, nur mit stärkerer Dämpfung und mit vorgegebener Anfangsamplitude von (sagen wir) 100 Einheiten. Die dritte ist die auf den Kopf gestellte zweite Kurve, was durch Subtraktion von 100 bewerkstelligt wird. Außerdem wird durch 100 dividiert, damit sie als Dämpfungsfaktor dienen kann. Mit diesen Faktor-Werten wird die Ausgangskurve multipliziert, und – voila! – das Resultat kann sich sehen (und hören) lassen.

Implementieren des weichen Toneinsatzes

Wir betrachten nur *Manual A*, denn die vier *Manuale* sind gleich und selbständig. Jedes *Manual* hat sechs Stimmen, jede Stimme hat drei Teiltonreihen, und jede dieser Reihen hat 16 Teiltöne. Man könnte den gewünschten Effekt jeder Stimme zuordnen (eine Einstellung für alle Stimmen), aber ich meine, die Teiltonreihen sollten separat behandelt werden können (also drei Werte für alle sechs Stimmen).

```
regA{ mellow=50,50,50; }
regA{ *1 mellow=50,50,50; }
```

Hier sind zwei Varianten einer **mellow**-Registeranweisung zu sehen: Für die drei Teiltonreihen des *Manuals* A ist ein Dämpfungswert von 50 angegeben. Die erste Variante kann überall zwischen Notenzeilen auftauchen. Die zweite speichert die Vorgabe als *Habitus* in einem Records-Pool, von wo sie durch Angabe von `*1` innerhalb einer Notenzeile geholt werden kann, um sogleich zu wirken. In solch einer Register-Zeile können mehrere (oder alle) Registeranweisungen enthalten sein. Auf die Interpretation von Partiturzeilen will ich an dieser Stelle aber nicht eingehen (die Syntax gibt's paar Seiten weiter unten), sondern die **mellow**-Funktion als solche betrachten. Die Dämpfungskurve in **Abb. 2** reiht sich zwanglos in bereits vorhandene Dämpfungen (**fading**, **sustain**) ein. Einst waren die Vorgaben noch nicht in der Partitur, sondern Teil des Programms, was sehr hinderlich war. Der Anschaulichkeit halber greife ich aber gern auf diesen Zustand zurück.

```
: adjust  ( u - r)  0 max 3000 min
                    ?dup IF drop 3000 THEN
                    draft_ IF 4 * THEN
                    1e ( u) s>f 1e5 f/ 1e f+ f/ ;

      5  adjust  0 :fade-A* f!    \ fading
     18  adjust  1 :fade-A* f!
     12  adjust  2 :fade-A* f!

     50  adjust  0 :sust-A* f!    \ sustain
     50  adjust  1 :sust-A* f!
     50  adjust  2 :sust-A* f!

     50  adjust  0 :mellow-A* f!  \ mellow
     50  adjust  1 :mellow-A* f!
     50  adjust  2 :mellow-A* f!
```

Die Funktion `adjust` ist nötig, um für die Vorgaben handliche Werte zu kriegen. Sie macht das durch die Formel: `r = 1/(1+u/100000)`. Beispiele:

```
     1 adjust f. 0.99999  ok
     5 adjust f. 0.99995  ok
    10 adjust f. 0.99990  ok
    50 adjust f. 0.99950  ok
   100 adjust f. 0.99900  ok
   500 adjust f. 0.99502  ok
  1000 adjust f. 0.99010  ok
  3000 adjust f. 0.97087  ok
     0 adjust f. 0.97087  ok
```

Wert 1 liefert die geringste Dämpfung. Für den weichen Toneinsatz heißt das, dass der Ton lange braucht, um seinen Nennwert zu erreichen. Dagegen unterdrückt das Maximum 3000 (oder auch 0) den **mellow**-Effekt hinreichend und lässt den Toneinsatz „knallen" (wie in *Top One*).

`:mellow-A*` bezeichnet ein Array-Element, wie es in der letzten Artikelfolge beschrieben worden ist:

```
create mellow-A*|  3 8 *  allot  ( 3 cells)
: :mellow-A* ( cell# -- addr) 8 * mellow-A*| + ;
```

Hierbei handelt es sich um die Speicherung der Vorgaben. Nun brauchen wir noch die Arrays für die beiden Steuerkurven aus **Abb. 2**. Diese müssen für jede der sechs Stimmen vorhanden sein, also für 18 Teiltonreihen:

```
create dflat-A*|  18 8 *  allot  ( 18 cells)
: :dflat-A* ( cell# -- addr) 8 * dflat-A*| + ;

create mel-A*|  18 8 *  allot  ( 18 cells)
: :mel-A* ( cell# -- addr) 8 * mel-A*| + ;
```

Immer wenn ein neuer Ton beginnt, werden (in der Routine `latch>gen-A`, hier nicht dargestellt) drei Array-Elemente mit `100e` geladen:

```
  3 0  DO  100e i :dflat-A* f!  LOOP
```

Dann geht's weiter zur Berechnung der Faktorkurve:

```
: mellow-A ( --)
    r/m_ :dflat-A* f@      \ deflate 100 -> zero
    r/v_ :mellow-A* f@ f*  \ damping value
    r/m_ :dflat-A* fdup f! \ update dflat
         100e fswap f-     \ upside down
         100e f/           \ make it <1
    r/m_ :mel-A* f! ;      \ mellow factor
```

Aufgerufen wird dieses Wort durch folgende bereits bekannte, nun aber aufgebohrte Routine:

```
\            <<< TONGENERATOR (A) >>>
: waver-A  ( --)
        empty-accu-A  \ AKKU LEEREN
  v/m 0 DO   i to v/m_ \   6 voices per manual
  r/v 0  DO  i to r/v_ \   3 p-rows per voice
  p/r 0   DO i to p/r_ \  16 partials per p-row
\ r/m                  \  18 p-rows per manual
\ p/m                  \ 288 partials per manual
                 v/m_ r/v * r/v_ + to r/m_
                 v/m_ p/v * r/v_ p/r * +
                          p/r_ + to p/m_
              sinus-A       \ KURVENBERECHNUNG
              fadings-A     \ ABKLINGVORGÄNGE
          LOOP mellow-A     \ WEICHER EINSATZ
         LOOP  phase-incr-A \ PHASE INKREMENT
        LOOP   timer-decr-A \ TIMER DEKREMENT
               store-accu-A ; \ AKKU UPDATE
```

Man sieht: Kurvenberechnung und Abklingvorgänge werden für jeden der 288 Teiltöne des *Manuals* aufgerufen, der weiche Toneinsatz dagegen nur für die 18 Teiltonreihen.

Die Abklingvorgänge sind von der Erweiterung nicht betroffen, wohl aber die Kurvenberechnung. Darin taucht nun `:mel-A*` als Faktor auf und verkleinert somit die Amplitude während des Tonanfangs:

```
\            <<< KURVENBERECHNUNG (A) >>>
: sinus-A  ( --)
  p/m_ :ampl-A* f@ f0> \ avoid multiply by zero
IF
  v/m_ :phase-AL* f@   \ root's phase (L)
  p/r_ 1+ s>f f*       \ multiply by partial#
         fsin          \ sinus (L)
  p/m_ :ampl-A* f@ f*  \ multiply by ampl
  r/m_ :mel-A* f@ f*   \ multiply by mellow
         accu-AL* f+!  \ update accu (L)

         stereo_    IF \ second channel (R)
  v/m_ :phase-AR* f@   \ root's phase (R)
  p/r_ 1+ s>f f*       \ multiply by partial#
         fsin          \ sinus (R)
  p/m_ :ampl-A* f@ f*  \ multiply by ampl
  r/m_ :mel-A* f@ f*   \ multiply by mellow
         accu-AR* f+!  \ update accu (R)
                  THEN
THEN ;
```

Mit diesen Ergänzungen sind bei passender Vorwahl sanfte Einschwinger zu hören:

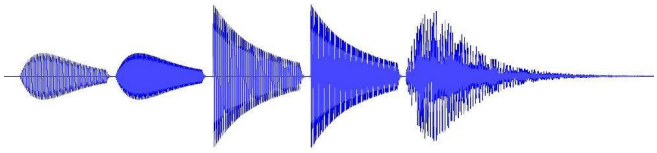


Abbildung 3: Harte und weiche Einschwingvorgänge

Wird dann noch das Fading abgeschaltet, fehlt für ordentliche Orgeltöne nur noch das Anblasgeräusch.

Ziehen der Tonhöhe im Prinzip

Der zweite unerfüllte Wunsch beim Verfassen von *Top One* war die **bender**-Funktion, das Ziehen der Tonhöhe. (*Les Paul* ist ohne diesen Effekt gar nicht vorstellbar.)

Ich werde mich für diesmal auf die schematischen Zeichnungen beschränken, denn ich möchte noch ein weiteres Thema unterbringen, damit das Gesamtbild der Wave Engine Gestalt annimmt.

Die **bender**-Funktion ist aufwändiger als der weiche Toneinsatz. Ein kurzes Hochziehen zur Nenn-Tonhöhe reicht nicht, denn es ist auch so etwas wie „in die Knie gehen“ erwünscht, also ein Absenken und langsame Rückkehr zur Nenn-Tonhöhe.

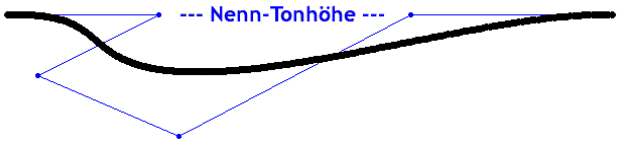


Abbildung 4: „Magische Punkte“ à la Bézier

Abb. 4 zeigt eine Bézier-Kurve (zur Konstruktion siehe Wikipedia) mit der gewünschten Form. Leider fehlt hier eine Zeit-Achse; die Kurve müsste zwischengespeichert werden, um sie für den zeitlichen Verlauf zu nutzen, was einen unnötigen Speicherbedarf zur Folge hätte.

Es muss auch anders gehen. Die abklingende Kurve aus **Abb. 2** strebt nach der Nulllinie. Was wir hier brauchen, ist ein Streben nach vorgegebenen Werten (wie die Rechteckkurve in **Abb. 5**).

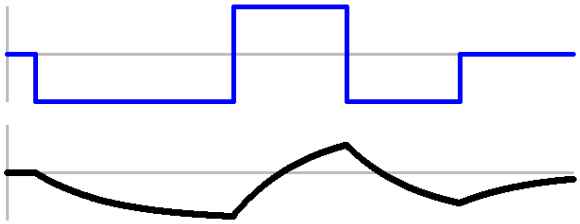


Abbildung 5: Kurve folgt (träge) den Vorgaben

Abb. 5 zeigt die Funktion: Die Kurve bemüht sich mit der ihr eigenen Trägheit, den Vorgaben zu folgen. Mit geeigneteren Vorgaben verschwinden die Knicke, **Abb. 6**:

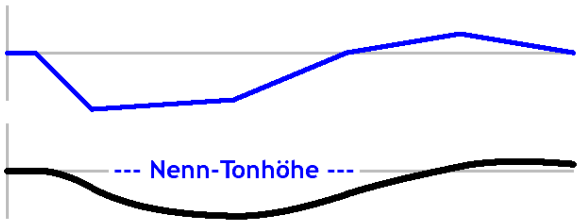


Abbildung 6: Die gewünschte Tonhöhen-Kurve

Diese Kurve muss nur noch dem Sinus-Generator angedient werden. (Dazu später mehr.)

Draft-Betriebsart

Man könnte vielleicht einwenden, dass der viele Aufwand den *Waver* unnötig träge werden lässt. Aber da er von vornherein nicht für Echtzeit konzipiert war, also keinen Klaviatur-Anschluss vorsieht, geht es allenfalls um das Warten aufs Ergebnis nach Programmstart. Hier kann durch eine Draft-Betriebsart viel Zeit eingespart werden. Wie schon aus der Wave-Datei ersichtlich, kann die Auflösung von 44 auf 11 Kiloframes pro Sekunde gedrosselt werden. Weiter kann auf die Berechnung höherer Teiltöne verzichtet werden sowie auf eine Reihe von Funktionen wie **mellow** und anderes. Es geht in aller Regel hauptsächlich darum, Fehler in der Abfolge der Noten aufzudecken. Rhythmisch heikle Stellen können zudem leicht für sich getestet werden, was nur wenig Zeit kostet.

Register-Syntax und Erkennung

```
\ === Partiturzeilen-Typen ===
\   fix{ ... }      ‘ Vorgaben (fest)
\   set{ ... }      ‘ Einstellungen (veränderbar)
\  regA{ ... }      ‘ Registrierung Manual A (ebenso B, C, D)
\     A{ ... }      ‘ Notenzeile für Manual A (ebenso B, C, D)
\     [< ... >]     ‘ Titelzeile: wird in die Wave-Datei übernommen
\     [[ ... ]]     ‘ Kommentar (auch mehrzeilig), blendet Code aus
\    [[[ ... ]]]    ‘ Kommentar (kann [[...]] beinhalten)
\
\ === Feste Vorgaben ===
\   fix{ dra=... } draft             fix{ ste=... } stereo
\
\ === Veränderbare Einstellungen ===
\   set{ man=... } manuals           set{ tem=... } tempo
\   set{ mic=... } micro             set{ tun=... } tuning
\
\ === Registrier-Anweisungen (alphabetisch) ===
\  regA{ acc=... } accent (2)       regA{ pa1=... } partials1 (16)
\  regA{ ang=... } angle (1)        regA{ pa2=... } partials2 (16)
\  regA{ arp=... } arpeggio (1)     regA{ pa3=... } partials3 (16)
\  regA{ bea=... } beat (1)         regA{ pre=... } preset (1)
\  regA{ ben=... } bender (12)      regA{ rit=... } ritard (2)
\  regA{ cre=... } cescendo (2)     regA{ sta=... } staccato (2)
\  regA{ dri=... } drift (2)        regA{ sus=... } sustain (3)
\  regA{ ech=... } echo (2)         regA{ tre=... } tremolo (2)
\  regA{ fad=... } fade (3)         regA{ tra=... } transpo (1)
\  regA{ fo1=... } formant1 (6)     regA{ vib=... } vibrato (2)
\  regA{ fo2=... } formant2 (6)     regA{ vol=... } volume (1)
\  regA{ mel=... } mellow (3)       regA{ wah=... } wahwah (4)
\
\ === Aufbau der Records ===
\    0  pa1 --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
\   16  pa2 --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
\   32  pa3 --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
\   48  fad --- --- sus --- --- mel --- --- sta --- tra acc --- cre ---
\   64  fo1 --- --- --- --- --- fo2 --- --- --- --- --- wah --- --- ---
\   80  ben --- --- --- --- --- --- --- --- --- --- --- ech --- dri ---
\   96  vib --- tre --- bea ang arp rit --- vol pre

\ === Offsets für den Zugriff in den Records ===
  00 constant pa1    16 constant pa2    32 constant pa3    48 constant fad
  51 constant sus    54 constant mel    57 constant sta    59 constant tra
  60 constant acc    62 constant cre    64 constant fo1    70 constant fo2
  76 constant wah    80 constant ben    92 constant ech    94 constant dri
  96 constant vib    98 constant tre   100 constant bea   101 constant ang
 102 constant arp   103 constant rit   105 constant vol   106 constant pre

\ Beispiel: Aktion für Register-Kommando "echo=" (noch nicht implementiert)
: ech:  ( a u -- a’ u’ f)  cr ." ### echo= ###"  true ;

\ === Zulässige Register-Kommandos ===
\ Hinweis: Hier reicht die voreingestellte Stacktiefe nicht ganz aus.
: regsym$          ( regular)                    ( short)
        [’] pa1:   c" partials="       [’] pa1:   c" par="
        [’] pa1:   c" partials1="      [’] pa1:   c" pa1="
        [’] pa2:   c" partials2="      [’] pa2:   c" pa2="
        [’] pa3:   c" partials3="      [’] pa3:   c" pa3="
        [’] fad:   c" fading="         [’] fad:   c" fad="
        [’] sus:   c" sustain="        [’] sus:   c" sus="
```

```
        ['] mel:   c" mellow="         ['] mel:   c" mel="
        ['] sta:   c" staccato="       ['] sta:   c" sta="
        ['] tra:   c" transpo="        ['] tra:   c" tra="
        ['] acc:   c" accent="         ['] acc:   c" acc="
        ['] cre:   c" crescendo="      ['] cre:   c" cre="
        ['] fo1:   c" formants="       ['] fo1:   c" for="
        ['] fo1:   c" formant1="       ['] fo1:   c" fo1="
        ['] fo2:   c" formant2="       ['] fo2:   c" fo2="
        ['] wah:   c" wahwah="         ['] wah:   c" wah="
        ['] ben:   c" bender="         ['] ben:   c" ben="
        ['] ech:   c" echo="           ['] ech:   c" ech="
        ['] dri:   c" drift="          ['] dri:   c" dri="
        ['] vib:   c" vibrato="        ['] vib:   c" vib="
        ['] tre:   c" tremolo="        ['] tre:   c" tre="
        ['] bea:   c" beat="           ['] bea:   c" bea="
        ['] ang:   c" angle="          ['] ang:   c" ang="
        ['] arp:   c" arpeggio="       ['] arp:   c" arp="
        ['] rit:   c" ritard="         ['] rit:   c" rit="
        ['] vol:   c" volume="         ['] vol:   c" vol="
        ['] pre:   c" pre="            ['] pre:   c" preset="
        ['] reg_dummybar  c" |"        ['] rem:   c" `" ( skip line)
        ['] reg_brace-right  c" }" ;

create regsym regsym$
            , ,  , ,  , ,  , ,  , ,     , ,  , ,  , ,  , ,  , ,  ( 10)
            , ,  , ,  , ,  , ,  , ,     , ,  , ,  , ,  , ,  , ,  ( 20)
            , ,  , ,  , ,  , ,  , ,     , ,  , ,  , ,  , ,  , ,  ( 30)
            , ,  , ,  , ,  , ,  , ,     , ,  , ,  , ,  , ,  , ,  ( 40)
            , ,  , ,  , ,  , ,  , ,     , ,  , ,  , ,  , ,  , ,  ( 50)
            , ,  , ,  , ,  , ,  , ,     0 ,  ( sentinel)

\ Beispiel für den Zugriff:
\ s" beat=123" regsym extract-token ( string count token flag)
\ .s <4> 161889205 7 3054668 -1  ok
\ "3054668 PERFORM" führt die gewünschte "beat"-Handlung aus.
\ "161889205 7 type" gibt die restliche Zeichenkette ("123") aus.
\ Wird kein Symbol gefunden (flag=0), wird die ganze Kette ausgegeben.

\ === Partitur: Komplette Registrierung für Record 1 ===
\     Auswahl und Reihenfolge beliebig.
\     Alle Null-Angaben durch "preset=0" überflüssig.

regA{ *1 partials=
      (1:5000  2:   0  3:   0  4:   0  5:   0  6:   0  7:   0  8:   0
       9:   0 10:   0 11:   0 12:   0 13:   0 14:   0 15:   0 16:   0),
      (1:3000  2:1000  3: 500  4: 300  5: 200  5: 100  7:   0  8:   0
       9:   0 10:   0 11:   0 12:   0 13:   0 14:   0 15:   0 16:   0),
      (1:   0  2:   0  3:1200  4: 500  5:   0  6: 400  7:   0  8: 200
       9:   0  10:  0 11:   0 12:   0 13:   0 14:   0 15:   0 16:   0);
      fading=8,20,40;                 ‘ 3 x Abklingen
      sustain=400,400,400;            ‘ 3 x Nachklingen
      mellow=0,0,0;                   ‘ 3 x weicher Einsatz
      staccato=1000,100;              ‘ Dämpfung, Vorlaufzeit
      transpo=0;                      ‘ Tonart-Transponierung
      accent=115,3;                   ‘ Lautstärke in Prozent, Töneanzahl
      crescendo=0,0;                  ‘ Zeit, Wert
      formant1=0,0,0,0,0,0;           ‘ 2 x Frequenz, Amplitude, Breite
      formant2=0,0,0,0 0 0;           ‘ 2 x Frequenz, Amplitude, Breite
      wahwah=0,0,0,0;                 ‘ 1|2, Affinität, Steilheit, Dots
      bender=0,0,0,0,0,0,0,0,0,0,0,0  ‘ Anfangswert, Affinität, xy-Paare
```

```
      echo=0,0;                     ‘ Zeit, Wert
      drift=0,0;                    ‘ Verstimmung des Manuals
      vibrato=0,0;                  ‘ Frequenzvibrato
      tremolo=0,0;                  ‘ Amplitudenvibrato
      beat=0;                       ‘ Phasenvibrato (links-reechts)
      angle=0;                      ‘ Stereo-Position
      arpeggio=0;                   ‘ Akkordbrechung (Dauer)
      ritard=0,0;                   ‘ langsamer werden (ritardando)
      volume=0;                     ‘ Lautstärke-Korrektur (0=neutral)
      preset=0;                     ‘ Voreinstellung (zuerst gelesen)
}

\ === Partitur: Registrierung der Teiltöne wie oben in Kurzschreibweise ===

regA{ *2 preset=0;
        par=(5000),(3000 1000 500 300 200 100),(3:1200 500 6:400 8:200); }
```

Von der Partitur zum Generator

Der Weg von den Musiknoten bis zur Generierung der Töne ist nicht ganz trivial. Um für diesmal bis zum *Generator* vorzudringen, gehe ich in der Beschreibung vom Ergebnis des Partitur-Interpreters aus, sonst verheddern wir uns in den Interpunktionen.

Ich spreche jetzt nicht von den zahlreichen Einstellungen und Registrierungen, sondern nur von *Tonhöhe*, *Tondauer* und *Habitus*. Das sind die drei Datentypen, die gepuffert werden müssen, weil sie im *Generator* für vier *Manuale* zugleich bereit stehen müssen, obgleich sie in der Partitur über mehrere Zeilen verteilt sind.

Der dem Interpreter nachgeschaltete *Collector* packt die Daten in das Format, in dem sie durch den Pufferspeicher transportiert werden. Eine Speicherebene enthält 8 Halbzellen. Ich habe ein (nicht standardisiertes) 16-bit-Format gewählt. Mit geringerem Speicherbedarf kann ich das kaum begründen. In meiner beruflichen „Karriere“ aber war er fast immer ein Thema, und das hängt mir halt noch nach.

Den *FIFO*-Speicher laden

Mit den Befehlen `habit!`, `durat!` und `pitch!` werden die Daten im *Collector* gesammelt und mit `coll>fifo` zum *FIFO* gesandt. *FIFO* („first in – first out“) bedeutet, dass es um einen Ringspeicher geht, der ständig bereit ist, Daten anzunehmen und abzugeben, so lange es nicht zu overflow oder underflow kommt.

```
0 value coll-N_    \ address of actual collector

\ Load actual collector with habit
: habit!    ( n --)
    dup 31 > IF |HABIT!| THEN  /error
                $001F and  ( 5 bit)
                coll-N_ w@ $FFE0 and or
                coll-N_ w! ;

\ Load actual collector with durat
: durat!    ( n --)  coll-N_ 2 + w! ;

\ Load actual collector with pitch
: pitch!    ( n --)
           bracketcount @ 0=      \ [accord]?
    IF    coll-N_ cell+ 12 0 fill \ no
          1 voice# !              \ choose voc #1
    ELSE  voice# @ 5 >            \ [accord ]
                IF |VOICE| THEN   \ too many vocs
    THEN  coll-N_ cell+     \ skip habit & durat
          voice# @ 1 2* + w!      \ write pitch
          bracketcount @          \ [accord]?
    IF    voice# incr             \ next voice
    THEN ;

\ Find counter address (given fifo address)
: >cnt-in    ( fifo-a -- in-a  )      ;
: >cnt-lock  ( fifo-a -- lock-a)  2 + ;
: >cnt-out   ( fifo-a -- out-a )  4 + ;

\ Find plain address due to in-counter
: >man-in  ( fifo-a -- plain-a)
    dup  >cnt-in @  no-of-plains mod ( a plain#)
    swap  man-plain ;                ( plain-a)

\ Transfer data from coll-A to fifo-A
: coll>fifo-A  ( --)
    fifo-A >cnt-in w@  fifo-A >cnt-lock w@ -
    nr-of-plains >=  IF  |FIFO-OFL|  THEN
    coll-A fifo-A >man-in 16 cmove \ transfer
    fifo-A >cnt-in incr ;
```

Adressierung des FIFO-Ringspeichers

Im Kopf jedes der vier (den *Manualen* zugeordneten) Pufferspeicher sind drei Zähler untergebracht (**Abb. 8**), welche die Adressierung der aktuellen 16-bit-Pufferebenen übernehmen. `cnt-in` dient als Zeiger auf die zuletzt beschriebene Ebene, `cnt-out` als Zeiger auf die nächst zu lesende Ebene, und `cnt-lock` verhindert, dass im Falle von Wiederholungsschleifen noch gebrauchte Daten

überschrieben werden. Dabei ist zu beachten: `cnt-out` darf `cnt-in` nicht überholen (underflow), `cnt-in` darf `cnt-lock` nicht überholen (overflow); `cnt-lock` folgt normalerweise `cnt-out`, bleibt jedoch bei Schleifenbildung am Schleifenanfang hängen.

Das mag soweit plausibel klingen, aber wer überholt in einem Ringpuffer wen? Wer ist vorn und wer ist hinten? Um hier klare Verhältnisse zu bekommen, laufen die Zähler linear von 0 bis maximal 65535 und werden erst „modulo Puffertiefe“ zu Zeigern. Gezählt werden „Töne“, und dazu dürfte die Zählerkapazität reichen.

Abb. 7 stellt beispielhaft einen Ringpuffer mit 10 Ebenen Tiefe dar. Gezählt wird zwar fortlaufend, aber adressiert werden nur die 10 Ebenen. Hier ist ersichtlich, dass `cnt-in` auf der Position von `cnt-lock` angekommen ist: Overflow! Man sieht: `cnt-in` darf `cnt-lock` nur maximal um die Puffertiefe vorauseilen, und obiger Satz muss heißen: `cnt-in` darf `cnt-lock` *plus Puffertiefe* nicht überholen.

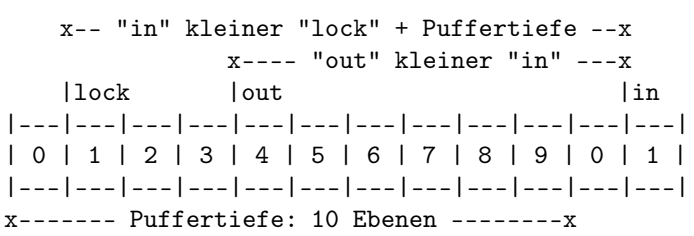


Abbildung 7: Overflow! „in“ hat „lock“ eingeholt.

Der Weg durch den Pufferspeicher

Abb. 8 zeigt den Weg der Daten vom *Collector* über den Pufferspeicher zum *Loader* (siehe **Abb. 3**). Letzterer (für jedes *Manual* separat vorhanden) versorgt den *Generator* direkt, so dass dieser auf die Daten aller beteiligten *Manuale* zugleich zugreifen kann. Dem *Loader* ist ein *Latch* (Zwischenpuffer) vorgeschaltet. Das ist erforderlich, damit während der aktuellen Generierung der Töne getestet werden kann, ob als nächstes ein Ton (Akkord) folgt oder eine Pause. Das entscheidet über die Art der Dämpfung (*Staccato* oder *Sustain*). Bei *Staccato* wird der aktuelle Ton verkürzt. (**Abb. 8** zeigt nur das *Latch*; der eigentliche *Loader* ist von gleicher Art und nachgeschaltet zu denken.)

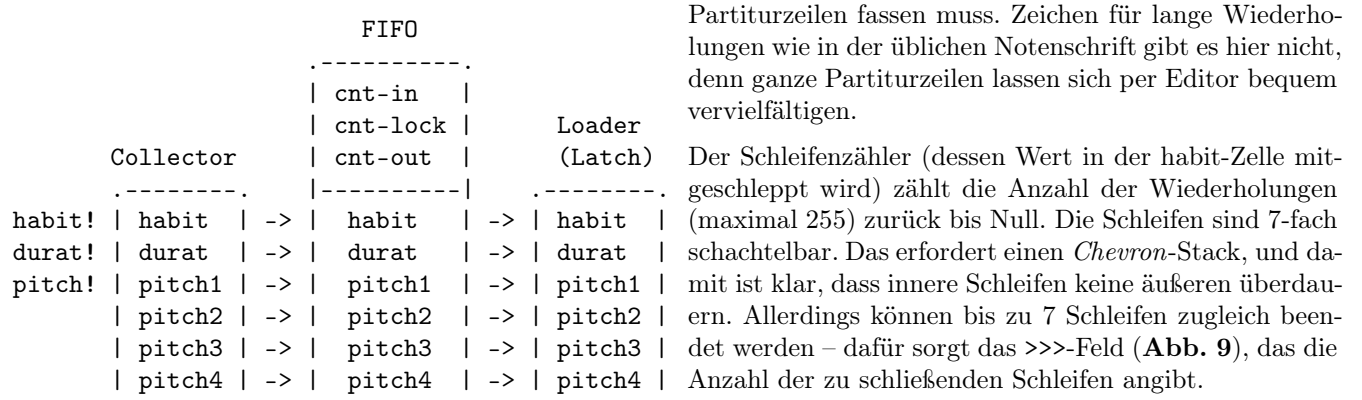


Abbildung 8: Daten passieren den *FIFO*-Speicher

Aus **Abb. 9** ist die Anordnung der transportierten Daten in acht 16-bit-Halbzellen ersichtlich. Die sechs Werte für die *Tonhöhe* (pitch1 bis pitch6) bewegen sich im Bereich 1 bis 478 (plus 0 für Pause und -1 für Tonverlängerung). Die *Tondauer* (durat) einer Viertelnote hat den Wert 720, der Maximalwert von 64800 entspricht 90 Viertelnoten oder 22,5 Viervierteltakten.

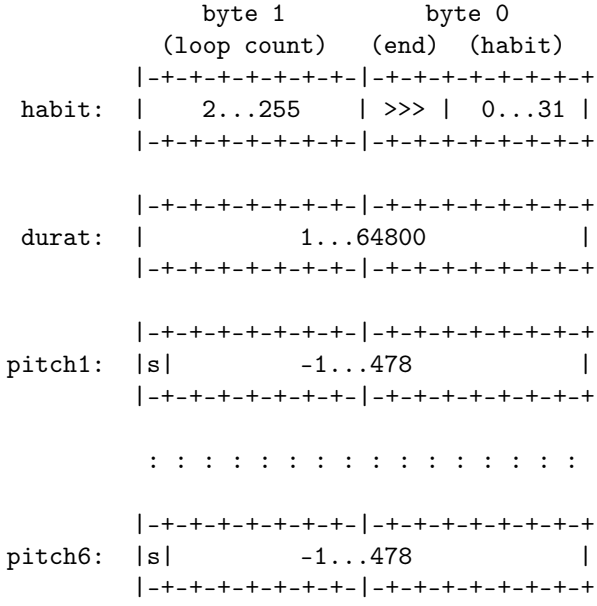


Abbildung 9: Eine *FIFO-Ebene im Detail*

Die 5 Bit rechts oben sind dem *Habitus* vorbehalten, also den maximal 32 *Habitus*-Marken, die vordefinierte Registrierungen aus den Records abrufen und in Gang setzen.

Wiederholungsschleifen

Kurze Wiederholungen werden dem *Generator* aufgebürdet, um den Pufferspeicher zu entlasten, da dieser ganze Partiturzeilen fassen muss. Zeichen für lange Wiederholungen wie in der üblichen Notenschrift gibt es hier nicht, denn ganze Partiturzeilen lassen sich per Editor bequem vervielfältigen.

Der Schleifenzähler (dessen Wert in der habit-Zelle mitgeschleppt wird) zählt die Anzahl der Wiederholungen (maximal 255) zurück bis Null. Die Schleifen sind 7-fach schachtelbar. Das erfordert einen *Chevron*-Stack, und damit ist klar, dass innere Schleifen keine äußeren überdauern. Allerdings können bis zu 7 Schleifen zugleich beendet werden – dafür sorgt das >>>-Feld (**Abb. 9**), das die Anzahl der zu schließenden Schleifen angibt.

Der Schleifenanfang wird durch `cnt-lock` bestimmt; das Schleifenende sitzt irgendwo zwischen `cnt-lock` und `cnt-in`. Der Zähler `cnt-out` bezeichnet die Pufferzelle, aus der gerade gelesen wird. Bei Wiederholungen wird er auf die Position von `cnt-lock` zurückgesetzt.

Wenn bei `cnt-lock` ein älteres Schleifenende anzutreffen wäre, gäbe es ein Problem. Damit dies nicht geschieht, müssen schließende *Chevrons* dem vorher notierten Ton mitgegeben werden. Deshalb werden „Töne“ aus dem *Collector* erst mit einem folgenden Partiturzeichen – z. B. auch dem Zeilenende (}), jedoch mit Ausnahme schließender *Chevrons* (> >> >>> etc.) – weitertransportiert.

Der Loader und die gap-Marke

Noch sind wir nicht beim *Generator* angekommen, darum noch dies: Der *Loader* übernimmt Daten aus dem *FIFO*-Puffer und stellt sie dem *Generator* zur Verfügung. Im *Generator* läuft für die Dauer des aktuellen Tons ein Countdown-Zähler (Timer). Bei Null wird der nächste Ton angefordert.

Wegen der *Staccato*-Funktion werden die Daten in einem Auffangregister (*Latch*) zwischengespeichert, ehe sie zum *Generator* gelangen. Im *Latch* kann erkannt werden, ob ein nächster Ton oder eine Pause ansteht. Im Fall eines folgenden Tons wird der aktuelle Ton vorzeitig abgedämpft. Die gap-Marke des *Timers* gibt die Größe des Vorlaufs an. Erreicht der *Timer* eines der beteiligten Kanäle die gap-Marke, so werden neue Daten aus dem *FIFO*-Puffer ins *Latch* übertragen und das gap-Flag gesetzt. Folgt (im *Latch*) eine Pause, so dauert der aktuelle Ton bis zum Zählerstand 0 an, worauf der Nachklang (*Sustain*) einsetzt. Folgt (im *Latch*) ein Ton, so werden bei Zählerstand 0 Daten vom *Latch* in den *Generator* übertragen. Das gap-Flag wird rückgesetzt.

Das gap-Flag ist erfordenlich, damit die *FIFO*-Daten nur einmal (auch noch nach Unterschreiten der gap-Marke) ausgelesen werden. Am Anfang wird das gap-Flag gesetzt (weil gap-Marker unterschritten) und Daten aus dem *FIFO* ins *Latch* übertragen. Von dort gelangen sie sofort (weil Zählerstand Null) in den *Generator* und der Zähler wird neu gesetzt. Mit Erreichen der gap-Marke gelangen nachfolgende Daten ins *Latch*, etc.

Zunächst wird ein *okay-Flag* = true gesetzt und die beteiligten Kanäle aufgerufen. Wenn einer der Kanäle die gap-Marke erreicht und keinen Nachschub im *FIFO*-Puffer vorfindet, wird das *okay-Flag* = false gesetzt und dadurch der *Generator* veranlasst, die Kontrolle an den Partitur-Scanner zurückzugeben.

```
: loader-A  ( f - f')
          requ-A_                \ request?
     IF   false to requ-A_       \ request off
          false to gap-A_        \ gap-Flag off
     THEN gap-A_ not             \ gap-Flag off?
IF        timer-A @ gaplim-A_ <= \ sentinel?
     IF   true to gap-A_         \ begin gap
          fifo>latch-A     and   \ update flag
THEN THEN timer-A @ 0=           \ timer=0?
     IF   false to gap-A_        \ end gap
          latch>gen-A      and   \ ld generator
     THEN ;
```

Symbole extrahieren

Und ich sollte noch `extract-token` zeigen, das im Abschnitt Register-Syntax Verwendung fand:

```
\ Extract a symbol from a string
: extract-token  ( $a u tab-a -- $a' u' token f)
            >r _leading r>
    BEGIN   >r over r@ @ count    ( a u a $a $u)
            tuck str= not         ( a u f)
    WHILE   r> 2 cells +          ( a u a")
            dup @ 0=              ( a u a" f)
        IF  drop 0 0  EXIT THEN   ( a u 0 0)
    REPEAT  r@ @ count nip        ( a u $u)
            rot over + -rot -     ( a" u")
            r> cell+ true ;       ( a" u" a' t)

\ Beispiel für den Zugriff:

: do-A ." <a>" ;
: do-B ." <b>" ;
: do-C ." <c>" ;

: tabel$  ['] do-A  c" A#"
          ['] do-B  c" B#"
          ['] do-C  c" C#" ;

  create tabel tabel$  , ,  , ,  , ,  0 ,

: do-it  ( a u --)  tabel extract-token
          IF PERFORM  ELSE drop ." ???" THEN
          ."  | " type ;

\   s" A#"     do-it  =>  <a> |  ok
\   s" B#C#"   do-it  =>  <b> | C# ok
\   s" B# C# " do-it  =>  <b> | C#  ok
\   s" A#B#C#" do-it  =>  <a> | B#C# ok
\   s" X#B#C#" do-it  =>  ??? | X#B#C# ok
```

Und damit genug für diesmal. Ich hoffe, dass das Gesamtkonzept ein bisschen deutlicher geworden ist. Der Umfang des *Waver*-Programms lässt nicht zu, in alle Details (der funktionierenden Teile) zu gehen. Aber bislang hat die Beschreibung dem Projekt recht gut getan, denn es geht deutlich flotter voran. Das letzte Heft (2011/4) füllte sich schneller als erwartet, so dass ich leider zu spät kam. Das soll mir nicht wieder passieren.

Beachten Sie bitte den Dateibereich der Website der Forth-Gesellschaft unter
http://www.forth-ev.de/filemgmt/viewcat.php?cid=54
sowie die Site des Autors unter
http://www.stocket.de/WE

IF-ELSE-THEN etc. als Colon-Definition ohne Assembler und ohne immediate

Fred Behringer

Dieser Artikel kann als Fortsetzung von [FB-4] betrachtet werden. Es wird gezeigt, dass auch die Konstrukte if-then, if-else-then, begin-until und begin-while-repeat ohne Assembler, mit den einfachsten Primitives aus [WS09] und ohne die Immediate-Eigenschaft als Colon-Definitionen aufgebaut werden können.

Seien `aaa` und `bbb` Forth-Worte beliebiger Bauart. Als Beispiel darf ich

```
: aaa 4 . 5 . 6 . ;
: bbb 7 . 8 . 9 . ;
```

verwenden. Selbstverständlich lässt sich beispielsweise die Konstruktion `if aaa else bbb then` bilden und verwenden - aber nicht interpretativ! Also irgendwie nur wie folgt:

```
: www if aaa else bbb then ;
```

Dabei ist es möglich, auf die klammernden Worte `aaa` und `bbb` zu verzichten. Also auch:

```
: www if 4 . 5 . 6 . else 7 . 8 . 9 . then ;
```

zu schreiben. Soweit die übliche Forth-Syntax (Weiteres beispielsweise zum ANS-Forth-Standard siehe [AF94]). Solche Konstruktionen sind ein typisches Beispiel für die Verwendung von Immediate-Worten, also von Worten, die schon während der Compilation ausgeführt werden (ausgeführt werden müssen). Dabei ist es für Rückwärtssprünge der Art von begin-again gar nicht nötig, das Sprungziel schon während der Compilation zur Verfügung zu haben. Compiliert wird „von links nach rechts“ und man kann sich erlauben, das Rückwärts-Ziel erst unmittelbar vor der Ausführung der `begin-again`-Konstruktion einzubringen. Das heißt, man braucht für diesen Zweck gar kein Immediate-Wort. Von dieser Erkenntnis habe ich für begin-again zum ersten Mal bei Coos Haak [CH93] Notiz genommen. Ähnliches gilt für den Einbau von Assembler-Befehlen. Mit anderen Worten, `begin-again` kann rein in High-Level-Forth konstruiert werden. Ich habe in [FB-4] versucht, diesen Gedanken auszudehnen, und mir die Frage gestellt, ob es in Forth vielleicht sogar möglich ist, ganz ohne Immediate-Worte auszukommen.

Etwas schwieriger als bei Rückwärtssprüngen ist es bei Vorwärtssprüngen. Wann soll der Sprungziel-Parameter eingebaut werden (Einpass-Compilation von links nach rechts)? Zur Ausführungszeit (Runtime) ist es zu spät. Man braucht ja die Sprungweite zum Beispiel bei `if-then` schon gleich am Anfang der Konstruktion, wenn `if` noch in Bearbeitung ist und der Sprung gegebenenfalls ausgeführt werden soll.

Wenn ich nun aber das Mark des Kontroll-Gerüstes `if-else` in ein klamerndes Wort stecke, also wenn ich `if aaa else` schreibe (Ähnliches bei `else bbb then` usw.), dann sieht die Situation anders aus. Die Wirkungsweise mit Klammerwort ist genau dieselbe wie ohne. Im Gegenteil, man kann ja auch mit dem herkömmlichen `if-else-then` die eben genannte Klammerung durch Hüllworte einführen, ohne dass sich am Geschehen etwas ändert! Aber mit dem Wort `aaa` steht sofort, gleich zu Anfang der Ausführung (Runtime), schon unmittelbar nach der Compilation, das Sprungziel, bezogen auf www, zur Verfügung, nämlich auf dem Returnstack als Adresse hinter `aaa` in der aufrufenden Colon-Definition, hier `www`.

Ich bin in [FB-4] davon ausgegangen, und tue das auch hier wieder, dass ein Compilator (Colon-Compiler) in Aktion treten darf, dass aber im Übrigen, ausgehend von einer beschränkten Zahl von Primitives, nur solche Worte (strikt als Colon-Definitionen) zum Aufbau des Systems zur Verfügung stehen, die zum jeweiligen Zeitpunkt schon definiert sind. Das Ganze läuft irgendwie auf eine Münchhausen-artige inkrementelle Metacompilation ohne expliziten Metacompiler hinaus.

Ich erhebe im vorliegenden Artikel, im Listing zumindest ab dem Wort `if`, folgende Modifikationsforderung an die Forth-Syntax:

```
: ww1 if aaa then                ;
: ww2 if aaa else bbb then       ;
: ww3 begin ccc while ddd repeat ;
```

`aaa bbb ccc ddd` sind ausführbare Forth-Worte. Alles, was „normalerweise“ zwischen `if` und `then` steht, muss in ein solches Forth-Wort, ich will es Hüllwort nennen, gepackt sein. Entsprechendes gelte für die Strukturpaare `if-else` und `else-then` und `while-repeat`. Im Innern eines solchen Strukturpaares darf jeweils nur immer ein einziges Hüllwort stehen. Das Hüllwort darf leer bleiben, es muss aber (als Worthülse) vorhanden sein. (Das Wort noop könnte beispielsweise als leeres Hüllwort dienen.) Zwischen `begin` und `while` ist kein Hüllwort nötig, zwischen `begin` und `until` auch nicht, zur Abrundung kann es aber auch dort gesetzt werden. Gleiches gilt für das schon in [FB-4] behandelte `begin-again` aus [CH93].

Zum Austesten verwende ich Turbo-Forth in der 16-Bit-Ausführung. Aus [FB-4] verwende ich ein paar Worte, die ich im Listing ohne Kommentar hier noch einmal anführe. Ich mache davon Gebrauch, dass ich in Forth (in Turbo-Forth auf jeden Fall) jedes Wort beliebig oft in den Quelltext schreiben kann, ohne dass das den Programmablauf prinzipiell stört. Die Beschaffung der Primitives ist nicht Untersuchungsgegenstand der vorliegenden Arbeit. Selbstverständlich beschaffe ich mir (für die

Beispiele und für Testzwecke) sämtliche benötigten Primitives (für die relevanten Textstellen sind das `dup r@ r> >r drop d+ - + swap`) über den in Turbo-Forth eingebauten Assembler.

Mit anderen Worten: Zum Ausprobieren lege ich nicht nur den Colon-Compiler, sondern das gesamte Turbo-Forth-Basissystem zu Grunde. Sollte ich dabei unbeabsichtigt über das Ziel hinausgeschossen sein, bitte ich freundlich um Hinweise.

Literatur

[AF94] ANS-Forth-Standard: The American National Standard for the Forth language (ANSI X3J14:1994).

[CH93] Haak, Coos: 'Lusstructuren maken zonder IMMEDIATE'. Vijgeblad 43 (1993), S.10. Auch unter http://www.forth.hccnet.nl/vijgebladarchief/ - 118k .

[FB-1] Behringer, Fred: Über Flags in Forth. Vierte Dimension 1/2011, S.33-34.

[FB-4] Behringer, Fred: Kontrollstrukturen als Colon-Definitionen und ohne die Immediate-Eigenschaft?. Vierte Dimension 4/2011, S. 35-40.

[WS09] Stricker, Willi: Minimaler Basis-Befehlssatz für ein Forth-System. Vierte Dimension 3/2009, S.15-17.

Listing

```
1  hex                      \ Alle Wertangaben sind hexadezimal zu verstehen.
2
3  \ In [FB-4] hatte ich die folgenden Kontrollstruktur-Elemente von Coos Haak
4  \ [CH93] verwendet, die ohne 'immediate' und ohne die Zuhilfenahme von
5  \ Assembler auskommen und die mit den allereinfachsten Primitives drop r> r@
6  \ >r arbeiten:
7
8  : begin ( -- ) r@ >r   ; \ Kopiert (in der Colon-Definition, die die Schleife
9                           \ enthaelt) die Adresse nach begin auf den R-Stack.
10 : again ( -- ) r> drop   \ Ersetzt auf dem R-Stack die Adresse nach again durch
11                r@ >r   ; \ die Adresse nach begin, springt also dann dorthin.
12 : exit  ( -- ) r> drop   \ Entfernt die Adresse nach begin vom Returnstack und
13                r> drop ; \ wirkt dann genau so wie das urspruengliche exit.
14
15 \ In [FB-4] hatte ich im Uebrigen geschrieben:
16
17 \ "Schleifen, die sich nur auf schon durchlaufene Stellen im Quelltext
18 \ beziehen, muessten eigentlich alle nach dem hier besprochenen Schema
19 \ behandelbar sein. Dazu gehoeren (neben dem oben erwaehnten begin-again
20 \ aus [CH93]) auch begin-until und begin-while-repeat. Bei begin-until
21 \ haette ich es beinah schon geschafft. Aber nur beinahe! Die weitere
22 \ Beschaeftigungsrichtung ist damit jedoch bereits vorgegeben."
23
24 \ Nun, im vorliegenden Artikel moechte ich ueber das inzwischen Erreichte
25 \ berichten. Die hier verwendete Returnstack-Akrobatik ist 'unendlich
26 \ schwoer', aber Forth macht es auch einem Nicht-Guru moeglich, sich daran zu
27 \ versuchen - zugegeben, manchmal mit fast nicht mehr vertretbarem Aufwand an
28 \ Geduld.
29
30 \ Ich war jedoch in meinen Erwartungen zu optimistisch. Bei begin-until habe
31 \ ich es inzwischen geschafft - ich werde gleich darueber berichten.
32 \ begin-while-repeat jedoch enthaelt einen Teil, bei welchem ein
```

```
\ Vorwaerts-Sprung-Ziel bekannt sein muss, das man sich nach meiner jetzigen
\ Sicht nicht anders beschaffen kann als schon waehrend der Compilation.
\ Dasselbe Problem besteht bei if-then und if-else-then. Ich habe mir dazu
\ einen Trick ueberlegt. Zunaechst aber darf ich begin-until erledigen.

\ Bevor ich mit meinen Vorschlaegen zu begin-until beginne, schnell diejenigen
\ Worte in Kurzdarstellung, die ich schon in [FB-4] vorgeschlagen hatte und
\ von denen ich auch hier wieder Gebrauch machen moechte.

\ begin und again aus [CH93] habe ich gerade erwaehnt. Sodann brauche ich:

: 0= ( n1 -- n2 )          \ n1=0 -> n2=-1 ; sonst n2=0
     0 -1 0 d+ swap drop 1 - ;

: ?branch2 ( fl -- )       \ Sprung um 2 Byte bei fl=0, kein Sprung sonst.
     0= 0 -1 0 d+ swap drop dup + r> + >r ;

code d+ ( d1 d2 -- d1+d2 ) \ Das habe ich in [FB-1] vorgeschlagen.
     ax pop  bx pop  cx pop  dx pop  cx push  dx push  ax push  bx push
      66 c,  ax pop   66 c,  bx pop    66 c,  ax bx add   66 c, bx push
             ax pop  bx pop  ax push bx push  next end-code

\ Und nun zu begin-until. Dazu fuehre ich -begin und (again) ein:

: -begin ( -- )  \ Dieses -begin entfernt die von (again) in until stammende
                 \ Ruecksprung-Adresse vom Returnstack.
     r> r>       \ Zwei Returnadressen-Ebenen tiefer springen.
     r> drop     \ Dort Adresse nach begin vom R-Stack nehmen und die noch auf
     >r >r ;     \ dem D-Stack geparkten Rsp-Adressen wieder auf den R-Stack
                 \ legen (zum Ruecksprung zur aufrufenden Colon-Definition).

: (again) ( -- ) \ Dieses (again) wird ausgefuehrt, wenn in until fl=0 ist.
     r>          \ Ruecksprung-Adresse auf dem Datenstack parken.
     r> drop     \ Beim Ausfuehren wieder zur Adresse nach begin springen (was
     r@ >r       \ also dem reinen Schleifenwiederholer again entspricht).
     2 +         \ Geparkte Ruecksprung-Adresse um 2 erhoeht auf den R-Stack
     >r ;        \ legen, so dass dann nach Rueckkehr zu until das Wort -begin
                 \ uebersprungen und also nicht mehr ausgefuehrt wird.

: until ( fl -- ) \ Das Wort 0= wirkt auf das ?branch2 als logische Negation.
     0=          \ Wenn bei diesem until (in der Colon-Definition) fl<>0 ist,
     ?branch2    \ dann wird das folgende (again) uebersprungen und -begin
     (again)     \ ausgefuehrt (in der Colon-Definition weiter nach until).
     -begin ;    \ Wenn dagegen fl=0 ist, dann wird das (again) ausgefuehrt
                 \ und zur Adresse nach begin zurueckgesprungen.

\ Die Bezeichnung (again), mit den Klammern, ist eine Verlegenheitsloesung, da
\ die Bezeichnung again schon fuer das herkoemmliche begin-again vergeben ist.
\ Ueberlegenswert waere es, ob man nicht mit der Funktion des Wortes again
\ auch hier durchkommt.

\ Es folgt ein Beispiel fuer begin-until.

: yy1 0 begin 1 + dup . dup 47 = until drop  cr ." Das war's." ;
: xx1 cr ." Beispiel 1: " cr yy1 cr ;

\ Und wie war das eigentlich mit if-then (ohne Assembler und ohne immediate)?
\ Es sollte keine Moeglichkeit bestehen, (schon waehrend der Compilation des
\ Quelltextes) Immediate-Worte auszufuehren. Wie soll dann aber das Sprungziel
\ fuer if bei fl=0 schon am Anfang von if-then bereitgestellt werden? Es ist
```

```
\ wirklich schwierig! Hier ein Kompromissvorschlag zur Loesung des Konflikts:

\ if-then kapselt den von Fall zu Fall (bei nicht erfuellter if-Bedingung) zu
\ ueberspringenden Quelltextteil ein. Eine ganz einfache Einkapselung, die
\ gleichzeitig schon gleich nach der Compilation das Sprungziel bereitstellt
\ (also wenn das Compilat bereits vorliegt, aber vom Runtime-Durchgang noch
\ keine Rede ist), kann ueber ein Forth-Wort erzielt werden, das man extra zu
\ diesem Zweck in die aufrufende Colon-Definition einbringt. (Der Sprung - so
\ er ueberhaupt anfaellt - findet ja in der rufenden Colon-Definition statt.)
\ Der Colon-Compiler tut uns den Gefallen, die Adresse unmittelbar hinter dem
\ extra eingefuehrten Wort (in der Colon-Definition) schon zur Compile-Zeit
\ als Rueecksprung-Adresse auf den Returnstack zu legen.

\ Ich beginne mit if-then, benoetige aber gleich noch einen Nebengedanken,
\ wenn ich if in ueblicher Art auch fuer if-else-then verwenden moechte.

\ Aufgrund der Syntax-Modifikation (siehe Textteil), von welcher ich hoffe,
\ dass sie sich im Endeffekt als nicht ganz so eigenwillig herausstellt, wie
\ sie zunaechst erscheinen mag, braucht zu keinem Zeitpunkt ein Sprungziel
\ explizit bekannt zu sein. Gesprungen wird (auf der entsprechenden Ebene der
\ Compilation in der Colon-Definition) nur immer um hoechstens ein einziges
\ Forth-Wort (ich will es Huellwort nennen). Die Sprungweite ist also immer
\ nur 2 oder noetigenfalls 0 (wenn naemlich fl=0), und das ist ein typischer
\ Anwendungsfall fuer meinen Vorschlag ?branch2. Das eigentliche Sprungziel
\ von if hin zu then oder von if hin zu else und dann von else hin zu then
\ kann als Rueecksprung-Adresse (per r> oder r@) dem Returnstack entnommen
\ werden.

\ ------------------------------------------------------------------
\ Achtung! Hier und im weiteren Verlauf moege gelten:
\ Huellworte duerfen leer sein, aber nicht einfach weggelassen werden.
\ ------------------------------------------------------------------

\ So und nicht anders haette ich mir das eigentlich vorgestellt. Mit ?branch2
\ komme ich aber leider nicht durch. Bei if-else-then wird entweder der Part
\ if-else durchlaufen und dann von else nach then gesprungen oder es wird von
\ if nach else gesprungen und der Part else-then durchlaufen. else muesste
\ genau wie if einen Flag-Wert tragen - und der Flagwert von else waere mit
\ dem Flagwert von if zu koppeln. Nun sind aber die Flagwerte von if und else
\ logisch komplementaer: entweder das eine oder aber das andere. Das nutze ich
\ aus.

\ Mit anderen Worten, mit ?branch2 komme ich nicht durch. Ich benoetige ein
\ Wort, das sich analog zu ?branch2 verhaelt, bei dem sich aber der Sprung,
\ wenn er stattfindet, ueber vier Byte (nicht nur ueber zwei) erstreckt. Ich
\ darf also konstruieren:

: ?branch4 ( fl -- )    \ Dieses Wort sorgt in einer Colon-Definition dafuer,
     0= 0 -1 0 d+ swap  \ dass die 2 Worte, die unmittelbar auf eben dieses
     drop dup + dup +   \ ?branch4 folgen, bei fl=0 uebersprungen werden.
     r> + >r ;          \ Bei fl<>0 erfolgt kein Sprung.

\ Die Sequenz dup + dup + haette natuerlich durch 4 * ersetzt werden koennen.
\ Ich moechte aber mit der gewaehlten Darstellung schon mal darauf aufmerksam
\ machen, dass ich ja bei meinen ganzen Ueberlegungen im Vorliegenden immer
\ davon ausgehe, dass das Einbringen von Ganzzahlen ins Compilat vom Compiler
\ klaglos erledigt wird.

\ Beim Ausprobieren beschraenke ich mich nicht auf den Colon-Compiler, sondern
\ mache vom gesamten (Turbo-)Forth-System Gebrauch. Ich mache mir also hier
```

```
153  \ insbesondere keine Gedanken ueber die Erfassung von (Ganz-)Zahlen durch das
154  \ System - eigentlich durch den Colon-Compiler.
155
156  \ Man koennte sich aber doch auch ueberlegen, ob nicht eventuell alle Zahlen
157  \ ueber dup und + und vielleicht noch ein weiteres Primitive (welches?)
158  \ erfasst werden koennen. Doch das ist ein anderes Thema, das hier nicht weiter
159  \ verfolgt werden soll.
160
161  \ Jetzt ein Beispiel:
162
163  : xx3 ?branch4 5 . 6 . 7 . 8 . ;
164  \ 9  0 xx3 [ret] 9 6 7 8 ok ( -- )
165  \ 9 -1 xx3 [ret] 5 6 7 8 ok ( -- 9 )
166
167  \ Ich habe das Beispiel mit Absicht komplizierter gestaltet, als es vielleicht
168  \ noetig gewesen waere. 0 und -1 sind die Flagwerte fuer ?branch4. Die 9 wird
169  \ bei fl=0 vom ersten Punkt konsumiert (und angezeigt), nachdem ja die 32 Bits
170  \ von (lit) 5 uebersprungen worden waren. Bei fl=-1 kommt die 5, die ja im
171  \ Compilat als (lit) 5 erscheint, zum Tragen und wird angezeigt; die 9 bleibt
172  \ dann auf dem Datenstack.
173
174  \ Fuer das jetzt folgende if-else-then schien es mir 'einfacher', nicht das
175  \ fertige Wort ?branch4 zu verwenden, sondern 'nur' dessen Wortkoerper explizit
176  \ und in voller Laenge direkt einzubauen. Damit bin ich dem Uebel aus dem Weg
177  \ gegangen, dass ich sonst mit unterschiedlichen Ruecksprung-Ebenen zu kaempfen
178  \ gehabt haette.
179
180  : if   ( fl -- ) 0= 0 -1 0 d+ swap drop dup + dup + r> + >r ;
181  : else ( -- ) r> 2 + >r ;
182  : then ( -- ) ;
183
184  \ Jetzt ein Beispiel fuer if-then und if-else-then mit Erlaeuterung. Es sei
185
186  : aaa 4 . 5 . 6 . ;
187  : bbb 7 . 8 . 9 . ;
188
189  \ Und hier das Beipiel mit den Huellworten aaa und bbb:
190
191  : ww1 if aaa          then ;
192  : ww2 if aaa else bbb then ;
193
194  \ In ww1 ist es sicher nicht auf Anhieb erkenntlich, warum das Kontrollwort if
195  \ auf Spruenge ueber 4 Byte gefasst sein muss - wenn naemlich fl=0. Der Grund
196  \ fuer Spruenge ueber 4 statt 2 Byte erhellt aus ww2. In der Konstruktion
197  \ if-else-then soll if natuerlich dasselbe if sein wie bei if-then. In if-then
198  \ richtet ein Sprung ueber 4 Byte (statt 2) keinen Schaden an: then ist ein
199  \ funktionsloses Wort.
200
201  \ Und wie ist es bei if-else-then (vergleiche ww2)? Wenn da if auf fl=0 trifft,
202  \ wird nicht nur aaa uebersprungen, sondern auch else. Fuer den Sprung ist das
203  \ Kontrollwort else aber ein genau so schoenes Forth-Wort wie aaa. Also wird
204  \ als Antwort auf fl=0 das Wort aaa ausgelassen, das Wort else uebersprungen
205  \ und dann direkt bei bbb fortgefahren. (Es lebe die Programmierfreiheit und
206  \ die Vermischbarkeit von System- und Anwendungsworten in Forth!)
207
208  \ Andererseits wird bei Aufruf von if mit fl<>0 erst das Wort aaa ausgefuehrt,
209  \ und dann das Kontrollwort else. Dass dann das Wort bbb uebersprungen wird,
210  \ ist ganz einfach dadurch zu erreichen, dass else (in der aufrufenden
211  \ Colon-Definition) einen Sprung ueber 2 Byte einleitet. Nichts anderes. Eine
212  \ Alternative gibt es im vorliegenden Kontrollwort-Geruest bei else nicht:
```

```
213  \ Waere beim Aufruf von if der Flagwert fl=0, dann wuerde mit aaa auch else
214  \ uebersprungen werden. else wuerde also fuer diesen Fall, naemlich fuer fl=0,
215  \ gar nicht erst in Betracht gezogen werden.
216
217  \ Es hat alles seinen Preis. Ich bezahle meine Syntax-Modifikation damit, dass
218  \ then im Compilat tatsaechlich auftritt. Das ist etwas gewoehnungsbeduerftig,
219  \ da das ja in der ueblichen Forth-Syntax nicht geschieht (Ueberpruefung per
220  \ see). Man kann aber auf das Dummy then nicht verzichten - wenn man das auf
221  \ 4-Byte-Spruenge eingerichtete if auch bei if-then (ohne else) verwenden will
222  \ (bei if-else-then waren 4-Byte-Spruenge noetig, um die Moeglichkeit, else
223  \ zusammen mit dem Huellwort aaa zu ueberspringen, auszuschoepfen).
224
225  \ Es gibt bei den im Vorliegenden besprochenen Kontrollworten im Compilat auch
226  \ kein ?branch. Und im Compilat zeigt das uebliche see die Worte if und else
227  \ an... und andere (kleinere oder groessere) Abweichungen vom Ueblichen.
228
229  \ Es folgt ein Beispiel, das nicht funktioniert (nachpruefen!). Was ist daran
230  \ falsch? Falsch im Sinne meiner Syntax-Modifikationen. Mit welch einfacher
231  \ Veraenderung bringt man es zum Laufen?
232
233  : ww3 if 4 if aaa then then 47 . ;
234
235  \ Ich muss zugeben, dass ich das Durchprobieren aller Moeglichkeiten (und
236  \ Unmoeglichkeiten) der Verschachtelungen nicht bis zum allerletzten denkbaren
237  \ Ende durchgefuehrt habe. Irgendwann muss aber jeder VD-Artikel auch mal sein
238  \ Ende finden - und wenn ich einen Schnitzer begangen haben sollte, waere ich
239  \ um Mitteilung dankbar.
240
241  \ Und hier noch ein funktionierendes komplexeres Beispiel, das sich aus den
242  \ bisher betrachteten (funktionierenden) Beispielen zusammensetzt. Ich fasse
243  \ es das eine Mal mit meiner Syntax-Modifikation und spreche es dann unter xx4
244  \ an, ein zweites Mal erfasse ich alles in der ueblichen Schreibweise des
245  \ if-else-then-Konstrukts unter xx5. Das Wort xx4 ist mit xx5 funktionsgleich
246  \ (nachpruefen!).
247
248  : xx4 ww2 ww1 xx1 cr ." Alles klar?" cr ;
249
250  : xx5
251     if aaa else bbb then
252     if aaa          then
253     cr ." Beispiel 1: " cr
254     0 begin 1 + dup . dup 47 = until drop  cr ." Das war's." cr
255     cr ." Alles klar?" cr ;
256
257  \ Sowohl xx4 wie auch xx5 koennen mit 4 verschiedenen Parameterpaaren
258  \ aufgerufen werden (0=false, -1=true):
259
260  \  0 -1 xx4/xx5 [ret]
261  \  0  0 xx4/xx5 [ret]
262  \ -1  0 xx4/xx5 [ret]
263  \ -1 -1 xx4/xx5 [ret]
264
265  \ Ich verzichte auf die Wiedergabe der Ergebnisse im Einzelnen.
266
267  \ Und jetzt die noch ausstehende Konstruktion begin-while-repeat . Mit meiner
268  \ Vermutung in [FB-4] habe ich mich leider vertan. Es ist denn doch nicht so
269  \ leicht wie bei begin-again und begin-until. Der Teil while-repeat entspricht,
270  \ wenn das Dazwischenstehende nicht ausgefuehrt wird, wenn also while mit fl=0
271  \ aufgerufen wird, einem Vorwaertssprung. Das leuchtet auch ein, wenn man
272  \ bedenkt, dass while in der herkoemmlichen Forth-Syntax nichts anderes bewirkt
```

```
273  \ als if. Und diese Aequivalenz von if und while kann ich auch, und werde ich
274  \ auch, hier ausnuetzen. Allerdings muss ich dazu meine Modifikationsforderung
275  \ an die Forth-Syntax ausdehnen - indem ich sie auch auf while-repeat anwende.
276
277  \ ----------------------------------------------------------------------------
278  \ Forderung: Die Forth-Bestandteile zwischen while und repeat muessen (wie bei
279  \ if-then und if-else und else-then) in ein Huellwort eingekapselt sein. Fuer
280  \ die Bestandteile zwischen begin und while ist ein Huellwort nicht noetig,
281  \ kann aber ebenso gut auch eingesetzt werden.
282  \ ----------------------------------------------------------------------------
283
284  \ Das fuer begin-while-repeat vorzusehende Wort begin bleibt dasselbe wie bei
285  \ begin-again und begin-until. Es legt die Adresse, die in der aufrufenden
286  \ Colon-Definition auf begin folgt, auf den Returnstack (siehe weiter oben im
287  \ Listing).
288
289  \ Und jetzt das Kontrollwort while. Wegen der Aequivalenz mit if (siehe weiter
290  \ oben im Listing) nimmt es nicht wunder, dass ich auch while mit dem
291  \ Wortkoerper von ?branch4 ausstatte - ?branch4, nicht ?branch2. Andererseits
292  \ muss bei while-repeat die mit begin eingeleitete Schleife (fuer while mit
293  \ fl=0) beruecksichtigt werden. Es nimmt also gleichermassen nicht wunder, wenn
294  \ ich mich auch stark an begin-until (siehe weiter oben im Listing) ausrichte.
295
296  : while ( fl -- )
297       0= 0 -1 0 d+ \ Diese, dann die zweite, die dritte und die fuenfte Zeile
298       swap drop    \ entsprechen dem Wortkoerper von ?branch4 (siehe unter
299       dup + dup +  \ if-else-then im Listing). dup + dup + entspricht 4 * .
300       dup          \ Duplikat von 0/4 zum Ausschalten von -begin bei fl<>0 .
301       r> + >r      \ Rueecksprung-Adresse + 4 Byte bei fl=0, + 0 Byte bei fl<>0.
302       ?branch2     \ -begin wird (in while) bei fl<>0 uebersprungen; bei fl=0
303       -begin ;     \ wird es ausgefuehrt.
304
305  : repeat ( -- )   \ Dieses repeat wird ausgefuehrt, wenn while auf fl<>0
306       r> drop      \ trifft. Es entspricht dem weiter oben besprochenen again:
307       r@ >r ;      \ Rueecksprung-Adresse vom R-Stack nehmen und durch Adresse
308                    \ nach begin in der aufrufenden Colon-Definition ersetzen.
309
310  \ Erlaeuterung: Sei
311
312  : ss0 begin 0 while aaa repeat 7 . ;
313  : ss1 begin 1 while bbb repeat 7 . ;
314
315  \ In ss0 wird das Paar aaa repeat uebersprungen und dann wird 7 . ausgefuehrt.
316  \ In ss1 wird das Wort bbb ausgefuehrt  und die Schleife dann ab 1 wiederholt.
317
318  \ Zur Abrundung ein verhaeltnismaessig einfaches Beispiel. Es stellt die Zahl
319  \ 8 durch mehrstufige Potenzierung der Basis 2 her. Natuerlich wird dieses
320  \ (viel zu umstaendliche) Beipiel nicht zur Anwendung empfohlen. Es soll nur
321  \ das Prinzip der begin-while-repeat-Schleife ohne Assembler und ohne immediate
322  \ erhellen. Man achte auf ccc als Huellwort! Ohne ein solches Huellwort geht
323  \ mein Vorschlag nicht. Mit dup + vermeide ich das Malzeichen bei 2 * .
324
325  : ccc dup + ;
326  : acht ( -- ) 1 begin dup 8 <> while ccc repeat . ;
327
328  \ Die Schleife ist bei repeat zu Ende. Der Anzeige-Punkt gehoert schon zur
329  \ Schleifen-Aussenwelt.
330
331  \ Vorsicht: Urspruenglich hatte ich beim Austesten der Schleife acht den
332  \ Bestandteil dup + ohne das eben eingesetzte Huellwort ccc wie folgt
```

```
\ geschrieben und mich darueber gewundert, dass das Ergebnis nicht meinen
\ Erwartungen entsprach:

\ : acht ( -- ) 1 begin dup 8 <> while dup + repeat . ;

\ Und nun noch das weiter oben im Listing (unter xx1/yy1) schon angefuehrte
\ Beispiel, diesmal aber ueber begin-while-repeat statt ueber begin-until
\ konstruiert.

: ddd 1 + dup . ;
: yy2 0 begin dup 47 < while ddd repeat drop cr ." Das war's." ;
: xx2 cr ." Beispiel 2: " cr yy2 cr ;
```

```
if (a == 1)
  if (b == 1)
    a = 42;
else
  b = 42;
```

```
if (a == 1)
{
  if(b == 1)
    a = 42;
}
else
  b = 42;
```

```
if a == 1:
    if b == 1:
        a = 42
else:
    b = 42
```

```
if [ $a -eq 1 ] ; then
  if [ $b -eq 1 ] ; then
    a=42
  fi
else
  b=42
fi
```

```
IF a = 1 THEN
   IF b = 1 THEN
      a := 42;
   END IF;
ELSE
    b := 42;
END IF;
```

```
IF a = 1 THEN
   IF b = 1 THEN a = 42
ELSE
    b = 42
END IF
```

Fossil — Quellcodearcheologie

Carsten Strotmann

Bei jedem langlebigen Softwareprojekt kommt irgendwann der Zeitpunkt, an dem der Entwickler in die Vergangenheit reisen möchte, um im Quellcode der Vergangenheit auf die Suche zu gehen: nach Fehlern, die sich unerkannt eingeschlichen haben; nach Funktionen, die in der Vergangenheit noch funktionierten oder nach dem einfachen Algorithmus, der doch viel besser war als der neue, clevere. In diesem Zeitpunkt freut sich der Entwickler, welcher den Quellcode seit Anbeginn in einer Softwareversionsverwaltung hat … wenn er diese denn hat …

Softwareversionsverwaltung

Auch der unorganisierteste Entwickler betreibt 'Ad–Hoc'–Versionsverwaltung. Diese 'Ad–Hoc'–Versionsverwaltung hat viele Gesichter, und wir alle habe diese schon selbst benutzt: Die mit Datumsstempel versehene Kopie des Quellcode-Verzeichnisses, die ZIP–Datei, die mehr oder weniger regelmäßig auch als Datensicherung erstellt wird: `software-v2-20120220.zip`.

'Ad–Hoc'–Versionsverwaltung ist nicht optimal, die Suche nach Änderungen und auch der Vergleich von Versionen ist manuell und mühsam. Kommerzielle– und auch Open–Source–Versionsverwaltungssysteme wie CVS, SVN, git oder Mercurial bieten eine ganze Reihe von Funktionen, sind aber umfangreiche Programmpakete, in die sich der Entwickler erst einarbeiten muss. Zusätzlich werden noch Webserver benötigt, eine spezielle Scriptsprache ist Bedingung für die Installation, und schnell ist ein Tag vergangen. Zu viel Aufwand für den Forth–Entwickler, er will ja schließlich die kostbare Zeit benutzen, um Forth zu schreiben, nicht um Softwareversionsverwaltung zu lernen.

Fossil

Die Software–Versionsverwaltungssoftware Fossil [1] hebt sich wohltuend von den übrigen SCM[1]–Werkzeugen ab und ist ganz im Forth-Sinne klein und effizient. Das fängt schon damit an, dass die Fossil–Installation eine einzelne ausführbare Binärdatei ist (fossil.exe unter Windows, fossil unter Unix/MacOS). Keine Installations-Prozedur erforderlich, sondern das Programm wird einfach an die Stelle auf der Festplatte kopiert, wo man sie denn haben möchte, und fertig. Die Fossil–Webseite bietet ausführbare Dateien für Linux, MacOS X und Windows, und den Quellcode zum Download an. Zum Betrieb von Fossil wird keine weitere Software benötigt, kein Webserver, keine Scriptsprache muss installiert sein, nur die Fossil–Programmdatei genügt.

Erste Schritte

Alle Quellcode–Dateien und die Änderungen an diesen Dateien, sowie alle anderen Metadaten, die Fossil speichert, werden pro Projekt in einer einzigen Datei gehalten, dem Projekt–Repository. Technisch gesehen ist das Projekt–Repository eine SQLite–Datenbank, aber dieser technische Hintergrund ist für die Benutzung von Fossil unwichtig.

Wenn wir ein neues Softwareprojekt unter Quellcodeverwaltung nehmen wollen, so müssen wir erst eine Repository–Datei erstellen. Die Repository–Datei kann in einem beliebigen Verzeichnis stehen, und kann auch später ohne Probleme umherkopiert oder auf einen anderen Rechner übertragen werden.

Ein Repository wird erstellt mit `fossil new <projektname>`. Nicht nur Quelltexte für Programmiersprachen können mit SCM–Systemen verwaltet werden, sondern auch Texte für die VD. Für diesen Artikel habe ich ein Repository namens 'vd-201201-fossil' angelegt:

[1] Source–Code–Management

Abbildung 1: Die *Timeline*–Ansicht in der Fossil–Web–Oberfläche

Abbildung 2: Die Detail–Ansicht für ein *Artifact* in der Fossil–Web–Oberfläche

```
# fossil new vd-201201-fossil
project-id: 071e05233e84654cc573cf79f6df21ebb05a24fa
server-id:  1dd3462892e01ecbaf62abd48de714d394328bb4
admin-user: cas (initial password is "abf70a")
```

Um mit einem Repository arbeiten zu können, müssen wir dieses öffnen:

```
# fossil open vd-201201-fossil
```

Der Befehl ‘fossil info’ gibt uns Informationen über das gerade geöffnete Repository:

```
# fossil info
project-name: <unnamed>
repository:   /home/cas/Documents/vd-201201-fossil
local-root:   /home/cas/Documents/
project-code: 071e05233e84654cc573cf79f6df21ebb05a24fa
checkout:     efc57e6bf2a94a74096cbd27a7860e2dae21bb24
              2012-02-22 02:30:22 UTC
tags:         trunk
comment:      initial empty check-in (user: cas)
```

Zum Anfang ist das Repository noch leer, wir müssen die Dateien hinzufügen, die wir gerne in der Quellcodeverwaltung haben möchten. Hierzu dient der Fossil–Befehl ‘add’. Mit ‘add’ können einzelne Dateien, aber auch ganze Dateibäume hinzugefügt werden. Wird dem ‘add’–Befehl ein Verzeichnis gegeben, so wird dieses Verzeichnis inklusive aller Unterverzeichnisse unter die Quellcodeverwaltung gestellt. In Falle des VD–Artikels ist es nur eine Datei:

```
# fossil add   fossil.tex
ADDED  fossil.tex
```

Die Datei ‘fossil.tex’ ist nun unter der Versionkontrolle, aber noch nicht in der Repository–Datenbank gespeichert. Erst wenn wir die Datei per ‘commit’ das erste Mal in das Fossil–System übernehmen, wird die Datei in das Repository gespeichert:

```
# fossil commit -m "Erste Version des fossil-Artikels"
New_Version: caf95931d089baaaef35cc70783613712da21c1b
```

Jede Änderung an einem Repository bekommt eine eindeutige kryptografische Checksumme, diese wird als Versionsnummer ausgegeben. Mit der Option ‘-m’ wird ein Kommentar zu dieser Änderung angegeben. Wird ‘-m’ weggelassen, so fragt Fossil nach einem Kommentar auf der Kommandozeile.

Nach ein paar Änderungen im Quellcode erfolgt ein weiterer Commit:

```
# fossil commit -m "Zweiter Commit"
New_Version: df0007f5e7259144b73dd98497d419a7f63f22b7
```

Somit haben wir nun zwei Versionen des Artikels in der Datenbank. Der Befehl ‘timeline’ gibt eine Übersicht der Änderungshistorie mit den Kommentaren:

```
# fossil timeline
=== 2012-02-22 ===
02:48:05 [df0007f5e7] *CURRENT* Zweiter Commit
                      (user: cas tags: trunk)
02:41:46 [caf95931d0] Erste Version des fossil-Artikels
                      (user: cas tags: trunk)
02:30:22 [efc57e6bf2] initial empty check-in
                      (user: cas tags: trunk)
```

Der Befehl ‘diff’ gibt die Änderungen zwischen den Versionen im Repository aus. Mit ‘fossil help diff’ bekommt man die Hilfe zum ‘diff’–Befehl angezeigt. Mit dem Befehl ‘revert’ kann zu einem früheren Zeitpunkt in der Quellcode–Historie zurückgekehrt werden.

Es ist gefahrlos möglich, ein Repository geöffnet zu halten, auch wenn der Rechner neu gestartet wird. Nur wenn das Repository verschoben oder auf einen anderen Rechner kopiert werden soll, sollte das Repository vorher mit ‘fossil close’ geschlossen werden. Vor einem ‘close’ erst noch alle Änderungen an den Quelldateien per ‘commit’ in die Datenbank übernehmen. Der Befehl

'help' gibt eine Liste aller Fossil–Befehle aus, und 'fossil help <Befehl>' gibt eine Beschreibung des Befehls aus.

Fossil Web-Oberfläche

Neben der Kommandozeilenschnittstelle bietet Fossil auch eine Web–Oberfläche auf Port 8080 der Loopback–Adresse. Dieser Webserver wird mit 'fossil server' gestartet. Im Web–Browser unter der URL http://localhost:8080/ befindet sich dann die Konfigurationsoberfläche für Fossil. Neben dem Projektnamen können über diese Oberfläche die Sicherheitseinstellungen für das Repository vorgenommen werden (wenn Fossil als Webserver über das Internet betrieben wird), und die Web–Oberfläche enthält auch ein Bug–Tracking–System um Fehler–Reports zu verwalten, sowie ein einfaches Wiki–System, welches zur Dokumentation des Projektes verwendet werden kann.

Fossil zeigt, dass auch mächtige Quellcode–Verwaltungssysteme nicht kompliziert zu installieren und bedienen sein müssen. Ein Fossil–Repository ist in wenigen Minuten erzeugt, Zeit, die später hundertfach durch die Funktionen des Systems wieder eingespart werden kann. Mit Fossil gibt es keine Ausreden mehr, Software–Versionsverwaltung sollte für jedes Projekt eingesetzt werden.

Links

[1] `www.fossil-scm.org` Fossil Homepage

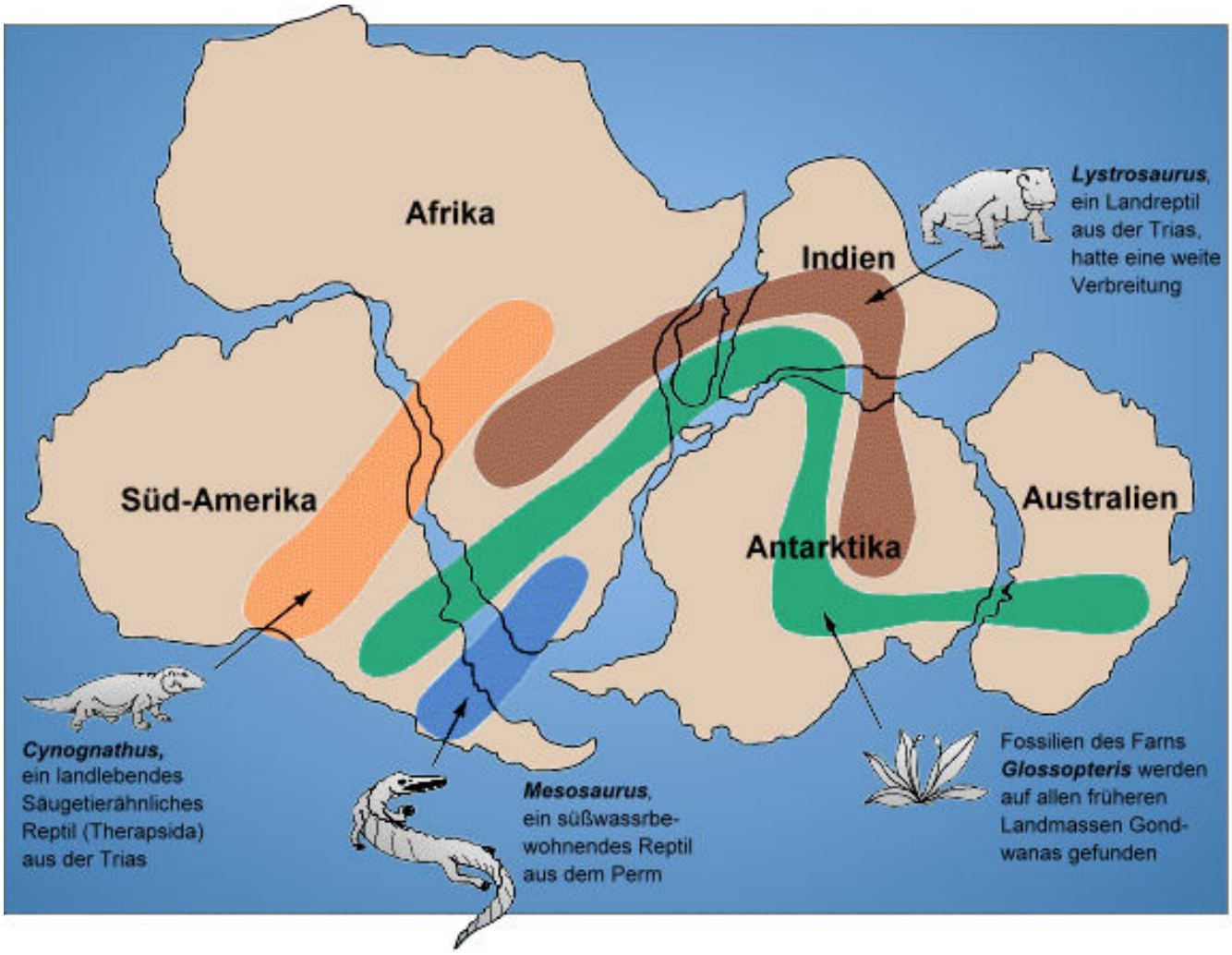


Abbildung 3: Lage der Fossilien, daher Rückschluss auf Gondwana
Quelle: `http://commons.wikimedia.org/wiki/File:Gondwana_fossil_map_ger.png`

Nachtrag: Morse4 repariert

Erich Wälde

Die in Heft 4/2011 der Vierten Dimension erwähnte vierte Variante des Morseprogramms war leider fehlerhaft. Hier eine korrigierte Version.

In Heft 4/2011 der VD [1] wurden drei Versionen eines Morseprogramms vorgestellt, die Carsten und ich in den Forth*h*bildungen verwendet hatten. Eine vierte Version, die die Zeichentabelle dauerhaft im Flashspeicher ablegt, hatte ich zwar erwähnt, war aber nur zum Herunterladen verfügbar.

In der Zwischenzeit fand ich heraus, dass die Funktion `morseemit` zwei Fehler beinhaltet. Zum einen war die Behandlung des Leerzeichens fehlerhaft, und zum anderen funktionierte das folgende Programmschnippselchen nicht richtig:

```
> : msg ." abs" cr ;
 ok
> morse msg endmorse
abs
 ok
```

Zwar wurden alle drei Zeichen auf der seriellen Konsole ausgegeben, aber *gemorst* wurde nur das finale `s`. Sehr merkwürdig.

Fehlersuche

Diverse Variationen von `msg` ließen kein Muster erkennen. Also bohrte ich die Funktion `morseemit` mit zusätzlichen Ausgaben auf:

```
: morseemit ( key -- )
  dup o-emit @ execute
  [char] . o-emit @ execute \ debug
  dup bl = if \ Leerzeichen == Wortende
    Wend
    drop
  else

    \ Grosse [A-Z] -> kleine Zeichen [a-z]
    dup $40 > over $5b < and if
      $20 +
    then

    \ debug
    [’] emit defer@ >r
    o-emit @ is emit
    dup .
    r> is emit

    \ Argument auf [a-z] begrenzen
    dup  $60 >
    over $7b < and if
    [char] _ o-emit @ execute \ debug
    \ index - TabellenOffset
      $60 -
      \ gepackten Wert holen
      mtable + @i
      \ entpacken und ausgeben
      unpack domorse
    else
      \ ungueltigen Code verwerfen
      drop
    then
  then
;
```

Das erzeugte die folgende Ausgabe:

```
> init hex morse s" abs" type endmorse
a.61 _b.62  _s.73 _ ok
> morse msg endmorse
a.6261  b.6162  s.73 _ ok
```

Das sah verdächtig danach aus, dass `itype` zwei Bytes als cell auf den Stapel legt, und damit zu große Werte an `morseemit` verfüttert. Nachdem das Problem soweit seziert war, war Abhilfe denkbar einfach.

```
: morseemit ( key -- )
  $00ff and \ fix itype
  \ altes emit ausfuehren
  dup o-emit @ execute
  ...
;
```

Mit einer einzigen, zusätzlichen Zeile verschwand das Phänomen. Damit war mir geholfen. Und damit verstand ich auch im Nachhinein, warum in der anderen Version von `morseemit` ein `$00ff and` enthalten ist.

Gegenrede

Man fragt sich aber, ob das nicht doch ein Fehler in `itype` ist, die Bytes nicht schön einzeln auf den Stapel zu legen. Matthias Trute hat schnell und unbürokratisch die Funktion `itype` korrigiert. Die Korrektur ist im `trunk` und später ab der Version 4.8 von amforth enthalten.

Mein Dank geht an Matthias Trute und die Teilnehmer des mittwöchlichen IRC–chats.

Referenzen

1. Forth*h*bildung — Wälde et al., Vierte Dimension 4/2011, Jahrgang 27, S. 9f

Listings

```
\ --- morse/base.fs -------------------------
\ 2011-10-26  EW
\ 2011-11-02  CS
\ arduino duemilanove + danger shield
\ morse code stuff
\ (Potsdam/Augsburg/Oberhausen)

\ make marker loads:
\   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/morse4.fs -----------------------
\ 2011-10-26  EW
\ 2011-11-04  CS
\ arduino duemilanove + danger shield
\ morse code stuff
\ (Potsdam/Augsburg/Oberhausen)
\ 4th version
\ 2012-02-23  EW  fixed itype

marker --morse--


\ die Werte in der Flashtabelle sind immer 2 Byte groß
\ die Bits brauchen nicht mehr in 8 bits gepackt werden,
\ sondern lediglich in das niedrige|hoehere Byte
: pack   ( bits #bits -- x )   >< or  ;
: unpack ( x -- bits #bits )
   dup $00ff and swap >< $00ff and ;

\ erstelle Tabelle im flash memory.
\ Werte sind 2 Bytes groß,
\ Index increments in 1 !

create mtable
0 ,                \    $60 offset
%00010  &2 pack ,  \ a  $61
%00001  &4 pack ,  \ b
%01010  &4 pack ,  \ c
%00001  &3 pack ,  \ d
%00000  &1 pack ,  \ e
%00100  &4 pack ,  \ f
%00011  &3 pack ,  \ g
%00000  &4 pack ,  \ h
%00000  &2 pack ,  \ i
%01110  &4 pack ,  \ j
%00101  &3 pack ,  \ k
%00010  &4 pack ,  \ l
%00011  &2 pack ,  \ m
%00001  &2 pack ,  \ n
%00111  &3 pack ,  \ o
%00110  &4 pack ,  \ p
%01101  &4 pack ,  \ q
%00010  &3 pack ,  \ r
%00000  &3 pack ,  \ s
%00001  &1 pack ,  \ t
%00001  &3 pack ,  \ u
%00001  &4 pack ,  \ v
%00011  &3 pack ,  \ w
%01001  &4 pack ,  \ x
%01011  &4 pack ,  \ y
%00011  &4 pack ,  \ z  $7a
0 ,
0 ,
0 ,
0 ,
0 ,
0 ,

```

```
variable o-emit

: domorse ( code #zeichen -- )
  \ Schleife Anzahl der Signale
  0 ?do
    \ Kopie anlegen
    dup
    \ erstes Bit maskieren
    1 and
    \ ist Bit gesetzt?
    if
      \ Strich
      lang
    else
      \ Punkt
      kurz
    then
    \ verbleibende Bits nach rechts schieben
    2/
  loop
  drop \ Argument loeschen
  Zend \ Pause Zeichenende
;

: morseemit ( key -- )
  $00ff and \ fix itype
  \ altes emit ausfuehren
  dup o-emit @ execute
  \ [char] . o-emit @ execute \ debug
  \ Argument == Leerzeichen?
  dup bl = if
    Wend
    drop
  else

    \ Grosse [A-Z] -> kleine Zeichen [a-z]
    dup $40 > over $5b < and if
      $20 +
    then

    \ debug
    \ [’] emit defer@ >r
    \ o-emit @ is emit
    \ dup .
    \ r> is emit

    \ Argument auf [a-z] begrenzen
    dup  $60 >
    over $7b < and if
    \ [char] _ o-emit @ execute \ debug
    \ index - TabellenOffset
      $60 -
      \ gepackten Wert holen
      mtable + @i
      \ entpacken
      unpack
      \ Signale ausgeben
      domorse
    else
      \ ungueltigen Code verwerfen
      drop
    then
  then
;

: morse
  [’] emit defer@ o-emit !
  [’] morseemit is emit
;
: endmorse
  o-emit @ is emit
;

: init
  init \ init aus base.fs
  [’] emit defer@ o-emit !
;
```

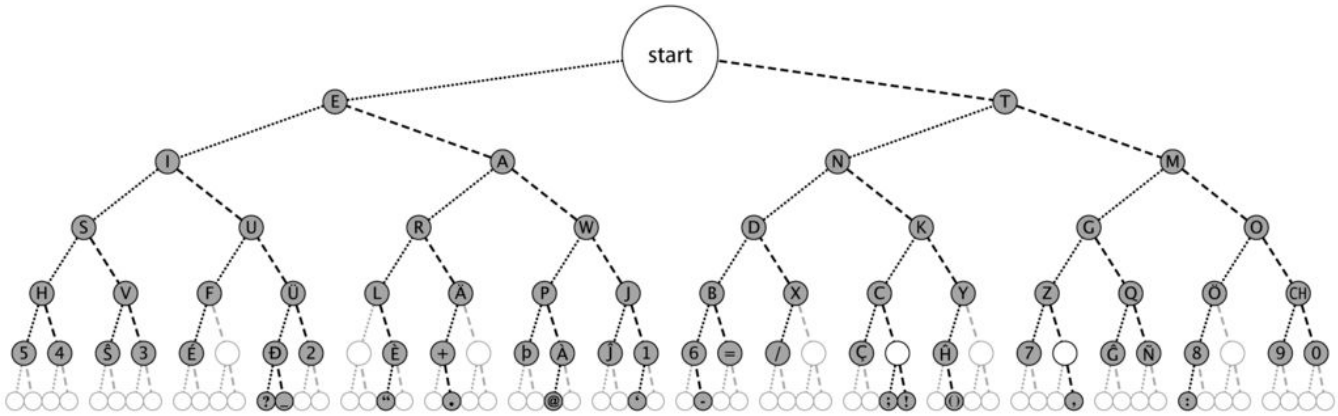


Abbildung 1: Morse Code Baum

CamelForth im FRAM des TI–MSP430FR5739

M. Kalus

Neulich teilte mir Dirk Brühl, unser Forthfreund in den USA, mit, dass es von Texas Instruments (TI) eine MCU gäbe, die FRAM statt flash als permanenten Speicher hat. Und dass dieses FRAM genau wie SRAM gelesen und geschrieben werden kann. Und dass diese MSP430–Prozessor–Familie einen zusammenhängenden Adressbereich hat. RAM, aber nicht-flüchtig? Klingt nach einer idealen Forth–MCU. Umfragen ergaben, dass noch kein Forth dafür vorhanden war. Aber es gab ein kleines kompaktes Forth für die MSP430–Familie, das CamelForth/430 aus 2009 von Bradford J. Rodriguez für das Tini430–board mit dem MSP430F1611–Chip, es legt den Forthkern im *flash* an, compilierte neue Forth–Worte jedoch ins RAM. Das müsste doch einfach auf das *FRAM* zu portieren sein, dachte ich. Zumal TI ein feines preiswertes FRAM–Experimentierboard beigesteuert hat, das MSP-EXP430FR5739, um alles auszuprobieren.

Dieses Platinchen steckt man einfach an ein USB–Kabel zum PC und alles klappt - die Stromversorgung, die serielle Schnittstelle zum Terminal und die Programmierschnittstelle. Alle drei wurden über den einen USB–Anschluss realisiert, eine mustergültige Lösung, die TI hier vorführt. Die Forth–Gesellschaft hat inzwischen 3 Exemplare des MSP-EXP430FR5739 in den Verleih genommen. Wer damit also spielen möchte, kann dies praktisch kostenlos tun. Viel Vergnügen damit.

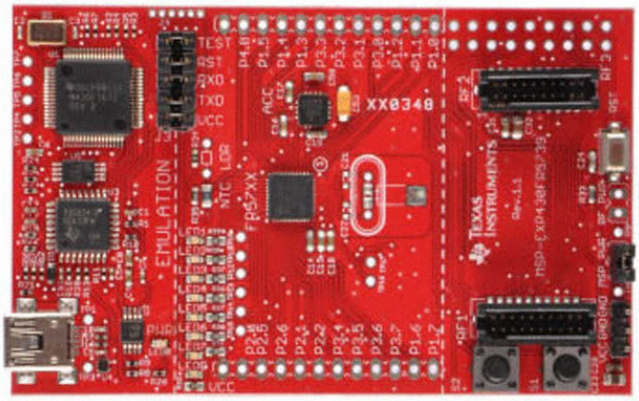


Bild 1: MSP-EXP430FR5739

CamelForth/430

Dieses kleine Forth kommt aufgeräumt in 5 Dateien daher. Die CPU und Forth–Modell–abhängigen Worte sind in der Datei *deps430FR5739.s43* versammelt, und in *core430FR5739.s43* stehen die Low–Level–Routinen. Die übrigen ANS–kompatiblen Forth–Worte findet man in *hilvl430FR5739.s43* und die Initialisierung der MCU und des Forth selbst ist im File *init430FR5739.s43* niedergelegt, also das, was nach reset als erstes durchlaufen wird. Die Interruptvektoren sind übersichtlich in *vecs430FR5739.s43* versammelt, auch der *reset*–Vektor wird dort gesetzt.

Rodriguez benutzte die IAR Systems MSP430 Workbench, „Kickstart“ genannt, für das CamelForth/430. Sein Projekt wurde wie es war probeweise im IAR compiliert, was auf Anhieb fehlerfrei ging. Sodann wurde das Demoprogramm von TI als Projekt in die IAR geladen, um die Verbindung zum MSP-EXP430FR5739 zu erproben. Auch das funktionierte sofort einwandfrei über den USB. Das Demopropgram erwies sich dann als eine wichtige Quelle für die forthige Initialisierung der MCU.

Speicher im MSP430FR5739 im Vergleich zum MSP430F1611

Wir bekommen derzeit 16KB FRAM im MSP430FR5739. CamelForth brauchte davon 6KB, passt also erstmal dort hinein.

```
; MEMORY MAP of the MSP430FR5739
; 0000-0FFF = peripherals (4 KB)
; 1000-17FF = bootstrap loader BSL0..3 (ROM 4x512 B)
; 1800-187F = info B (FRAM 128 B)
; 1880-18FF = info A (FRAM 128 B)
; 1900-19FF = N/A (mirrored into info space)
; 1A00-1A7F = device descriptor info (FRAM 128 B)
; 1A80-1BFF = unused (385 B)
; 1C00-1FFF = RAM (SRAM 1 KB)
; 2000-C1FF = unused (41472 B)
; C200-FF7F = code memory (FRAM 15743 B)
; FF80-FFFF = interrupt vectors (FRAM 127 B)
```

Bild 2: FRAM

Der MSP430F1611 bietet deutlich mehr Speicher, sowohl Ram als auch Flash. Und ist damit sicherlich eine für Forth interessante MCU. Das dürfte auch der Grund gewesen sein für Rodriguez, sein CF auf dies MCU zu bringen. Doch finde ich das FRAM so interessant, dass CF einfach darauf ausprobiert werden musste.

```
; MEMORY MAP of the MSP430F1611
; 0000-01FF = peripherals
; 0200-09FF = 2KB mirrored RAM
; 0A00-0BFF = unused
; 0C00-0FFF = 1KB boot memory
; 1000-10FF = 256B information memory
; 1100-18FF = 2KB mirrored RAM
; 1900-38FF = 8KB RAM
; 3900-3FFF = unused
; 4000-FFFF = 48KB flash ROM
; FFE0-FFFF = interrupt vectors
```

Was zu tun war, um die ersten Lebenszeichen des CF430FR zu erhalten.

Zunächst wurde mit den Dateien des CamelForth/430 das CF430FR als neues Projekt im IAR angelegt. Darin wurden dann alle Forthworte, die für den flash-Zugriff gemacht waren, auskommentiert - netterweise waren solche Worte mit einem *i* für *instruction memory* gekennzeichnet. Daraus folgte ein Haufen Fehlermeldungen beim *make* wegen der nun unbekannten Namen. Etliche davon waren erfreulicherweise schon von Rodriguez als *alias* angelegt worden, und konnten daher einfach umgewidmet werden, also `IHERE` zu `HERE` und `I@` zu `@` usw. Auch dabei zeigte sich, dass CF gut strukturiert ist. Weitere i-Worte mussten sodann im Quellcode aufgesucht und durch ihre gewöhnliche Funktion ersetzt werden. So wurde nach und nach eine fehlerfreie Übersetzung fertig und ließ sich in die Experimentierplatine laden, nachdem auch die Speicherzuweisungen im Code und für den Linker an die FRAM Speichersegmente angepasst worden waren.

Doch es gab daraufhin leider noch kein ok vom frisch erzeugten Forth. Denn die serielle Schnittstelle war noch nicht passend konfiguriert. Dafür musste noch die Datei *init430FR5739.s43* an die neue CPU angepasst werden. Die gut gemachten TI–Handbücher zum Chip und zum Experimentierboard halfen dabei weiter[1]. Insbesondere die korrekte Initialisierung der seriellen Schnittstelle zum Terminal für `KEY` und `EMIT` wurde neu geschrieben, sowie die Initialisierung der Ports, die Registerverteilung usw. Schließlich wurde noch eine kleine Testschleife ans Ende des Forthkerns hinzugeladen, die von dem IAR–Debugger durchlaufen werden konnte, und siehe da, das I/O funktionierte, und über die blauen LEDs der Platine und dann auch im Terminal wurde angezeigt, dass die Verbindung hergestellt war.

Die `KEY EMIT` Testschleife

Um nach dem Reset mit der Testschleife zu starten, wurde ihr vorübergehend das label INITIP zugeordnet und die Forthregister im init430FR5739.s43 dazu passend gesetzt:

```
; Forth registers
    MOV     #RSTACK,SP  ; set up stack
    MOV     #PSTACK,PSP
    MOV     #UAREA,&UP  ; initial user pointer
    MOV     #INITIP,IP ; testschleife
    MOV     #0,TOS
    NEXT
```

So zeigt der IP also auf die Testschleife, und startete den Test nach dem *reset*. Später konnte dann statt `#INITIP` das `#COLD` eingesetzt werden, um mit dem Forthsystem selbst zu starten.

In der endlosen Testschleife wurde einfach nur `KEY` abgefragt und per `!LEDS` an die blauen LEDs auf der Platine ausgegeben[2], und auch per `EMIT` an das Terminal. Die Schwierigkeit lag darin, das Assemblerstückchen `STORELEDS` mit dem Debugger des IAR zu testen. In der endlosen Schleife des inneren Interpreters des Forth dauerte es zu lange. Ich hab dann einfach das Codestück dem *init* angehängt, so dass es beim *reset* ohne das Forthsystem zu durchlaufen ausgeführt wurde. So ließ es sich einfach debuggen, zumal der IAR–Debugger den I/O mit anzeigt. Danach konnte `!LEDS` als Forthwort gefasst und in die High–Level–Testschleife einbezogen werden.

```
; TESTING 8x blue LEDs in a row.
;   (portpinX->---resistor---LED---GND)
; LED1 - PJ.0    LED5 - P3.4
; LED2 - PJ.1    LED6 - P3.5
; LED3 - PJ.2    LED7 - P3.6
; LED4 - PJ.3    LED8 - P3.7

; MSP-EXP430FR5739 FRAM Experimentation board blue LEDS output.
    ; ( c  -- )
    HEADER  STORELEDS,5,'!LEDS',DOCODE
    BIC     #0x000F,&PJOUT ; LED1..4 off
    BIC.B   #0x00F0,&P3OUT ; LED5..8 off
    MOV     TOS,X          ; X = scratch register R10
    AND     #0x000F,X      ; don't change any but LEDs
                           ; lower nible
    BIS     X,&PJOUT       ; set the LED bits
    MOV     TOS,X          ; X = scratch register R10
    AND     #0x00F0,X      ; don't change any but LEDs
                           ; higher nible
    BIS.B   X,&P3OUT       ; set the LED bits
    MOV @PSP+,TOS          ; do_drop
    NEXT

; TESTING FORTH EXECUTION
    PUBLIC INITIP
INITIP:
    DW  KEY,DUP,STORELEDS,EMIT
    DW  lit,0,qbran
    DW  INITIP-$
    DW  bran,-2

; INITIP   equ COLD+2
lastword equ link

        END
```

Vom ersten Test zum ok und dann weiter

Doch obwohl nun die blauen LEDs den `KEY` zeigten und auch das `EMIT` richtig rüber kam ins Terminal, das ersehnte OK fehlte noch. Klammerte man den Test aus und setzte `INITIP equ COLD+2` in Kraft, gab es leider nichts zu sehen auf dem Terminal. Der Verdacht fiel auf `TYPE`, in dem was noch nicht stimmen konnte. So wurde zunächst mal ein `.ID` gemacht, an bekannter Adresse im FRAM, das dann von `TYPE`, welches auch in die Testschleife kam, ausgegeben werden konnte. Und als das ging, wurde `.ID` in das `COLD` eingehängt, um zu sehen ob die Ausgabe auch von dort kam. Und auch das ging

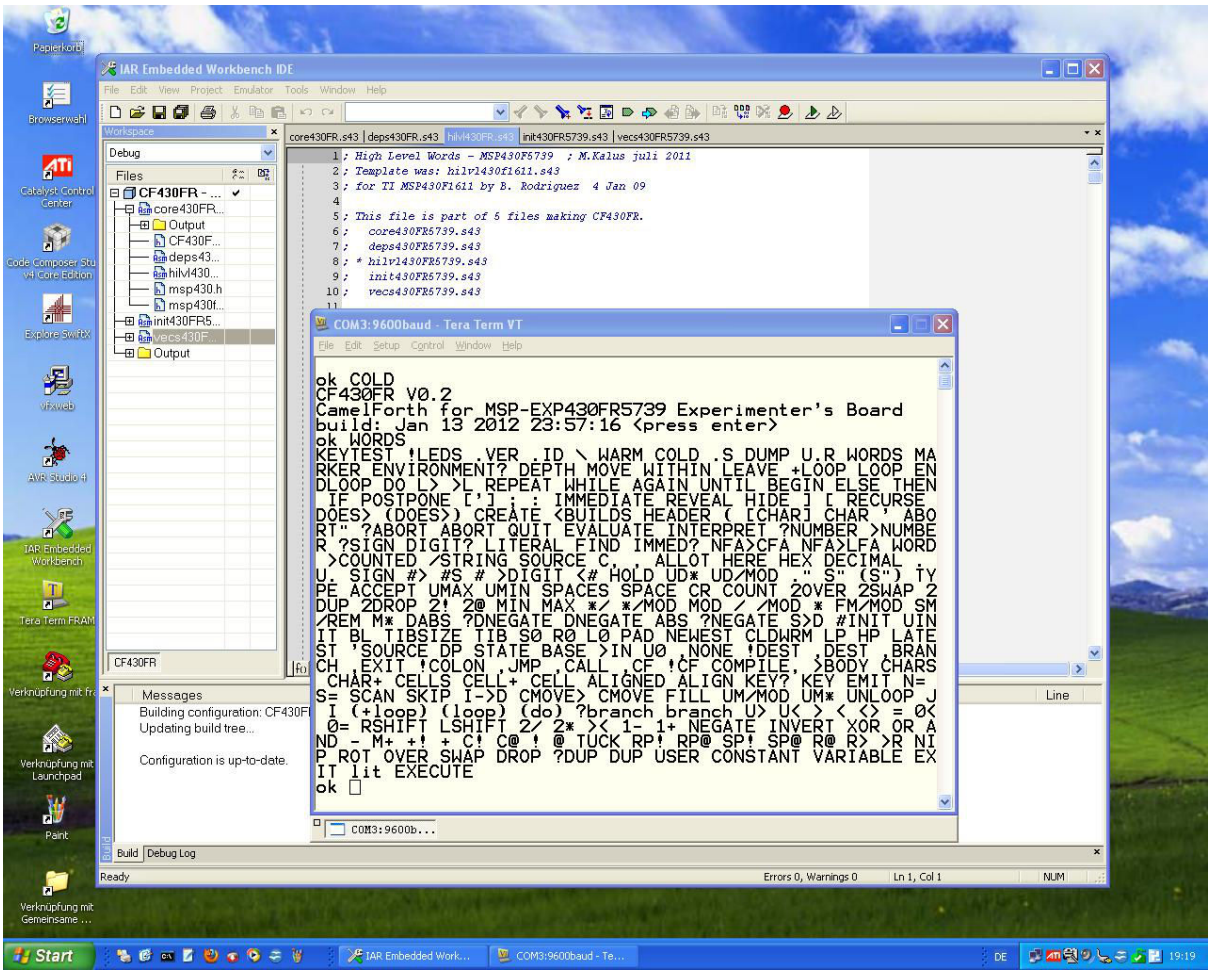


Bild 3: IAR und CF430FR im Terminal

nun, TYPE war glücklich ans FRAM angepasst worden, das OK war da.

```
;ID        0          1         2
;            12345678901234567890
FRid: DB 15,'CamelFort V0.2 '
      EVEN

; .ID      --     type id
        HEADER  DOTID,3,'.ID',DOCOLON
        DW CR,lit,FRid,COUNT,TYPE
        DW EXIT
```

Mit dieser Erfahrung war es dann möglich, auch das Wort ." und seine Faktoren an das FRAM anzupassen, so dass auch der Klassiker

```
: test ."hallo world";
```

es nun im CF430FR tat. Welche Freude nächtens, wenn das dann geht! Die weiteren Versuche ergaben dann noch Fehler in VARIABLE, was am CREATE lag. CREATE war noch für die Harvard–Speicheraufteilung des MSP430F1611 ausgelegt gewesen, und verhielt sich wie CONSTANT. Das DOCON war der gemeinsame Code, den auch DOVAR sowie docreate nutzten. Aber es brauchte nur ein copy&paste und darin dann genau ein einziges Zeichen weniger, um die Anpassung ans FRAM zu machen. Aus dem

```
DOCON:  ; -- x          ; CONSTANT fetches cell
                        ; from PFA to TOS
        SUB #2,PSP      ; make room on stack
        MOV TOS,0(PSP)
        MOV @W,TOS      ; fetch from parameter
                        ; field to TOS
        NEXT
```

wurde so zusätzlich erzeugt

```
docreate: ; -- a-addr   ; CREATE fetches its address
DOVAR:    ; -- a-addr   ; VARIABLE fetches its address
        SUB #2,PSP      ; make room on stack
        MOV TOS,0(PSP)
        MOV  W,TOS      ; move parameter field
                        ; address to TOS
        NEXT
```

Da im W–Register sowieso immer die nächste pfa liegt, also unsere gesuchte Adresse, braucht sie nur auf den Datenstack geschoben zu werden.

Dann gab es noch den Umstand, dass der Linker des IAR die init–Datei hinter die core–Datei ins FRAM packte, also auf höhere Adressen, und damit der dictionary pointer DP mitten in die Initialisierung zeigte. Was solange gut ging, bis etwas neu compiliert wurde und dann ein reset oder COLD kam. Aber es war nicht weiter schwer, das label ROMDICT an das Ende von init zu verlagern, und dann ging auch das. Auch war so (erneut) bewiesen,

dass man mit Forth im RAM alles machen kann, auch den eigenen Prozess abschießen. So, damit hat auch der MSP430FR5739 ein Forth und kann damit unter die Lupe genommen werden.

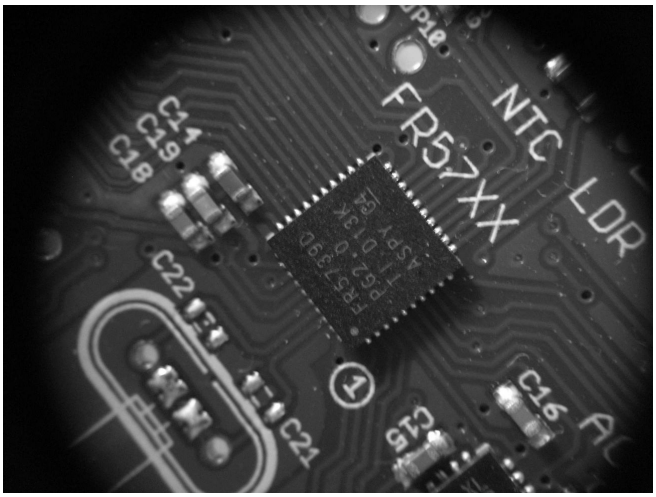


Bild 4: MSP430FR5739

Bleibt noch zu erwähnen, dass CamelForth case–sensitiv angelegt ist, und man lauter Fragezeichen bekommt, wenn man das nicht gewöhnt ist. Es muss also `.S` heißen, und `WORDS`, also tatsächlich in Großbuchstaben, um ordentliche Ausgaben zu bekommen.

Bring es ans Laufen, dann mach es hübsch.

Nun da es soweit funktioniert, kann man sich überlegen, was alles noch hübscher werden sollte am CF430FR, bevor man es an Rodriguez übergibt, was ich vorhabe. So müssten Kommentare verschwinden, die noch für die *Flash*–Version waren und nun ausgedient haben, andere müssten hinzu. Auch die Versionsgeschichte gehört überarbeitet. Das Verhalten des CF bei einem *forth crash test* oder *reset* war noch nicht zufrieden stellend. Es gab im CF nur `COLD` als Eintrittspunkt, ausgeführt nach Stromausfall oder Reset. Es fehlt noch ein ordentlicher Warmstart. Vom Forth aus hinzu kompilierte Worte sollen ja beim Neustart noch da sein. Auch müsste ein Mechanismus her, mit dem der Anwender wahlweise einen Neustart des Forthkerns durch `COLD` oder `WARM` von außen herbeiführen kann. Ferner müsste der *anstest0.8* mal darüberlaufen. Daher steht die Portierung nun auch im Repository der Forth–Gesellschaft [3], und wer möchte, kann mitmachen, das CF430FR weiter zu debuggen. Die Portierung erfolgte übrigens mit Windows XP auf einem Athlon64–Prozessor–3000+ und der IAR–Kickstart–Version 5.

Fußnoten

[1] `http://www.camelforth.com`
[2] Die Idee dazu fand ich im core file des Muster CamelForth vor. !LED wurde lediglich an das vorliegende Board angepasst.
[3] `http://www.forth-ev.de/repos/CF430FR`

Links

`http://www.ti.com/-->suchen:MSP-EXP430FR5739`
Es wird beworben als: "The Experimenter Board helps designers quickly learn and develop using the new MSP430FR57xx MCUs, which provide the industry's lowest overall power consumption, fast data read /write and unbeatable memory endurance."
`http://de.wikipedia.org/wiki/Ferroelectric_Random_Access_Memory`

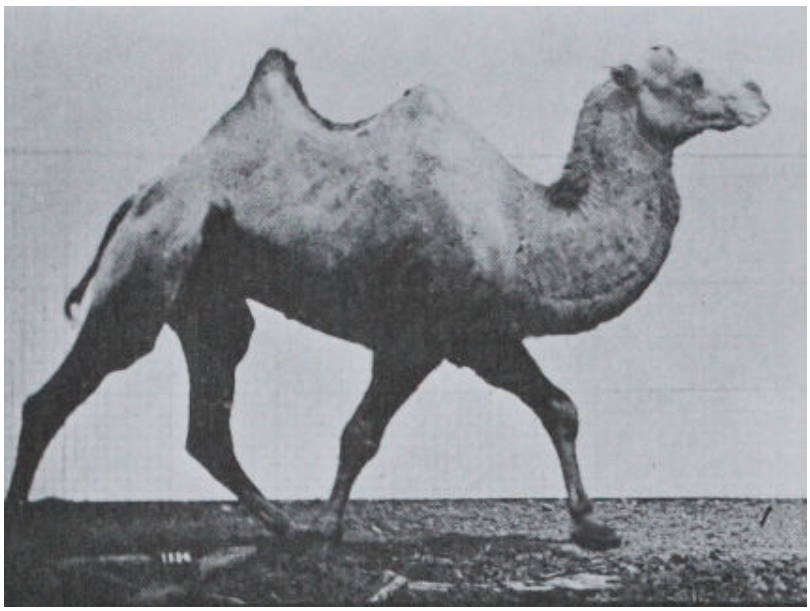


Bild 5: Kamel im Passgang

4e4th auf dem TI–LaunchPad im MSP430G2553

M.Kalus

4E4th ist ein kleines Forth, das auf dem Texas–Instruments–LaunchPad abläuft. Es steckt in der MCU MSP430G2553. Du kommunizierst mit dem 4E4th mit Hilfe eines Zeilen–Editors. der über einen Terminal–Emulator betrieben wird. 4E4th basiert auf der MSP430–CamelForth–Version 0.3 von Brad Rodriguez, welches er für das TI–Tini430–Board mit dem MSP430F1611 geschrieben hatte. Es ist ein ANS–Forth, und belegt knapp 8K im FLASH der MSP430G2553–MCU (0xE000-0xFFFF). Weitere 8K (0xC000-0xDFFF) sind frei für eigene Experimente. 4E4th ist, wie das CamelForth auch, freie Software (GNU General Public License).

Das LaunchPad von TI

Derzeit stark beworben wird das LaunchPad genannte Platinchen MSP-EXP430G2 von TI. Im Gegensatz zu seinem etwas größeren Bruder, dem MSP-EXP430FR mit dem FRAM, kostet das LaunchPad grad mal 4,30 US$, ist also quasi geschenkt. Die Platine hat einen klassischen 20DIL–Sockel für die MCUs. Der MSP430G2553 steckt schon. Das LaunchPad hat auch den *Emulation* genannten Teil an Bord, in dem die beiden Funktionen USB-serieller Wandler und BSL (boot strap loader) realisiert sind. Über den BSL wird die MCU mit einem Programm beladen, in unserem Falle mit dem 4e4th. Dieses kleine Forth funktioniert dann klassisch über den UART der MCU und die serielle Schnittstelle via USB am Terminal. Versorgt wird die Platine auch gleich über den USB. Sehr praktisch das Ganze — einfach USB anstöpseln, fertig.

Making of 4e4th

Die Schritte, um vom CamelForth zum 4e4th zu kommen, waren ähnlich wie beim Bruderchip mit dem FRAM. Aber weil die MSP430G553–MCU Flash als Hauptspeicher hat, waren andere Maßnahmen nötig, das Forth darin vernünftig handhabbar zu machen. Denn das Flash lässt sich nur dann zellenweise beschreiben, wenn es $FFFF ist, also erased, Bits können nur von H auf L gesetzt werden. Umkehren, also zurücksetzen auf $FFFF, kann man es hingegen nur sektorenweise. Die Sektorgröße ist 256 Byte. So lässt sich das Forth–Dictionary wohl im Flash aufbauen, aber ein `FORGET` ist nicht möglich, und auch `MARKER` geht nur auf Flashsektorgrenzen. Darum gibt es ein `WIPE`, um das komplette User–Flash zurückzusetzen. Der Forth–Kern residiert schreibgeschützt im oberen Teil des Flash.

So musste auch sichergestellt werden, dass die User-Area im RAM und der Stand des Forth im User–Flash übereinstimmen. Und, da man nicht in den Forth–Kern im oberen Flash schreiben kann, muss die User-Area woanders hin gerettet werden können. Dafür gibt es das INFO-Flash in der MCU. Diese 32 Zellen kleinen Bereiche infoA bis infoD lassen sich einzeln löschen und wieder beschreiben und sind daher ideal dafür. Ein `SAVE` sichert die User–Area dorthin. Beim Booten kann dann anhand des `CRC`–Wertes aus dem User–Flash und der User–Area festgestellt werden, ob Übereinstimmung besteht. So kommt man nach einem Reset immer in den zuletzt gesicherten Zustand zurück. Über die User–Variable `APP` kann dabei ein Autostart erreicht werden.

Das 4e4th selbst ist grad mal knapp über 7K klein, so dass 8K für eigene Experimente bleiben. Darin enthalten sind neben dem ANS core word set schon einige Worte für die Peripherie des LaunchPad. So lassen sich mit `SET` und `CLR` und `TOGGLE` einzelne Bits wortweise stellen, und mit `CSET CLCR` und `CTOGGLE` auch byteweise. Die beiden Ports `P1` und `P2` sind damit sehr einfach zu bedienen und damit auch die beiden LEDs auf dem Board. Es gibt die Abfrage des Tasters `S2?` und auch ein `AT-XY` und `PAGE` für VT-100–Terminals, so dass mit etlichen Experimenten sogleich losgelegt werden kann. Ein Programm, um zu morsen, ist damit ein Klacks.

Das 4e4th wird im forth-ev-wiki unterstützt. Wer LaunchPads haben will, wende sich an den Autor, es sind noch einige auf Lager. Aber immer mehr Händler haben es inzwischen auch im Sortiment.

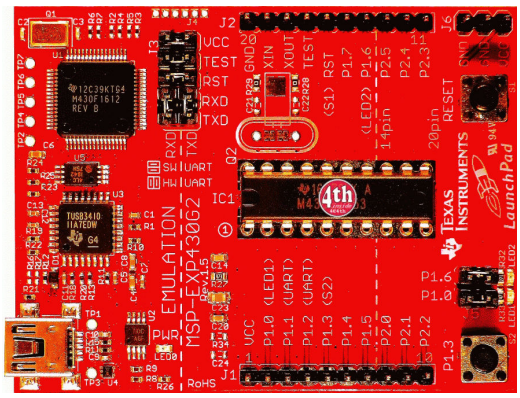


Bild 1: Das TI–LaunchPad mit Forth Inside

Links

http://www.forth-ev.de/wiki/doku.php/projects:4e4th:start
http://www.4e4th.eu/
http://www.camelforth.com/

4e4th auf dem Launchpad

```
4E4th v0.34 Apr 15 2012|110001110 Cold
 ok
cr cr words cr cr

S2? S2 GREEN RED P3 P2 P1 UNUSED MEM MEMTOP MEMBOT CGET CTOGGLE CCLR CLR
CSET SET MS 1MS BIN PAGE AT-XY BELL .VER \ 2CONSTANT WIPE FACTORY COLD WARM
BOOT SAVE VALID? APPCRC ITHERE crc (crc ccrc .S DUMP U.R WORDS MARKER
FLALIGNED NOOP ENVIRONMENT? DEPTH MOVE WITHIN LEAVE +LOOP LOOP ENDLOOP DO L>
>L REPEAT WHILE AGAIN UNTIL BEGIN ELSE THEN IF POSTPONE [’] ; : IMMEDIATE
REVEAL HIDE ] [ RECURSE DOES> (DOES>) CREATE <BUILDS HEADER ( [CHAR] CHAR ’
ABORT" ?ABORT ABORT QUIT EVALUATE INTERPRET ?NUMBER >NUMBER ?SIGN DIGIT?
LITERAL FIND IMMED? NFA>CFA NFA>LFA WORD >COUNTED /STRING SOURCE IC, I,
IALLOT IHERE C, , ALLOT HERE HEX DECIMAL . U. SIGN #> #S # >DIGIT <# HOLD
UD* UD/MOD IWORD ." IS" (S") (IS") TYPE ACCEPT UMAX UMIN SPACES SPACE CR
COUNT 2OVER 2SWAP 2DUP 2DROP 2! 2@ MIN MAX */ */MOD MOD / /MOD * FM/MOD
SM/REM M* DABS ?DNEGATE DNEGATE ABS ?NEGATE S>D APPU0 INFOB COR #INIT UINIT
BL TIBSIZE TIB S0 R0 L0 PAD CAPS APP NEWEST IDP LP HP LATEST ’SOURCE DP STATE
BASE >IN U0 ,NONE !DEST ,DEST ,BRANCH ,EXIT !COLON ,JMP ,CALL ,CF !CF COMPILE,
>BODY CHARS CHAR+ CELLS CELL+ CELL ALIGNED ALIGN ZERO KEY? KEY EMIT N= S= SCAN
SKIP I->D CMOVE> CMOVE FILL UM/MOD UM* UNLOOP J I (+loop) (loop) (do) ?branch
branch U> U< > < <> = 0< 0= RSHIFT LSHIFT 2/ 2* >< 1- 1+ NEGATE INVERT XOR OR
AND - M+ +! + D->I IC! I! FLERASE C! C@ ! @ TUCK RP! RP@ SP! SP@ R@ R> >R NIP
ROT OVER SWAP DROP ?DUP DUP USER CONSTANT VARIABLE EXIT lit EXECUTE

ok
green .s <2> 64 33 ok
green ctoggle ( grüne LED ändern ) ok
red cclr ( rote LED hell ) ok
green cset ( grüne LED dunkel ) ok
s2? . 0 ( Taster gedrückt? ) ok
ok
: combined ( mask1 addr1 mask2 addr2 -- mask3 addr3 )
   rot over - Abort" not the same address"
   rot rot or swap ; ok
ok
: blink ( -- )
   BEGIN  s2? 0=  WHILE 100 ms  red green combined ctoggle  REPEAT ; ok
ok
blink ( wildes Geblinke bis zum Drücken des Tasters ) ok
```

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, im Sommer (Mai–September) im Chilli Asia Dachauer Str. 151, 80335 München, im Winter in der Pizzeria La Capannina, Weitlstr. 142, 80995 München (Feldmoching–Hasenbergl).

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`

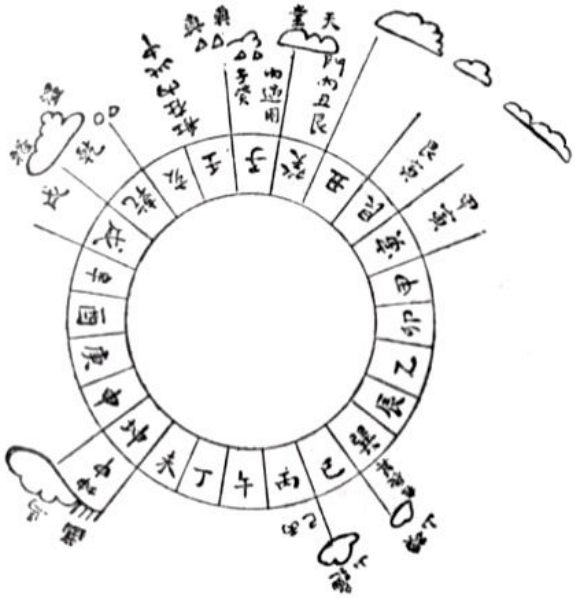
Spezielle Fachgebiete

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`

8.–11. März 2012 Forth-Tagung



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.

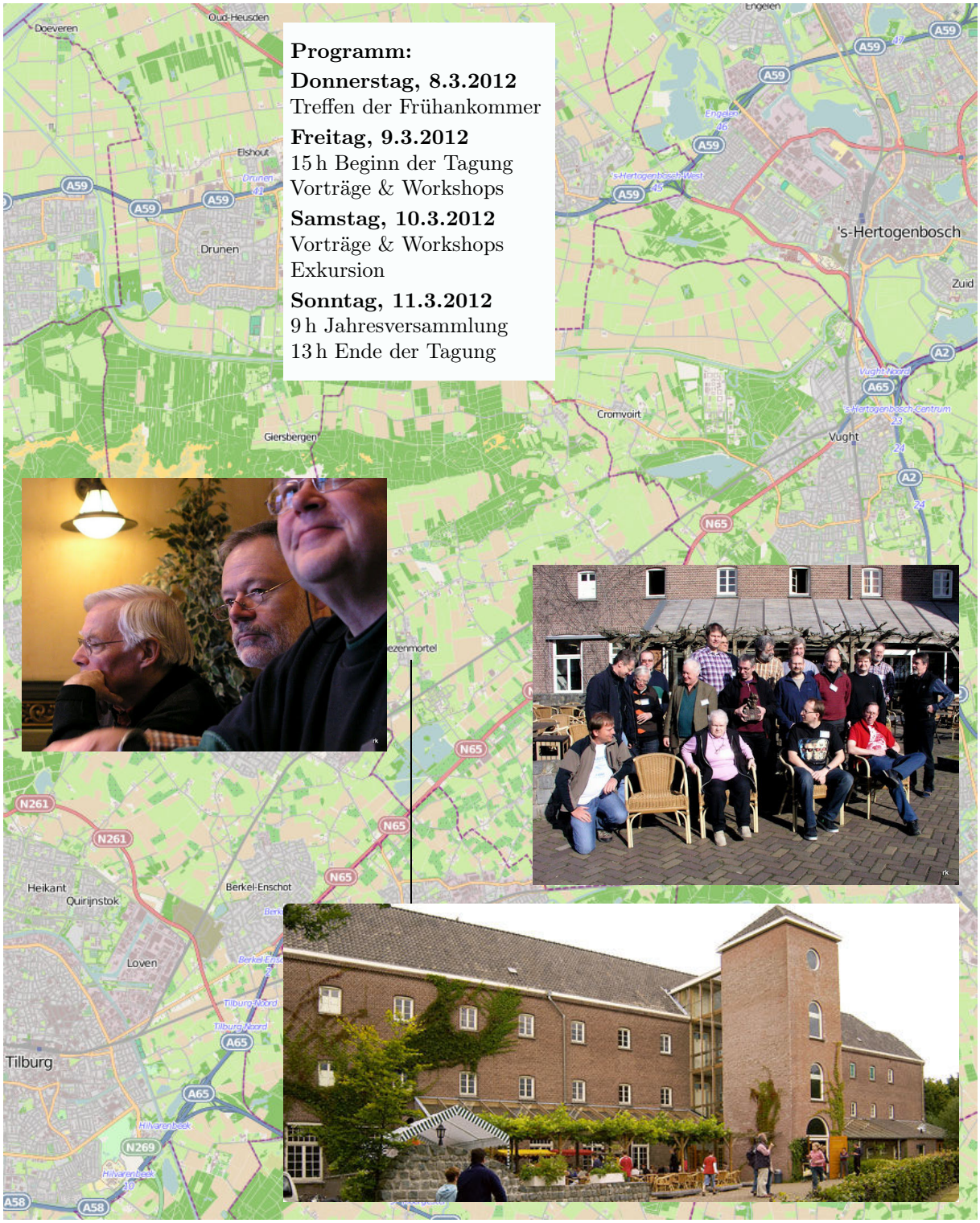
Schön wars bei der

Forth–Tagung 2012 vom 8. bis 11. März 2012

im Beukenhof

Capucijnenstraat 46 · 5074 PJ Biezenmortel · Noord–Brabant · Niederlande

http://www.denieuweklasse.nl/recreatief/accommodaties/beukenhof



Programm:

Donnerstag, 8.3.2012
Treffen der Frühankommer

Freitag, 9.3.2012
15 h Beginn der Tagung
Vorträge & Workshops

Samstag, 10.3.2012
Vorträge & Workshops
Exkursion

Sonntag, 11.3.2012
9 h Jahresversammlung
13 h Ende der Tagung

Anreise siehe: http://www.openstreetmap.org/?lat=51.62313&lon=5.18049&zoom=16&layers=M

Quellen: www.denieuweklasse.nl/recreatief/accommodaties/beukenhof, openstreetmap.org, wikimedia.org