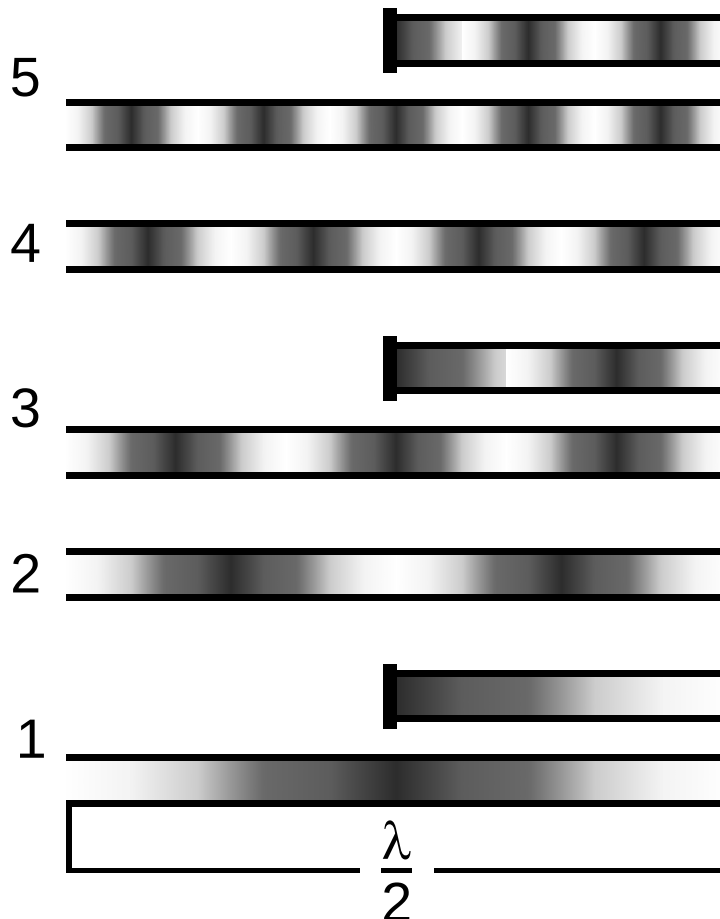




*für Wissenschaft und Technik, für kommerzielle EDV,
für MSR-Technik, für den interessierten Hobbyisten*



In dieser Ausgabe:



Adventures 8: Wortliste schließen

RSC-Forth-Decompiler

Adventures 9: Funklöcher

Bootmanager und FAT-Reparatur

Wave Engine (1)

Xchars im Microcontroller

Forth-Compiler-Hilfsbefehle als
High-Level-Befehle

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.

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

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.

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“!

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

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, KKFORH und viele PDVersionen). FORTH-Hardware (z.B. Super8) und Literaturservice. Professionelle Entwicklung für Steuerungs- und Meßtechnik.

Adventures 8: Wortliste schließen	5
<i>Erich Wälde</i>	
Adventures 9: Funklöcher	7
<i>Erich Wälde und Martin Bitter</i>	
Wave Engine (1)	13
<i>Hannes Teich</i>	
Forth-Compiler-Hilfsbefehle als High-Level-Befehle	20
<i>Willi Stricker</i>	
RSC-Forth-Decompiler	22
<i>Dirk Brühl</i>	
Bootmanager und FAT-Reparatur	28
<i>Fred Behringer</i>	
Xchars im Microcontroller	37
<i>Bernd Paysan</i>	



Abbildung 1: Huangshan, gehört eigentlich nicht hierher



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

Manuskripte und Rechte

Berücksichtigt werden alle eingesandten Manuskripte. Leserbriefe können ohne Rücksprache wiedergegeben werden. Für die mit dem Namen des Verfassers gekennzeichneten Beiträge übernimmt die Redaktion lediglich die presserechtliche Verantwortung. Die in diesem Magazin veröffentlichten Beiträge sind urheberrechtlich geschützt. Übersetzung, Vervielfältigung, sowie Speicherung auf beliebigen Medien, ganz oder auszugsweise nur mit genauer Quellenangabe erlaubt. Die eingereichten Beiträge müssen frei von Ansprüchen Dritter sein. Veröffentlichte Programme gehen — soweit nichts anderes vermerkt ist — in die Public Domain über. Für Text, Schaltbilder oder Aufbauzeichnungen, die zum Nichtfunktionieren oder eventuellem Schadhaftwerden von Bauelementen führen, kann keine Haftung übernommen werden. Sämtliche Veröffentlichungen erfolgen ohne Berücksichtigung eines eventuellen Patentschutzes. Warennamen werden ohne Gewährleistung einer freien Verwendung benutzt.

Liebe Leser,

Draußen ist Sommer, Badewetter, und auf meiner etwas längeren und durchaus abenteuerlichen Chinareise (Abbildung 1) habe ich mir eine süße Ablenkung angelacht — das gehört aber nicht hierher. Abenteuer, die hierher gehören, kommen von Erich Wälde, und zwar gleich zwei Stück (eines davon hat er gemeinsam mit Martin Bitter erlebt).

Trotzdem bleibt Zeit für das Setzen einer VD, und natürlich kommen dann auch noch die ganzen Probleme hoch, die das bisher so macht, und es werden Lösungen gesucht. Das vergrößerte Redaktionsteam ist ein Teil der Lösung, die Zusammenarbeit innerhalb des vergrößerten Teams die andere Herausforderung. Wenn einer allein vor sich hinwerkelt, ist die Zusammenarbeit immer am leichtesten, aber die ganze Last bleibt eben an dem einen hängen.

Wollen wir bei Subversion bleiben, das ja inzwischen auch nicht mehr der letzte Schrei ist? Der Apache auf unserm Vserver, der nur wegen Subversion und Trac noch läuft, tut auch das nicht so stabil, wie wir das wollen, und braucht immer mal wieder einen Tritt. Und die Tickets bei Trac nutzt niemand, aber dafür sind sie dann voll mit Spam. Dafür gibt's im Repository jetzt auch eine Graustufen-Variante des PDFs, das beim Begutachten des Erscheinungsbildes von eingebundenen Grafiken hilft.

Immerhin: Das größere Redaktionsteam hat anscheinend die Artikel-Klemme etwas reduziert. Johannes Teich stellt seine „Wave Engine“ vor, ein Programm, das eine Notenschrift (ASCII) in eine Wave-Datei umwandelt.

Willi Stricker strickt nach wie vor an seinem eigenen kleinen Forth-Prozessor, und entdeckt dabei, dass man jetzt auf einmal Wörter in High-Level definieren muss, die man ansonsten immer in Assembler geschrieben hat — weil der Assembler eines Forth-Prozessors eben Forth ist.

Dirk Brühl betätigt sich als Archäologe, und gräbt alte Hardware aus, die aber immer noch tut. Auch Fred Behringer beschäftigt sich mit antiken Datenspeichern namens „Disketten“, und zwar unter DOS (ich habe dazu gar kein Laufwerk mehr). Nachwuchsfragen lösen solche Artikel nicht, nostalgische Gefühle sind eher etwas für die, die von Anfang an dabei waren.

Die Arbeit an der Winter-Ausgabe 2011 ist schon gestartet. Erich Wälde übernimmt im Turnus die Aufgabe des Chefredakteurs. Carsten wird Fossil vorstellen, ein modernes, aber dennoch eher leichtgewichtiges Versionskontrollsystem, das die Arbeitsabläufe etwas vereinfachen könnte — vielleicht probieren wir das für die nächste VD schon mal aus. Wie immer werden interessante Artikel, aber auch Nachrichten aus der Forth-Welt gesucht. Autoren, die L^AT_EX können, reichen am besten gleich so ein. Das Redaktions-Team der Vierten Dimension wird unter der E-Mail-Adresse 'vd@forth-ev.de' erreicht.

Auf dem Titelbild sind schwingende Luftsäulen zu erkennen, Abbildung aus Wikipedia.

Viele Grüße und viel Spaß beim Lesen dieser Ausgabe, *Bernd*

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/filemgmt/index.php>

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

Ulrich Hoffmann Kontakt: Direktorium@Forth-ev.de

Bernd Paysan

Ewald Rieger



Adventures in amforth: Wortliste schließen

Erich Wälde

Beim Lesen in den bekannten Forth-Büchern ([2], [3], [4]) bekommt man vorgeführt, dass getrennte Wortlisten vom Programmierer eingesetzt werden können. Normalerweise ist nur eine Wortliste mit dem Namen **FORTH** vorhanden und aktiv. **amforth** kann gemäß dem ANS94 Standard gleichzeitig acht aktive Wortlisten benutzen (seit Version 3.3).

Der Programmierer kann eine spezielle Wortliste erstellen und diese schließen. Der Anwender kann danach nur noch die Worte in dieser Liste benutzen, das Forthsystem ist für ihn nicht mehr benutzbar. Der gezeigte Quelltext wurde auf **amforth** 4.2 getestet.

Vorarbeiten

Ohne Änderung des **amforth**-systems sind die Worte **get-current**, **get-order** und **order** verfügbar. Sie werden von **amforth** intern benutzt. Allerdings ist die Bedeutung der beispielsweise von **get-order** auf den Stapel gelegten Zahlen nicht jedem sofort ersichtlich.

Um mit Wortlisten sinnvoll arbeiten zu können, fügt man in der Datei **dict_appl.inc** folgende Zeile hinzu:

```
.include "dict_wl.inc"
```

Dadurch werden die Worte **set-current**, **set-order**, **only**, **forth-wordlist**, **also**, **previous**, **definitions**, **forth** und **wordlist** verfügbar.

Wortlisten werden durch eine Zahl, die **wid**, gekennzeichnet. Die genannten Befehle verwenden diese Zahl und nicht den Namen der Wortliste. Die **wids** der aktiven Wortlisten werden in der Suchliste gespeichert. Die Suchliste liegt bei **amforth** im EEPROM und übersteht das Ausschalten des Mikrokontrollers unbeschadet.

- **forth-wordlist** ist eine Konstante (value), die die **wid** der Wortliste **FORTH** auf den Stapel legt.
- **wordlist (-- wid)** erzeugt eine leere Wortliste und legt die neue **wid** auf den Stapel.
- **get-current (-- wid)** legt die **wid** der Wortliste, in die derzeit neue Definitionen eingefügt werden, auf den Stapel.
- **set-current (wid --)** veranlasst, dass die nachfolgenden Definitionen in die Wortliste mit der angegebenen **wid** eingetragen werden.
- **definitions (--)** holt die oberste **wid** aus der Suchliste und ruft damit **set-current** auf. Nachfolgende Definitionen werden in diese oberste Wortliste eingetragen.
- **get-order (-- wid0 .. widN n)** produziert die **wids** aller aktiven Wortlisten und deren Anzahl.
- **set-order (wid0 .. widN n --)** speichert die angegebenen **wids** in dieser Reihenfolge neu in der Suchliste.
- **order (--)** gibt die von **get-current** und **get-order** produzierten **wids** aus.
- **also (--)** dupliziert den obersten Eintrag in der Suchliste und verschiebt die weiteren Einträge um einen Platz nach unten (wie **dup**).
- **previous (--)** löscht den obersten Eintrag in der Suchliste und verschiebt die weiteren Einträge um einen Platz noch oben (wie **drop**).

- **only (--)** ersetzt die Suchliste durch eine neue, in der lediglich die **FORTH**-Wortliste enthalten ist.
- **forth** ersetzt den obersten Eintrag der Suchliste durch die **wid** der Wortliste **FORTH**.

Neue Wortliste anlegen

Mit diesen Worten lässt sich das Wort vocabulary schreiben (siehe **lib/vocabulary.frt**):

```
1 \ lib/vocabulary.frt
2
3 \ create a vocabulary, at runtime replace
4 \ the first entry in the search-list
5 : vocabulary ( "char" -- )
6     wordlist constant
7     does>
8     i@ >r
9     get-order swap drop
10    r> swap
11    set-order
12 ;
```

vocabulary erzeugt eine Konstante, welche die **wid** einer neu erzeugten, leeren Wortliste enthält. Ruft man das Wort auf, dann wird der oberste Eintrag der Suchliste mit der genannten **wid** überschrieben. Um eine neue Wortliste **<application>** mit neuen Worten zu füllen, schreibt man also

```
vocabulary <application>
also <application> definitions
: neuesWort ." Hallo aus Wortliste <application>" cr ;
...
```

der Erfolg lässt sich mit **order** und **words** überprüfen

```
> order
40
2 40 14 ok
> words
neuesWort <application> vocabulary ...
```

Die Liste **<application>** lässt sich mit **previous** wieder aus der Suchliste verbannen:

```
> previous
ok
> order
40
1 14 ok
> words
<application> vocabulary ...
```



Adventures 8: Wortliste schließen

Die Wortliste <application> wird bei der Suche nicht mehr berücksichtigt. Neue Worte werden allerdings immer noch in diese Liste eingetragen. Erst ein Aufruf von `definitions` regelt das:

```
> definitions
ok
> order
14
1 14 ok
>
```

Wortliste schließen

Um eine Wortliste zu schließen, muss man die Suchliste komplett überschreiben. `sealed` lässt sich zunächst die Suchliste ausgeben, sichert `n` und den obersten Eintrag, und ersetzt dann alle Einträge mit dem Wert `-1`. Der oberste Wert wird dann wieder korrekt eingesetzt und die Suchliste gespeichert.

```
: sealed ( -- )
  get-order          \ wl.0 .. wl.top n
  over over swap >r >r \ store n and top wid
    0 ?do drop loop  \ clear order entries
  r@ 0 ?do -1 loop drop \ replace with -1
  r> r> swap         \ restore top wid and n
  set-order          \ write search list
;
```

Um den Erfolg von `sealed` besser zu sehen, fügen wir die Worte `order` und `words` zur Wortliste <application> hinzu. Und damit wir zum Spielen nicht in dieser kurzen Liste für alle Ewigkeit — naja, bis zum nächsten flashen des Mikrokontrollers — stranden, fügen wir außerdem einen Notausgang in die Liste:

```
also <application> definitions
: order order ;
: words words ;
: escape only definitions ;
```

```
also <application> sealed
ok
> order
40
1 40 ok
> words
escape words order neuesWort ok
> neuesWort
Hallo aus der Wortliste <application>
ok
> .s
?? -13 3
> only
?? -13 5
> 14 24 e!
?? -13 9
> escape
ok
> words
sealed <application> vocabulary ...
```

Überraschung

Beim Testen stellte sich dann heraus, dass man das Wort `sealed` gar nicht braucht. Steht in der aktuellen Suchliste nur ein Eintrag, dann überschreibt <application> diesen Eintrag, und man hat sich flugs ausgesperrt. `amforth` hat eben keine Sicherheitsnetze:

Benutzung auf eigene Gefahr!

Referenzen

1. <http://amforth.sourceforge.net/>
2. Leo Brodie — Starting Forth, 1987, Prentice Hall, S. 219ff
3. Elizabeth D. Rather — Forth Application Techniques, 2006, Forth Inc., S. 107ff
4. Gert-Ulrich Vack — Programmieren mit FORTH, 1990, Verlag Technik Berlin, S. 189ff

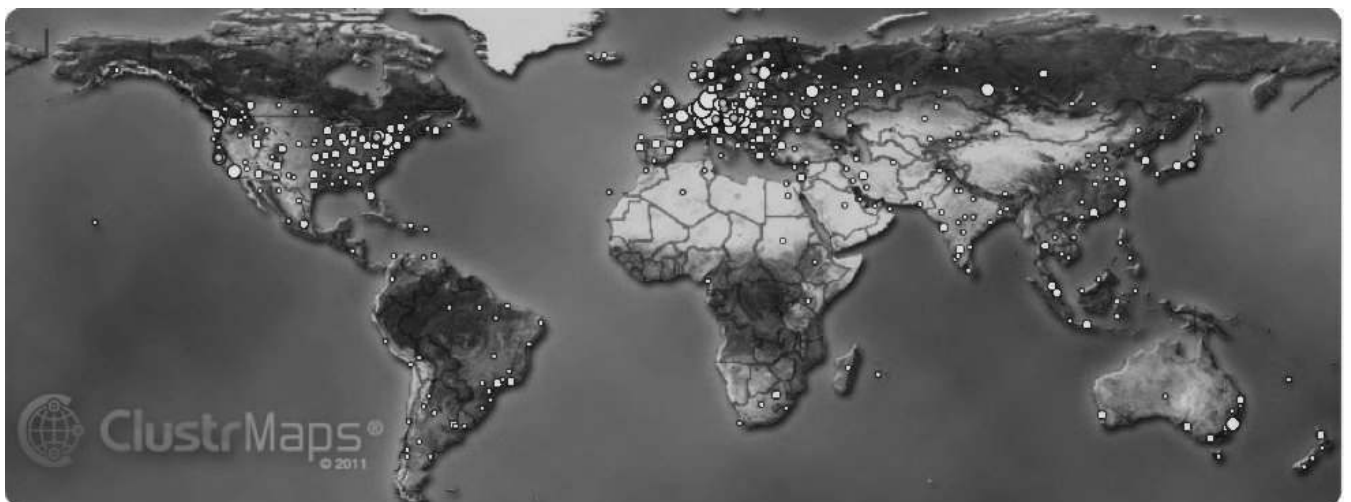


Abbildung 1: Verteilung des amforth auf der Welt

Adventures 9: Funklöcher!

Erich Wälde und Martin Bitter

Es kommt der Tag im Leben eines Kontrollettis¹, wo es was zu messen gibt, an einer Stelle, an der ein Kabel (Strom, Kommunikation) keine Option ist. Das ist der Tag, wo er sich vielleicht daran erinnert, dass es bei Pollin [2] so nette kleine Module mit dem Namen rfm12 gibt, die auf den ISM-Funkbändern senden und empfangen. Also flugs welche bestellt und bekommen und damit geht der Ärger richtig los. Stellt sich heraus, dass die Module kleine Primadonnen sind, die richtig gestreichelt werden wollen, bevor das mit der hauseigenen remote sensing-Lösung einigermaßen funktioniert. Dieser Artikel beleuchtet die Grundlagen. Ein weiterer Artikel zeigt, wie man käufliche Außensensoren für übliche Wetterstationen (Technoline, 3) erfolgreich belauscht. Ein dritter Teil wird zeigen, wie man seine eigene Funkstrecke aufzieht, beispielsweise um damit den aktuellen Wasserstand aus der dunklen Zisterne im Hellen zu vermelden.

Grundsätzliches

Die Vergabe des Funkspektrums wird bekanntlich von diversen, strengen Behörden geregelt. Für den *freieren* Gebrauch gibt es vorgegebene Frequenzbänder, etwa die sogenannten ISM (*i*ndustrial, *s*cientific, *m*edical)-Bänder [4], von denen zwei bei 434 MHz und 868 MHz liegen. Die Übertragung kann in Frequenzmodulation stattfinden, z.B. wählt man einen Frequenzhub von ± 90 kHz. Damit können zwei Zustände 0 und 1 kodiert werden. Die zeitliche Abfolge dieser beiden Zustände leiht man sich von der seriellen Schnittstelle aus (4800 Baud).

Für die ISM-Funkbänder gibt es vorgefertigte Module zu kaufen, etwa von der Firma Hope [8]. Sie werden über den SPI-Bus an einen Mikrokontroller angeschlossen. Für den Versuchsaufbau auf dem Schreibtisch benötigt man keine Antenne, ansonsten einen Draht von ca. 17 cm Länge (entspricht $\lambda/4$ bei 434 MHz). Auf mikrocontrollernet.de [6] finden sich längliche Forumseinträge, die sich mit den Tücken der Initialisierung und des Betriebs beschäftigen [7]. Der Hersteller bietet ein Datenblatt [9] und Beispielprogramme [10] zum Herunterladen an. Einen guten Einstieg bietet auch ein Artikel in Elektor [11] vom Januar 2009.

Verbindung zum Funkmodul

Bevor man hier vorschnell zum LötKolben greift, und den delikaten Seitentaschen der Module irgendwelche Drähte verpasst: Die Seitentaschen brechen bei zu viel Kraft aus und dann ist leider Schluss mit lustigen Kontakten. Es empfiehlt sich, die ebenfalls erhältlichen Adapterplatinen zu verwenden.

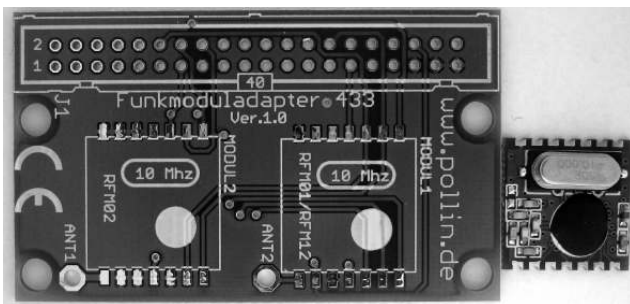


Abbildung 1: Funkmodul und Adapterplatine

Außerdem muss an dieser Stelle gesagt werden, dass es die Funkmodule in dreierlei Ausführung gibt: den Sender rfm01, den Empfänger rfm02 und den Transceiver rfm12, welcher senden und empfangen kann, wenn auch nicht gleichzeitig. Alle hier beschriebenen Experimente wurden mit dem rfm12 gemacht. Die Fähigkeiten der verschiedenen Module unterscheiden sich minimal, aber entscheidend.

Die Funkmodule werden über den SPI-Bus [5] angeschlossen. Der Controller fungiert als *master* und initiiert alle Datentransfers. Daten zum Funkmodul werden über den Pin MOSI (*master out slave in*) übertragen. Gleichzeitig werden Daten vom Funkmodul am Pin MISO (*master in slave out*) empfangen. Der Transfer wird mit dem vom *master* getriebenen Pin SCK synchronisiert. Jeder Puls auf SCK erzeugt den Transport von einem Bit. Damit sich das Funkmodul angesprochen fühlt, legt der Controller die *chip-select*-Leitung /CS des Funkmoduls auf *low*.

In der einfachsten Konfiguration braucht man lediglich vier Verbindungen zwischen Controller und Funkmodul:

MOSI	→	DIN
MISO	←	DOUT
SCK	→	CLK
PORTC.2	→	/CS

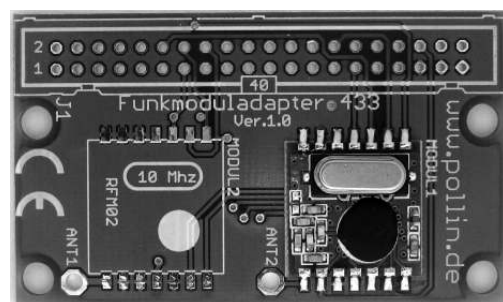


Abbildung 2: Funkmodul auf Adapterplatine aufgelötet

Man kann den Pin /CS auch auf den Pin /SS des Controllers legen. Das hat aber den Nachteil, dass das Funkmodul beim Programmieren des Controllers dazwischenredet, wenn man nicht mit Widerständen dagegen vorgeht. Ich habe mich entschieden, den /CS des Funkmoduls über den Pin C 2 anzusteuern. Der Preis ist allerdings, dass ich

¹ a. zwanghaft Kontrolle ausübender Mensch; b. Mikrokontroller programmierender Mensch; c. Geld- oder Erbsenzähler



die JTAG-Verbindung mit `-jtag` stilllegen muss. Selbstverständlich besteht freie Auswahl, was den für /CS verwendeten Kontrollerpin angeht.

```
PORTC 2 portpin: _rfm12 \ rfm select, needs -jtag
PORTB 5 portpin: _mosi
PORTB 6 portpin: _miso
PORTB 7 portpin: _clk
```

Mit dem Funkmodul reden

Die SPI-Schnittstelle des Mikrokontrollers wird in den *master mode* gebracht. Dazu muss man den Pin /SS unbedingt vorher auf *high* setzen, sonst lässt sich der *master mode* nicht aktivieren.

```
\ ewlib/spi.fs
PORTB 4 portpin: /ss
: +spi ( -- )
  \ ss high          \ activate pullup!
  _mosi high _mosi pin_output
  _clk low _clk pin_output
  \ _miso pin_pullup_on \ not needed, see datasheet
  $53 SPCR c! \ enable, master mode, f/128 data rate
;
: -spi ( -- ) 0 SPCR c! ;
```

Das Funkmodul wird mit 2-Byte-Daten angesprochen. Der Befehl `spirw` überträgt aber nur ein Byte. Also definieren wir dazu das Wort `><spi`: Das Argument (1 Zelle, TOS) wird in zwei Bytes zerlegt, die beiden Bytes werden nacheinander übertragen und die Antwortbytes wieder zu einer Zelle zusammengefügt.

```
\ transfer 1 cell
: ><spi ( x -- x' )
  dup >< spirw
  swap spirw
  swap >< +
;
```

Um mit dem Funkmodul zu reden, aktiviert man zuerst den *chip select* (`_rfm12 low`) und wartet dann darauf, dass das Funkmodul die Leitung DOUT auf *high* legt. Danach darf man die nächsten zwei Bytes übertragen. Die Übertragungsroutinen sehen so aus:

```
\ wait for wireless data tx ready
: w? ( -- ) _rfm12 low begin _miso pin_high? until ;
\ write 1 cell to wireless control
: (>wc) ( x -- x' ) _rfm12 low ><spi _rfm12 high ;
\ write 1 cell to wireless control, drop reply
: >wc ( x -- ) (>wc) drop ;
\ read wireless status
: wc? ( -- x ) 0 (>wc) ;
: w.status wc? 4 hex u0.r ;
\ write wireless data (1 byte)
: >w ( c -- ) $00ff and $b800 + w? (>wc) drop ;
\ read wireless data (1 byte)
: <w ( -- c ) $b000 w? (>wc) ;
```

Das Wort `w?` wartet darauf, dass das Funkmodul seine Bereitschaft signalisiert. Das Wort `(>wc)` kontrolliert den *chip select* für das Funkmodul und benutzt `><spi`, um die Daten zu übertragen. Beispielsweise wird mit der Sequenz

```
$5aa5 w? (>wc) drop
```

der Wert `$5aa5` an das Funkmodul übertragen und die Antwort verworfen.

Initialisierung

Bevor sich aber irgendetwas rührt, muss das Funkmodul richtig initialisiert werden. Man kann die Sequenz aus dem Beispielprogramm [10] übernehmen. Das ist auch erst mal gut so, denn das Datenblatt der Funkmodule liest sich nicht sehr flüssig.

```
: rfm12.init
  $80d7 >wc \ EL, EF, 434 MHz band, 12.0pF
  $82d9 >wc \ !er, !ebb, ET, ES, EX, !eb, !ew, DC
  $a67c >wc \ 434.150 MHz; Band Mitte: 433.920 MHz
  $c647 >wc \ 4800 baud
  $94a0 >wc \ VDI, FAST, 134 kHz, OdBm, -103dBm
  $c2ac >wc \ AL, !m1, DIG, DQD4
  $ca81 >wc \ FIFO8, SYNC, !ff, DR
  $c483 >wc \ @PWR, NO RSTRIC, !st, !fi, OE, EN
  $9850 >wc \ !mp, 90kHz, MAX OUT
  $e000 >wc
  $c800 >wc
  $c000 >wc
;
```

Schon an der Anzahl der Bytes kann man sehen, dass man sich hier leicht vertun kann. Das Ergebnis ist dann fast immer, dass die Übertragung nicht funktionieren will. Erschwerend kommt dazu, dass wir dabei nichts *sehen* können.

Rauschen empfangen

Solchermaßen ausgestattet versuchen wir, ob unser Funkmodul etwas empfängt. Bei der Initialisierung des Funkmoduls ist es zwingend notwendig, den Status des Moduls zu lesen. Damit werden die Interrupts des Moduls gelöscht. Wenn sich beim Probieren so gar nichts rührt, dann kann das Lesen des Status die Angelegenheit deutlich verändern.

```
: rfm12.rx $82d9 >wc ;
: rfm12.off $8201 >wc ;
```

```
: rfm12.rx.all
  -jtag
  +spi +rfm12
  rfm12.init w.status cr
  rfm12.rx
  $ca87 >wc \ ???
  hex
  begin
    <w 2 u0.r space
  key? until
  key drop
  rfm12.off
;
```

Man stellt fest, irgendetwas wird empfangen. Und das ist auch gleich schon die erste Erkenntnis: es wird *immer irgendetwas* empfangen. Und wenn es nur das große Rauschen ist.

```
> ver
amforth 4.5 ATmega32 ok
> rfm12.rx.all
0120
```



```

FF FD FF FF FF FF FD F9 FF F9 FD FF F9 FF FF
FD FF F9 FB FF FD F9 FF FF FD FF F9 FD FD FD
FF FF FF FD FF FD F9 FF FF FF FF FF FF FD
FF FF FD FD FB FF F9 FF FF FF FF FF F9 FF FD
FD FF FF FF FD F9 F9 FF FF FD FD F9 FF FF F9
FF FF F9 FD FD FD FD FF F9 FF FD FD F9 FB FF
FF FF F9 FB F9 FD FF FD F9 FB F9 F9 FF FF FD
FF FF FD F9 FF FF FB FF F9 FF FF FF FB F9 F9
FF FF F9 FF FD FD FF FF F9 FD FD F9 FF FF FD
F9 FD F9 FF FF FF FF FD FF FF FF F9 FF FF FF
FF FD F9 FF F9 FF FF FD FD FB FB FF FF FD FF
FF FF FF FB FF FD F9 FF FD FB F9 FF FF FD FF
...
FF FD F9 FF  ok
>

```

Für meinen Geschmack sind hier zu viele F drin, aber das hängt u.U. von der Mittenfrequenz und der eingestellten Empfindlichkeit ab.

Nullen senden

Dem Ätherrauschen zuzuschauen ist nicht für jeden beruhigend, also machen wir uns daran, den Äther mit Nullen zu füllen, damit sich das Rauschen ändert. Dazu braucht man eine zweite Station, die als Sender konfiguriert wird. Dabei ist zu beachten, dass es im ISM-Band nicht gestattet ist, den Sender dauernd anzuschalten. Vorgesehen ist, dass ein Sender lediglich 1 % der Zeit aktiv ist, also höchstens eine halbe Sekunde pro Minute. Für erste Tests werden wir uns daran nicht halten, aber sobald wir die Nullen gesichtet haben.

Auf der Sendestation definieren wir Worte, um den Sender bequem ein- und ausschalten zu können:

```

: rfm12.tx.on ( -- ) $8238 >wc ;
: rfm12.tx.off ( -- ) $8208 w? >wc ;
: init
  -jtag
  +spi +rfm12
  rfm12.init
  w.status cr
;

```

Auf der Empfängerstation starten wir wie oben das Empfangsprogramm `rfm12.rx.all`. Auf der Sendestation starten wir den Sender

```

> init
0120
ok
> rfm12.tx.on
ok

```

Jetzt sollten auf dem Empfänger lauter 00 Bytes empfangen werden. Nach

```

> rfm12.tx.off
ok

```

verschwinden sie wieder und das Rauschen kehrt in Gestalt von mehr oder weniger zufälligen Bytes zurück. Wenn an dieser Stelle tatsächlich Nullen empfangen worden sind, solange der Sender eingeschaltet war, dann ist die erste große Hürde geschafft: Der Empfänger sieht die Informationen der Sendestation. Stellt sich sofort danach die Frage, wie man denn nun ein richtiges Datenpaket verschickt, und nicht nur Nullen.

Ein Datenpaket versenden

Ein wohlgestaltetes Datenpaket besteht aus der Präambel (\$aa Bytes), den magischen FIFO start Bytes \$2d \$d4, den zu übertragenden Daten \$30 \$31 \$32 \$33 \$34 und einer Postambel (Post-Präambel ???). Diese gehört lediglich zum guten Ton. Auch darf man das Funkmodul nicht zu früh abschalten — die Bytes werden schließlich irgendwann verschickt und der Controller wartet nicht auf das Ende, wenn man ihm das nicht beibringt (w?). Die zu verschickende Bytefolge soll so aussehen:

```
aa aa aa aa 2d d4 30 31 32 ... 3F aa aa aa
```

Die Präambel hat die Aufgabe, dem Empfänger die Chance zu geben, sich auf ein gültiges Signal einzuregeln. Den Empfänger kann man so konfigurieren, dass er zwar zuerst das große Rauschen, und dann die Präambel empfängt, diese Daten aber verwirft. Erst der Empfang der magischen Bytes \$2d \$d4 führt dazu, dass der Empfänger den Empfang der nachfolgenden Bytes an den Controller signalisiert.

```

: rfm12.tx.data ( xN .. x1 N -- )
  rfm12.tx.on
  $aa >w \ sync pattern
  $aa >w
  $aa >w

  $2d >w \ magic bytes
  $d4 >w

  $10 0 do \ payload data
    i $30 + $b800 + w? >wc
  loop

  $b8aa w? >wc \ sync pattern
  $b8aa w? >wc
  $b8aa w? >wc

  rfm12.tx.off
;

```

Ein Datenpaket empfangen

Der Empfänger kann alle Bytes empfangen, so wie vorher gezeigt. Allerdings können die Daten bitweise verschoben sein und sind dann schlecht zu finden. Leichter geht es, wenn man den Empfänger so konfiguriert, dass er auf die oben erwähnten magischen Bytes wartet und danach Datenbytes empfängt. Um den Empfang eines Bytes anzuzeigen, wird wie beim Senden die Leitung MISO auf *high* und die Interrupt-Leitung des Funkmoduls auf *low* gezogen. Beides kann benutzt werden, um das Abholen der empfangenen Daten durchzuführen. Der Controller entscheidet, wann er genug Daten gesehen hat. Dann teilt er dem Funkmodul mit, dass es auf die nächsten magischen Bytes warten soll.

```

: rfm12.rx.clearfifo $ca81 >wc $ca83 >wc ;
: rfm12.rx.data ( n -- )
  \ start rx, wait for "2d d4" pattern
  $82c8 >wc
  rfm12.rx.clearfifo
  hex
  w? \ wait for wireless to become "ready"
  \ read n bytes

```



```
( n ) 0 ?do <w 2 u0.r space loop
;
```

Zuerst starten wir den Empfänger, welcher dann auf die magischen Bytes zum Start der FIFO wartet:

```
> init
2120
ok
> &20 rfm12.rx.data
```

Dann verschicken wir auf dem Sender einen 16 Byte langen Datenblock:

```
> rfm12.tx.data
ok
>
```

Dann solle auf dem Empfänger eben dieses Datentelegramm plus weitere 4 Bytes erscheinen (von Hand umgebrochen):

```
> &20 rfm12.rx.data
30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D
3E 3F AA AA FF FF ok
>
```

Man sieht am Ende zwei der drei angehängten AA-Bytes, dann wird der Sender ausgeschaltet, bevor das dritte verschickt wurde.

Die hier gezeigte Empfangsroutine blockiert so lange, bis tatsächlich ein Datenblock empfangen wurde. Das ist brauchbar, aber vielleicht auch unpraktisch. In einem späteren Teil wird die Verwendung einer Interrupt-Service-Routine für den externen INT0-Interrupt gezeigt. Dann kann der Mikrokontroller in der Wartezeit bequem andere Dinge erledigen.

Paketinhalt

Bei der Gestaltung des Datenblocks hat man freie Hand. Es gibt auch schon jede Menge Protokolle (z.B. [12]), die für diesen Einsatzzweck erfunden wurden. Im Moment verwende ich folgende Definition für Datenpakete mit der konstanten Länge 16 Bytes. Darin ist ausreichend Platz, um den Datensatz für einen Sensor zu übertragen.

Referenzen

1. <http://amforth.sourceforge.net/>
2. <http://www.pollin.de>
3. technoline TX3-TH o.ä.
4. <http://de.wikipedia.org/wiki/ISM-Band>
5. http://de.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
6. <http://mikrokontrollernet.de>
7. <http://www.mikrocontroller.net/articles/RFM12>
8. <http://www.hoperf.com>
9. <http://www.hoperf.com/upload/rf/RFM12.pdf>
10. http://www.hoperf.com/upload/rf/RF12_code.pdf
11. Funk für den Controller — Elektor Januar 2009 (www.elektor.de)
12. <http://www.hth.com/snap/>

Größe	
1	Adresse Sender
1	Adresse Empfänger
1	derzeit unbenutzt (Pakettyp?)
1	derzeit unbenutzt (Datenlänge?)
10	Daten
2	Checksumme

Im Pakettyp könnten Informationen von der Bauart ACK, NACK, BUSY etc. transportiert werden.

So Sachen...

Und dann gibt es natürlich noch die Dinge, die man nur sehr mühsam herausfindet. Ich konnte recht bald Datenpakete von einem Kontroller zu einem anderen senden. Allerdings funktionierte das nicht über Tage hinweg. Irgendwas blieb nach ein paar Stunden mehr oder weniger hartnäckig stecken. Mehrere Dinge habe ich mühselig herausgefunden:

- Funk ist eine analoge und keine digitale Technik. Es kommt vor, dass eine regelmäßige Übertragung, sagen wir mal jede Minute ein Datenpaket, stundenlang gut funktioniert. Dann ändert man *irgendetwas*, und nichts geht mehr, obwohl es doch gerade eben noch ging. Normalerweise hat man ja sein Programm als erstes im Verdacht, aber das ist nicht unbedingt der Fall.
- Der Abstand zwischen den Stationen spielt eine Rolle, vor allem auf dem Schreibtisch. Die Wellenlänge bei 434 MHz beträgt knapp 70 cm. Stationen, die auf dem Schreibtisch stehen, sind normalerweise näher zusammen. Die Situation kann sich stark verändern, wenn man die Stationen verrückt oder dreht.
- Ich habe mich lange gescheut, Antennen (17 cm Draht) zu spendieren, weil ja bei Abständen von ein paar 20 cm alles auch so funktioniert. Dachte ich zumindest. Und manchmal hat es ja auch funktioniert. Aber Antennen dran und die Sendeleistung reduzieren (z.B. -12 dB), dann klappt das viel besser.

Listings

```

1  \ 2011-07-13 adv9_1.fs
2
3  include lib/misc.frt
4  include lib/bitnames.frt
5  include lib/ans94/marker.frt
6  include atmega32.frt
7
8  marker --start--
9
10 PORTC 2 portpin: _rfm12 \ requires -jtag
11 PORTB 5 portpin: _mosi
12 PORTB 6 portpin: _miso
13 PORTB 7 portpin: _clk
14
15 \ /ss +spi -spi ><spi
16 include spi.fs
17 \ +rfm12 rfm12.init
18 \ >wc wc? w? >w <w w.status
19 include rfm12.fs
20
21 \ --- rfm12 ---
22 : rfm12.rx $82d9 >wc ;
23 : rfm12.off $8201 >wc ;
24
25 : rfm12.rx.all \ no sync
26 -jtag
27 +spi +rfm12
28 rfm12.init w.status cr
29 rfm12.rx
30 $ca87 >wc \ ???
31 hex
32 begin
33   <w 2 u0.r space
34   key? until
35   key drop
36   rfm12.off
37 ;
38
39 : rfm12.tx.on ( -- ) $8238 >wc ;
40 : rfm12.tx.off ( -- ) $8208 w? >wc ;
41 : init
42 -jtag
43 +spi +rfm12
44 rfm12.init
45 w.status cr
46 ;
47
48 \ send a data frame
49 : rfm12.tx.data ( xN .. x1 N -- )
50   rfm12.tx.on
51   $aa >w \ sync pattern
52   $aa >w
53   $aa >w
54
55   $2d >w \ magic bytes
56   $d4 >w
57
58   $10 0 do \ payload data
59     i $30 + $b800 + w? >wc
60   loop
61
62   $b8aa w? >wc \ sync pattern
63   $b8aa w? >wc
64   $b8aa w? >wc
65
66   rfm12.tx.off
67 ;
68
69 \ receive a data frame
70 : rfm12.rx.clearfifo $ca81 >wc $ca83 >wc ;
71 : rfm12.rx.sync $82c8 >wc ;
72 : rfm12.rx.data ( n -- )
73   rfm12.rx.sync
74   rfm12.rx.clearfifo
75   hex
76   w?
77   \ read n Bytes
78   ( n ) 0 do <w 2 u0.r space loop
79 ;

```

```

1  \ 2010-05-24 EW ewlib/spi.fs
2  \ spi, using hw interface
3  \ needs in dict_appl.inc:
4  \ .include "words/spirw.asm"
5
6  \ words:
7  \   +spi ( -- )
8  \   -spi ( -- )
9  \   ><spi ( x -- x' ) transfer 2 bytes
10
11 \ Needs at least these definitions
12 \ SPI
13 \ $2D constant SPCR \ SPI Control Register
14 \ $2F constant SPDR \ SPI Data Register
15 \ $2E constant SPSR \ SPI Status Register
16
17 \ needs lib/bitnames.frt
18
19 \ SPCR (control register)
20 \ . 7 SPIE spi interrupt enable
21 \ . 6 SPE spi enable
22 \ . 5 DORD data order, 0 msb first
23 \ . 4 MSTR master/slave mode, 1 master
24 \ . 3 CPOL clock polarity, 0 clock low on idle
25 \ . 2 CPHA clock phase, 0 sample on lead. edge
26 \ . 01 SPIR data rate, 00 f/4, 01 f/16,
27 \ 10 f/64, 11 f/128
28 \ SPE | MSTR | SPIR0 ==> $51
29
30 \ needs these defined before loading:
31 PORTB 4 portpin: /ss
32 \ PORTB 5 portpin: _mosi
33 \ PORTB 6 portpin: _miso
34 \ PORTB 7 portpin: _clk
35
36 : +spi ( -- )
37 /ss high \ activate pullup!
38 _mosi high _mosi pin_output
39 _clk low _clk pin_output
40 \ not needed, see datasheet
41 \ _miso pin_pullup_on
42
43 \ enable, master mode, f/128 data rate
44 $53 SPCR c!
45 ;
46 : -spi 0 SPCR c! ;
47

```



```

48 \ transfer 1 byte
49 \ use spirw ( c -- c' )
50
51 \ transfer 1 cell
52 : ><spi ( x -- x' )
53   dup >< spirw
54   swap spirw
55   swap >< +
56 ;
57
58 \ fin
59
60 1 \ 2011-05-29 EW ewlib/rfm12.fs
61 2 \ words: +rfm12 rfm12.init
62 3 \
63   >wc wc? w? >w
64 4
65 5 \ wait for wireless data tx ready
66 6 : w? ( -- )
67 7   _rfm12 low
68 8   begin _miso pin_high? until
69 9 ;
70
71 10 \ write 1 cell to wireless control
72 11 : (>wc) ( x -- x' )
73 12   _rfm12 low ><spi _rfm12 high ;
74 13 \ write wireless control, drop reply
75 14 : >wc ( x -- ) (>wc) drop ;
76 15 \ read wireless status
77 16 : wc? ( -- x ) 0 (>wc) ;
78 17 : w.status wc? 4 hex u0.r ;
79 18
80 19 \ write wireless data
81 20 : >w ( c -- )
82 21   $00ff and $b800 +
83 22   w? (>wc) drop
84 23 ;
85 24 \ read one byte
86 25 : <w ( -- c )
87 26   $b000
88 27   w? (>wc)
89 28 ;
90 29
91 30 : +rfm12
92 31   _rfm12 low _rfm12 pin_output
93 32 ;
94 33 : rfm12.init
95 34 \ read status register
96 35 \ wc? drop
97 36 \ init sequence

```

```

37 $80d7 >wc
38 $82d9 >wc \
39 \ Band Mitte: 433.920 MHz
40 \ $a67c >wc \ 434.150 MHz;
41 $a6b8 >wc \ 434.300 MHz
42 \ $a530 >wc \ 433.620 MHz
43 $c647 >wc \ 4800 baud
44 $94a0 >wc \ bandwidth: 134 kHz
45 \ $9480 >wc \ 200 kHz
46 $c2ac >wc
47 $ca81 >wc
48 $c483 >wc \ AFC
49 \ $c400 >wc \ AFC test without
50 $9854 >wc \ (5+1)*15 = 90 kHz Hub, -12dB
51 \ $9857 >wc \ (5+1)*15 = 90 kHz Hub, -21dB
52 \ $9894 >wc \ (9+1)*15 = 150 kHz
53 $e000 >wc
54 $c800 >wc
55 $c000 >wc
56 ;
57
58 \ Sende/Empfangsfrequenz
59 \ f = 10 * C1 * ( C2 + F/4000 ) [MHz]
60 \ 434 MHz: C1 = 1 C2 = 43
61 \ F: 96 <= F <= 3903
62 \
63 \ f = 10 * 1 * ( 43 + F[0..11]/4000 )
64 \ 10 * ( 43 + $06b8/4000 )
65 \ 10 * ( 43 + 1720/4000 )
66 \ = 434.300 MHz
67 \ f: 430.24 <= f <= 439.75 [MHz]
68 \ hub: Df = (M[0..3] + 1) * 15 kHz
69

```

Die folgenden Zeilen müssen in der Datei dict_appl.inc hinzugefügt werden:

```

1 ; --- additional words ---
2 .include "dict_wl.inc"
3 .include "words/fill.asm"
4 .include "words/1ms.asm"
5
6 .include "words/notequalzero.asm"
7 .include "words/uzerodotr.asm"
8 .include "words/udotr.asm"
9
10 .include "words/no-jtag.asm"
11 .include "words/spirw.asm"

```

Wave Engine (1)

Hannes Teich

In der letzten VD-Nummer (2011/2) habe ich unter dem Titel *Top One Partitur* das Thema kurz angeschnitten und ein Musikstück in zweierlei Notenschrift präsentiert, in üblicher und in maschinenlesbarer. Diese *Partitur* wird von einem *Forth-Programm* gelesen und in eine akustisch abspielbare wav-Datei übersetzt, die leicht ins populäre mp3-Format gewandelt werden kann.



Aus Sicht des Partitur-Verfassers handelt es sich um eine viermanualige Orgel mit großem Tonumfang, nämlich vom *Subkontra-C* mit 16,3 Hz bis zum *sechsgestrichenen C* mit 8345 Hz, also 9 Oktaven. Jedes Manual kann sechs Töne gleichzeitig spielen. Dieser PC-Orgel, ich nenne sie ab sofort *Wave Engine*, können verschiedene Klänge entlockt werden, die vorab in der Registrierung festgelegt werden. Zudem ist sie imstande, sich von der üblichen Tonstimmung etwas zu entfernen.

Verstehen wir uns recht: Hier funktioniert nichts live, man kann keine Klaviatur anschließen, und man muss nach dem Start einige Zeit auf das Ergebnis warten. Da erhebt sich natürlich die Frage, wozu das Ding gut ist, — und da wird die Sache persönlich. . .

Was mich bewogen hat

Einerseits ist es ein verschleppter Jugendtraum, wenigstens andeutungsweise die furiosen Klänge eines Les Paul nachahmen zu können, jenes Multiplayback-Gitaristen, der in den Fünfzigern mit seinem *new sound* die Musikszene aufgemischt hat (siehe z. B. www.stocket.de/WE). Davon abgesehen, war es ein nachwirkender Schock für mich, eines Tages erkennen zu müssen, dass mit unserem Tonsystem etwas nicht in Ordnung ist.

Was da nicht stimmt, ist leicht gesagt und schwer einzusehen. Im letzten Heft hatte ich auf (m)einen Artikel verwiesen, der dieses Thema näher behandelt. Auch Wikipedia tut gute Dienste. Hier nur in aller Kürze:

Ein Lamento

Musikalische Intervalle sind durch ganzzahlige Schwingungsverhältnisse definiert: Oktave 2/1, Quinte 3/2, Quarte 4/3, Durterz 5/4, Mollterz 6/5 etc. Ein Mathematiker sieht sofort: Die passen nicht nahtlos zueinander. In der Praxis werden Quinten und Terzen so verbogen, dass sie dennoch passen. Ein 85-tastiges Klavier hat einen Tonumfang von sowohl 7 Oktaven als auch 12 Quinten.

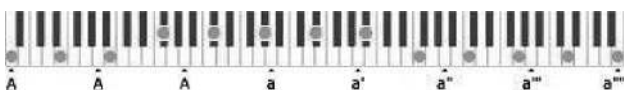


Abbildung 1: 12 Quinten = 7 Oktaven?

Der eigentliche Unterschied, das *pythagoreische Komma*, wurde wegjustiert. In der Praxis hat sich das allerdings bestens bewährt.

Verwerflich finde ich allerdings, diesen (rein akustischen) Kompromiss auf die *Semantik* zurückschlagen zu lassen, also auf die Bedeutung der *gemeinten* Töne im Zusammenhang eines Musikstücks. Aber genau das geschieht in zunehmendem Maße. Weil der Kompromiss zu einer gleichmäßigen *Zwölfteilung* der Oktave führte, denken Komponisten und Interpreten mehr und mehr in Zwölfteil-Oktaven. Das können sie sich nur leisten, weil die temperierte Stimmung sich nicht zu weit von den reinen Intervallen entfernt. Diese sind die eigentlichen Bausteine des Tonsystems. Die Oktave ist ein Paradebeispiel für die *Definition durch Verhältniszahlen*.

Das Ohr ist tolerant, die Semantik nicht. Der Einwand, so genau ginge es doch nicht zu in der Musik, greift nicht, denn die Definition einer Quinte oder Terz hat mit Genauigkeit ebenso wenig zu tun wie die Winkelsumme im Dreieck.

Was also ist zu tun?

Die „Wave Engine“ trägt solchen Bedenken dadurch Rechnung, dass sie — zusätzlich zur üblichen Stimmung — praktisch *reine Intervalle* erzeugen kann. Da aber ein System aus reinen Intervallen in ein grenzenloses Tongestrüpp ausufern würde, muss, allerdings nur akustisch, ein wenig geschummelt werden. Durch einen glücklichen Umstand führt eine Teilung der Oktave in 53 gleiche Mikro-Intervalle, *Kommas* genannt, zu nicht mehr wahrnehmbar kleinen Fehlern.

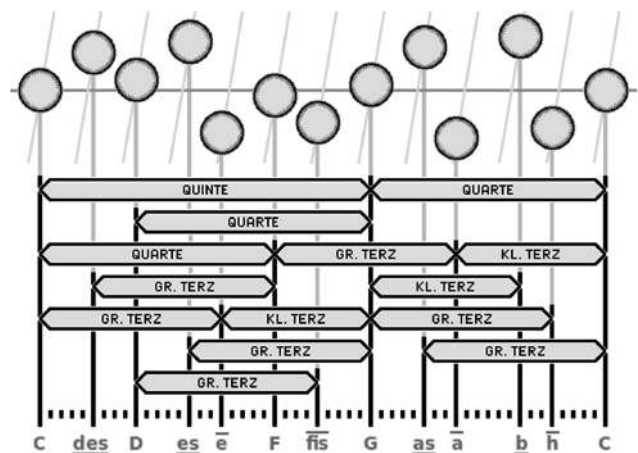


Abbildung 2: Oktavteilung mit reinen Intervallen



Hier ist die *Zwölftel-Oktave* verschwunden. *Ganztöne* und *Halbtöne* treten in mehrerlei Größen auf. Groß geschriebene Töne (C, D, F, G) sind quintverwandt, die übrigen stehen zu ihnen im Terzabstand. Ein unterstrichenes *es* erklingt ein Komma höher als *Es*, ein überstrichenes *fis* ein Komma niedriger als *Fis*. Beim Übergang in andere Tonarten ändert sich die Auswahl aus den 53 Tonhöhen entsprechend. Pythagoreische und syntonische *Kommas* bleiben erhalten (als 1/53 Oktave). Die Ballone (Abb. 2) zeigen den Versuch, durch einen Lupeneffekt grafische Ordnung in die Menge der Töne zu bringen, wobei die Waagrechte für die Klavierstimmung steht. Und damit genug der Musiktheorie. Wikipedia gibt gern weitere Auskunft (z. B. unter *Reine Stimmung*).

Der Sinus-Generator

Nun aber „in medias res!“ Was wir zunächst brauchen, ist eine Methode zur Tonerzeugung. Billige Rechteck-Kurven kommen nicht in Frage. Ein Sinus-Generator übernimmt die Erzeugung fast aller Klänge. Aber weil ein simpler Sinuston eine Zumutung wäre, muss er abklingen und mit Obertönen versetzt werden. Die *Wave Engine* spendiert dafür 15 Obertöne (das sind mitsamt Grundton 16 Teiltöne oder *partials*).

Ich beginne mit dem Wesentlichen, ohne jedes Fitzelchen zu kommentieren. Leo Brodie hat dieses Verfahren „fun-down approach“ genannt. :-)

```
\ -----
\ waver-A - Sinus-Generator für Manual A. (B, C, D ebenso.)
\ -----
\ <<< Definitionen gemeinsam für alle Manuale >>>
\ 6 constant v/m \ voices per manual
\ 0 value v/m_ \ loop index
\ 3 constant r/v \ p-rows per voice
\ 0 value r/v_ \ loop index
\ 16 constant p/r \ partials per p-row
\ 0 value r/v_ \ loop index
\ 48 constant p/v \ partials per voice
\ 0 value p/v_ \ not used
\ 288 constant p/m \ partials per manual
\ 0 value p/m_ \ # of actual partial
\ -----
\ <<< KURVENBERECHNUNG (MANUAL 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
    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
    accu-AR* f+! \ update accu (R)
  THEN
  THEN ;
\ -----
\ <<< ABKLINGVORGÄNGE (MANUAL A) >>>
: fadings-A ( --)
  p/m_ :ampl-A* f@ \ 6*16*3=288
  gap-A v/m_ :newton-A @ and \ new tone following?
  IF
    stac-A* f@ f* \ staccato
  ELSE
    v/m_ :pflag-A @ \ pause running?
    IF
      r/v_ :sust-A* f@ f* \ sustain
    ELSE
      r/v_ :fade-A* f@ f* \ fading
```

```
THEN
  THEN p/m_ :ampl-A* f! ;
\ -----
\ <<< TONGENERATOR (MANUAL A) >>>
: waver-A ( --)
  empty-accu-A \ AKKU LEEREN (A)

  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

  v/m_ p/v * \ 0 - 48 - 96 - 144 - 192 - 240
  r/v_ p/r * + \ 0 - 16 - 32
  p/r_ + \ 0 - 1 - 2 - ... - 14 - 15
  to p/m_ \ 0 ... 287

  sinus-A \ KURVENBERECHNUNG (A)
  fadings-A \ ABKLINGVORGÄNGE (A)

  LOOP ( p/r_ ) \ next partial in p-row
  LOOP ( r/v_ ) \ next p-row of voice

  phase-incr-A \ PHASE INKREMENTIEREN (A)

  LOOP ( v/m_ ) \ next voice of manual

  timer-decr-A \ TIMER DEKREMENTIEREN (A)
  store-accu-A ; \ AKKUINHALT AUFADDIEREN (A)
\ -----
```

Es gibt insgesamt:

- 4 Manuale (A, B, C, D);
- 6 Stimmen (*voices*) pro Manual;
- 3 Teiltonreihen (*p-rows*) pro Stimme;
- 16 Teiltöne (*partials*) pro Teiltonreihe.

Das ergibt 48 Teiltöne pro Stimme, 288 Teiltöne pro Manual und 1152 Teiltöne insgesamt. Ganze zwei der drei Teiltonreihen sind nur für die Einschwingvorgänge zuständig, für die das Ohr besonders empfindlich ist.

Zu Beginn wird der Akku geleert:

```
: empty-accu-A 0e accu-AL* f! 0e accu-AR* f! ;
```

Der Tongenerator für das Manual A (*waver-A*) durchläuft drei geschachtelte DO...LOOP-Schleifen, die äußere für 6 Stimmen pro Manual, die mittlere für 3 Teiltonreihen pro Stimme, die innere für 16 Teiltöne pro Reihe. Die Schleifen-Indizes werden in Values geladen, das erschien mir am übersichtlichsten.

Die Daten aller Teiltöne werden der Kurvenberechnung *sinus-A* zugeführt, wo die Sinuspunkte ermittelt werden. Die Aktion *multiply by partial#* ist nötig, weil die höheren Teiltöne den Sinus entsprechend schneller durchlaufen. Der berechnete Sinuswert wird sodann mit dem zugehörigen aktuellen Amplitudenwert multipliziert und im Akku mit den anderen Stimmen aufaddiert.

Es gibt dreierlei Dämpfungsvorgänge: das normale Abklingen eines gezupften Tons, „*fading*“ genannt, dann das Nachklingen eines nominell beendeten Tons, „*sustain*“, sowie das, was ich „*staccato*“ genannt habe.



Abbildung 3: Dreierlei Dämpfungstypen

Hier sind alle drei zu sehen: Die Amplitude schwindet langsam („fading“), am Ende sieht man den Nachklang eines beendeten Tons („sustain“), und zwischen den Tönen die „staccato“-Dämpfung, ohne die unliebsame hörbare Spitzen auftreten würden. „staccato“ wird nur vor einem neuen Ton gebraucht, nicht vor einer Pause — da gilt „sustain.“ Deshalb ist eine Vorausschau („new tone following?“) nötig, beiläufig erwähnt.

Die Anfangs-Amplituden der Teiltöne sind durch die Registrierung festgelegt. Nicht benutzte Stimmen könnten eigentlich unterdrückt werden, im Moment durchlaufen sie noch den Generator, den das nicht weiter stört.

Phase inkrementieren erfolgt nur bei den Grundtönen, denn die Obertöne hängen als Vielfache davon ab.

```
: phase-incr-A
  v/m_ :freq-A* f@      \ frequency
        vibr-A* f@ f/    \ 1-x (pitch down)
  v/m_ :phase-AL* f+!    \ increment phase (L)

  v/m_ :freq-A* f@      \ frequency
        vibr-A* f@ f*    \ 1+x (pitch up)
  v/m_ :phase-AR* f+! ;  \ increment phase (R)
```

Die sechs Stimmen pro Manual haben jede ihre eigene Sinus-Phase. Die kleine Korrektur der Tonhöhe („pitch-down“, „pitch-up“) für die beiden Stereo-Kanäle bewirkt eine leichte *Schwebung*, wie man sie vom Klavier kennt, das für einzelne Töne sogar drei leicht gegeneinander verstimmte Saiten vorsieht.

Der Timer (**timer-A** in **waver-A**) bestimmt die Dauer eines Tons. Hier wird er heruntergezählt:

```
: timer-decr-A    -1 timer-A +! ;
```

Am Ende werden die Akkuwerte der vier Manuale zusammengemischt (mono oder stereo):

```
: store-accu-A    accu-AL* f@    accu-L* f+! \ mono & stereo
                  stereo_ IF    accu-AR* f@    accu-R* f+! \ stereo only
                  THEN ;
```

Damit ist Generator A beschrieben. Die Manuale B, C und D werden separat, aber analog bedient.

Aufgerufen werden die vier Routinen wie folgt:

```
: waver ( --)      0e accu-L* f!
                   0e accu-R* f!
                   manual-A_ IF waver-A THEN
                   manual-B_ IF waver-B THEN
                   manual-C_ IF waver-C THEN
                   manual-D_ IF waver-D THEN
                   accu-L* f@
                   stereo_ IF accu-R* f@
                   THEN write-frame ;
```

Die Wave-Datei

Mit **write-frame** kommen wir zum Thema *Aufbau der Wave-Datei*. Zur Illustration eine lächerlich kleine Datei im little-endian-Format, die als „Musik“ nur 8 Bytes enthält, nämlich \$1111, \$2222, \$3333, \$4444.

```
-----
0:  52 49 46 46  5C 00 00 00  57 41 56 45  66 6D 74 20
10: 10 00 00 00  01 00 02 00  44 AC 00 00  88 58 01 00
20: 04 00 10 00  64 61 74 61  08 00 00 00  11 11 22 22
30: 33 33 44 44  4C 49 53 54  28 00 00 00  51 57 45 34
40: 34 20 2D 20  51 27 73 20  57 61 76 65  20 45 6E 67
50: 69 6E 65 20  31 37 2D 61  75 67 2D 30  38 20 31 33
```

```
60:  3A 33 30 20
```

```
-----
0 0000 52494646 RIFF                                groupId
4 0004 5C000000 92 << to be patched                waveChunkSize
8 0008 57415645 WAVE                                riffType
-----
12 000C 666D7420 fmt                                formatChunkID
16 0010 10000000 16                                formatChunkSize
20 0014 0100    1 no compr                          wFormatTag
22 0016        0200 2 stereo (mono: 1)              wChannels
24 0018 44AC0000 44100 (draft: 11025)              dwSamplesPerSec
28 001C 10B10200 176400 (draft: 44100)             dwAvgBytesPerSec
32 0020 0400    4 (bytes per frame)                wBlockAlign
34 0022        1000 16 (blkalign/chnl*8)            wBitsPerSample
-----
36 0024 64617461 data                                dataChunkID
40 0028 08000000 8 << to be patched                 dataChunkSize
44 002C 1111    $1111                                (sample 1 left)
46 002E        2222 $2222                            (sample 1 right)
48 0030 3333    $3333                                (sample 2 left)
50 0032        4444 $4444                            (sample 2 right)
-----
52 0034 4C495354 LIST                                listChunkID
56 0038 28000000 40 << to be patched                 listChunkSize
60 003C 51574534 QWE44 - Q's Wave                   (text)
64 0040 34202D20
68 0044 51277320
72 0048 57617665
76 004C 20456E67 Engine                             (text)
80 0050 696E6520
84 0054 31372D61 17-aug-08 13:30                    (text)
88 0058 75672D30
92 005C 38203133
96 0060 3A333020
-----
```

Der Anfang ist immer gleich: RIFF, Gesamt(rest)länge und WAVE. Es folgen die drei Abschnitte („Chunks“ genannt): Format, Daten und Text.

Im Format-Chunk (**fmt**) wird gewählt: Mono/Stereo und Nomalbetrieb/Draft (44 KHz oder 11 KHz). Die Werte für **wFormatTag**, **wBlockAlign** und **wBitsPerSample** sind hier unveränderlich.

Der Daten-Chunk (**data**) läutet die Musik ein: Sie wird in einer Folge von 16-bit-Frames abgelegt und enthält einfache Werte, quasi wie eine digitalisierte Schallplattenrille.

Im List-Chunk (**LIST**) wird Text versteckt, der nur durch einen Dump zum Vorschein kommt.

Während des Aufbaus der Wave-Datei sind die Längen der Chunks nicht bekannt, daher müssen sie später gepatcht werden.

Wave-Dateien kennen noch weitere Chunks, die hier nicht gebraucht werden — Wikipedia weiß da mehr.

Mit **write-frame** stehe ich noch in der Pflicht:

```
: write-frame ( stereo: r-L r-R --- | mono: r-L --)
  mono_ IF    f>s minimax w>buf \ in Datei schreiben
              2 data-bytes +!
  ELSE       fswap
              f>s minimax \ Amplitude begrenzen (L)
  [ hex ]    FFFF and \ low word
              f>s minimax \ Amplitude begrenzen (R)
  [ hex ]    10000 * + \ high & low word
  [ decimal ] n>buf \ in Datei schreiben
              4 data-bytes +! \ Datenbytes aufaddieren
  THEN ;
```

Die Werte bewegen sich (mit 16 bit pro Kanal) zwischen -2^{15} und $+2^{15} - 1$. Mit **minimax** werden diese Grenzen erzwungen. **w>buf** ruft **\$>buf** auf und schreibt damit in



einen Zwischenpuffer, der sich bei Überlauf in die Wave-Datei ergießt.

```
\ Schreibpuffer laden und ggf. in die Wave-Datei über-
\ tragen. So lange die Daten nicht in den Puffer passen,
\ wird der Puffer randvoll gefüllt und sodann in die
\ Wave-Datei entleert. Dann wird der Datenrest in den
\ Puffer geladen.
: $>buf ( addr len --)
  BEGIN  bufsize bufcnt @ - ( free) >r
        dup r@ >= \ len >= free ?
  WHILE  over wbuffer bufcnt @ + r@ move \ transfer
        swap r@ + swap \ update addr
        r> - \ update len
        wbuffer bufsize >file \ empty wbuffer
        0 bufcnt ! \ zero byte count
  REPEAT r> drop \ discard free
        dup \ len > 0 ?
  IF >r wbuffer bufcnt @ + r@ move \ transfer
        bufcnt @ r@ + bufcnt ! \ update byte count
        bytecnt @ r> + bytecnt ! \ dataChunk byte count
  ELSE 2drop \ discard addr len
  THEN ;

\ 4-byte-Wert in den Schreibpuffer schreiben
: n>buf ( u --) tmp ! tmp 4 ( addr len) $>buf ;

\ 2-byte-Wert in den Schreibpuffer schreiben
: w>buf ( u --) tmp ! tmp 2 ( addr len) $>buf ;

\ 1-byte-Wert in den Schreibpuffer schreiben
: c>buf ( u --) tmp ! tmp 1 ( addr len) $>buf ;
```

Partitur: Die Manualzeilen (Notenzeilen)

Das Thema Partitur bringt mich in ein Dilemma: Es hat mir das meiste Kopfzerbrechen bereitet und hat mit Forth eigentlich nichts zu tun. In der ersten Version der *Wave Engine* konnten noch Forth-Worte eingestreut werden. Das habe ich verworfen, mit Rücksicht auf Nicht-Forther und um die Eingabe fehlerfester zu machen.

Ich möchte nun so vorgehen, dass ich die Regeln für die Partitur quasi neu entwickle, denn das Ringen um Lösungen ist doch eigentlich der interessanteste Teil eines Projekts. Bevor die Angelegenheit forthig werden kann, muss die Aufgabenstellung klar sein, sonst hängt der Code in der Luft. Beginnen wir mit den *Musiknoten*, die aufwändige Registrierung wird dann später behandelt.

Die Partitur ist eine einfache Textdatei. Ich gebrauche im Folgenden die Bezeichnung *Partiturzeilen*, wobei diese Zeilen unabhängig von Textdateizeilen sind. Sie können sich über mehrere Textdateizeilen erstrecken oder auch zu mehreren in einer Textdateizeile Platz finden.

Irgendwann habe ich mich entschlossen, Partiturzeilen durch geschwungene Klammern kenntlich zu machen. Die wichtigsten Partiturzeilen sind die *Manualzeilen* für die vier (virtuellen) Manuale:

```
A{ ... }   B{ ... }   C{ ... }   D{ ... }
```

Enthalten müssen sie in erster Linie *Musiknoten*, die aus Tonhöhe und Tondauer bestehen. Um Platz zu sparen, wird die Tondauer nicht jeder einzelnen Note zugeordnet, sondern gilt für alle folgenden Noten. Daneben gibt es den Unterstrich _ für eine *Pause* und die Tilde ~ für die *Verlängerung* einer vorhergehenden Note.

Als Beispiel hier der Anfang von „Hänschen klein“ in C-Dur auf Manual A:

```
A{ :1/4  4G 4E 4E ~ 4F 4D 4F ~ 4C 4D 4E 4F 4G 4G 4G ~ }
```

Es handelt sich um Viertelnoten, zwischen die mit Hilfe der Tilde drei Halbenoten eingestreut sind. Die Ziffer 4 gibt die Oktave an, in der sich der Song abspielt.

„0C“	bedeutet „C	(Subkontra-Oktave)	16,3 Hz
„1C“	bedeutet „C	(Kontra-Oktave)	32,6 Hz
„2C“	bedeutet C	(Große Oktave)	65,2 Hz
„3C“	bedeutet c	(Kleine Oktave)	130,4 Hz
„4C“	bedeutet c'	(Eingestrichene Oktave)	260,8 Hz
„5C“	bedeutet c''	(Zweigestrichene Oktave)	521,6 Hz
„6C“	bedeutet c'''	(Dreigestrichene Oktave)	1043,1 Hz
„7C“	bedeutet c''''	(Viergestrichene Oktave)	2086,3 Hz
„8C“	bedeutet c'''''	(Fünfgestrichene Oktave)	4172,5 Hz
„9C“	bedeutet c''''''	(Sechsgestrichene Oktave)	8345,0 Hz

Das ist der Tonumfang von 9 Oktaven. Die Frequenzangaben leiten sich vom Kammerton a' mit exakt 440 Hz ab.

Mit diesen Vereinbarungen können wir schon fast musizieren, aber ein bisschen was fehlt noch. So muss dafür gesorgt sein, dass die Manuale zugleich erklingen wie bei einer echten Orgel. „Hänschen klein“ mit Bass-Begleitung kann etwa so aussehen:

```
A{ :1/4  4G 4E 4E ~ 4F 4D 4F ~ 4C 4D 4E 4F 4G 4G 4G ~ }
D{ :1/4  3C _ 3C _ 2G _ 2G _ 3C _ 2G _ 3C _ 3C _ }
```

Da die Zeilen „normal“ gelesen werden, ist klar, dass die erste Zeile zwischengespeichert werden muss, um mit der zweiten zugleich zu erklingen. Und es ist auch klar, dass beide Zeilen zeitlich gleich lang sein müssen, da sonst alles Weitere nicht mehr synchron laufen kann. Ich habe vor, bei Verletzung der Sorgfaltspflicht einen automatischen Ausgleich am Zeilenende einzubauen, aber den gibt es noch nicht.

Damit zusammenhängend besteht die Forderung, dass die Tondauer-Angaben in Bruchform ohne Rundungen auskommen, da sonst die gemeinsame Zeilenlänge nicht garantiert ist. Ich habe gegrübelt und folgendes beschlossen: Das Maß für die Tondauer wird gerastert mit einer Auflösung von 1/2880 einer Ganznote. Und nicht etwa eines Taktes, denn ein Dreivierteltakt bringt nur drei Viertelnoten unter. Das Programm sorgt dafür, dass nur Nenner zugelassen werden, die in 2880 ganzzahlig enthalten sind. Die Zähler sind beliebig. Wichtig war mir, dass sich 2880 durch 9 teilen lässt, der Triolen wegen.

:1/4 ist gleichbedeutend mit :720/2880 oder nur :720. Des Weiteren stellte sich in der Praxis heraus, dass eine *Justage der Notendauern* erforderlich ist. Bei der Phrasierung von Tonfolgen klingt „dadldadl“ einfach besser als „dadadada.“ Also sind zwei Klangfarben im Wechsel nötig, und zudem sollen „da“ und „dl“ verschiedene Länge aufweisen können, um den Melodierhythmus nicht steif und zackig erklingen zu lassen. Es „swingt“ bzw. „groovt“ dann einfach nicht. Die Freiheiten, die sich ein menschlicher Interpret nimmt, müssen in der Partitur mitformuliert werden, leider (oder auch nicht).

Die „dadldadl“-Notenpaare müssen also justiert werden:

```
A{ :1/8+90 4C :1/8-90 4D :1/8+90 4E :1/8-90 4F }
```

Die erste Note wird um 90 Einheiten verlängert, die zweite um 90 Einheiten verkürzt, und so weiter. Hier eine vereinfachte Schreibweise, die auch mit längeren Rhythmus-Ketten funktionieren würde:

A{ :1/8+90,1/8-90; 4C 4D 4E 4F }

Die Schreibweise der Noten ist noch nicht komplett. 3C 3D 3E 3F 3G 3A 3B 3C — das ist die *C-Dur*-Tonleiter in der 3. (der „kleinen“) Oktave. Dazu kommen aber noch die „schwarzen Tasten.“ Die Note *fis* wird als 3F#, *ges* als 3Gb geschrieben. Das deutsche *h* wird zu 3B, das deutsche *b* zu 3Bb — das entspricht der internationalen Gepflogenheit.

Und weil wir gerade dabei sind (siehe Abb. 2): Ein unterstrichenes *des*, das um ein Komma höher erklingt als ein *Des*, wird in der eingestrichenen Oktave als 4Db\, und ein überstrichenes *fis*, das um ein Komma tiefer klingt als ein *Fis*, als 4F#/ geschrieben. Hier noch ein Vorschlag für übliche Notenschrift:



Abbildung 4: Feinjustage für (fast) reine Stimmung

Ein Manual muss auch *mehrstimmig* spielen können. Dafür braucht es eine Notation, und die sieht so aus: Was von eckigen Klammern umschlossen wird, ertönt zugleich. In diesem Fall zwei Stimmen:

```
A{ :1/4 [4G 4E] [4E 4C] [4E 4C] [~ ~ ] }
A{ [4F 4D] [4D 3B] [4F 3B] [~ ~ ] }
A{ [4C 3E] [4D 3G] [4E 4C] [4F 4D] }
A{ [4G 4E] [4G 4E] [4G 4E] [~ ~ ] }
```

Geltungsbereiche, auch schachtelbar, von Notenlängen und/oder Habitus-Marken werden rundgeklammert:

```
A{ :1/4 *1 4C 4D 4E ( :1/8 *2 4F 4G 4A 4B ) 5C 4C }
```

Dann gibt's noch (schachtelbare) Schleifen. Die vier Töne in spitzen Klammern werden zweimal gespielt:

```
A{ *1 :1/4 4C 4D 4E 4F <2 4G 4A 4B 5C > 5D 5E 5C 4C }
```

Damit sind die Manualzeilen fast beschrieben — bis auf eine Kleinigkeit, die ich *Habitus* genannt habe. Das ist eine simple Marke (Beispiel: *12), die eine vorher definierte Umregistrierung vornimmt oder (im einfachsten Fall) einen Akzent für die nächste Note setzt, diese also etwas lauter ertönen lässt als ihre Nachbarn.

Und das war's auch schon mit den Manualzeilen. Dies:

```
fix{ ... } set{ ... } reg{ ... }
{{ title }} [[ comment ]] [[[ comment ]]]
```

sind weitere Partiturzeilen-Typen. Dazu später mehr.

So werden Arrays gebastelt:

```
\ Ein 4-byte-Array :abc und der Zugriff darauf.
\ +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
\ | >0< | >1< | >2< | >3< | >4< | >5< |
\ +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
\ create abc| 6 4 * allot \ 6 4-byte-Werte
\ : :abc ( cell# -- addr) \ cell# = 0...5
\ 4 * abc| + ;
\
\ 111 0 :abc ! ok \ 1. Wert schreiben
\ 666 5 :abc ! ok \ 6. Wert schreiben
\ 0 :abc @ . 111 ok \ 1. Wert lesen
\ 5 :abc @ . 666 ok \ 6. Wert lesen

\ Ein Floatpoint-Array :xyz* und der Zugriff darauf.
\ +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
\ | >0< | >1< | >2< |
```

```
\ +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
\ create xyz*| 3 8 * allot \ 3 8-byte-Werte
\ : :xyz* ( cell# -- addr) \ cell# = 0...2
\ 8 * xyz*| + ;
\
\ 111e 0 :xyz* f! ok \ 1. Wert schreiben
\ 666e 2 :xyz* f! ok \ 3. Wert schreiben
\ 0 :xyz* f@ f. 111 ok \ 1. Wert lesen
\ 2 :xyz* f@ f. 666 ok \ 3. Wert lesen
```

Goodies

Die folgenden Funktionen stecken in einer Prelude-Datei, die auch anderswo zum Einsatz kommt.

```
\ -----
\ Increment a cell in memory.
\ a may be >' value-name >body<.

: incr ( a --) dup @ 1+ swap ! ;
\ -----
\ Check a char for matching with a given char table
\ (counted string). If matching, return the character,
\ otherwise zero. Called by _leading and _trailing .

: matching? ( c a -- c|0) ( c a)
dup 0= ( c a f)
IF ." matching? error" QUIT
THEN count ( c a u)
0 ?DO over swap ( c c a)
dup i + c@ ( c c a c')
rot xor 0= ( c a f)
IF nip i + c@ UNLOOP EXIT THEN ( char)
LOOP 2drop 0 ; ( zero)
\ -----
\ Discard leading blanks (sp, tab) from a character string.

here 2 c, 32 ( sp) c, 9 ( tab) c, constant blanks
\ same as: create blanks 2 c, 32 ( sp) c, 9 ( tab) c,

: _leading ( $a1 u1 -- $a2 u2)
dup 0= IF EXIT THEN
BEGIN over c@ blanks matching? ( a u c|0)
WHILE proceed ( a' u')
REPEAT ;
\ -----
\ Discard trailing blanks from a character string.
\ Like -trailing , but also skipping tabs.

: _trailing ( $a1 u1 -- $a2 u2)
dup 0= IF EXIT THEN
BEGIN 1- 2dup + c@ ( a u-1 c)
blanks matching? ( a u-1 c|0)
0= ( no blank) ( a u-1 0|-1)
UNTIL 1+ ; ( a u)
\ -----
\ In Gforth vordefiniert:
\ OPEN-FILE ( c-addr u wfileid -- wfileid wior)
\ READ-LINE ( c-addr u1 wfileid -- u2 flag wior)
\ -----
\ Read a line from a text file and store it in PAD.
\ Trailing blanks (sp, tab) are cut off.
\ Input: file-ID, memory address, memory size.
\ Output: u (number of characters successfully read,
\ not including line delimiters, if any); f for flag.
\ If the file is elapsed, flag is false, otherwise true.

: read-a-line { fileID pad padsize -- pad u f }
pad padsize fileID READ-LINE ( u f wior)
( 0=ok) drop >r ( u) ( f)
pad swap _trailing ( a u) ( f)
r> ; ( a u f)
\ -----
```



Im Interpreter wird's komplexer. Ich kommentiere (funk-

```

\ Aus 1..53 und oct# = 0..9 den Notenwert 1..478 bilden.

: justify ( val oct# -- notenwert)
    0 max 9 min swap 52 + 53 mod
    swap 53 * + 1+
    dup 478 > IF 53 - THEN ;

-----

\ Eine Musiknote (z.B. 4F#/ oder 4Eb\ ) wird verarbeitet.
\ oct# ist die Oktavnummer (0..9).
\ extract-A..G liefert die Werte der Grundtöne
\ (C= 1 D= 10 E=19 F=23 G=32 A=41 B=50).
\ Vorzeichen (#, b) verändern den Wert um 5 bzw -5.
\ Komma-Vorzeichen (\, /) verändern den Wert um 1 bzw -1.
\ Vorzeichen können auch mehrfach auftreten.
\ Der resultierende Wert wird an pitch! übergeben.
\ Called by exec-note.

: note ( a u oct# -- a' u' true)
    >r extract-A..G ( a u c|val f) ( oct#)
    not IF ." note " |ERR| THEN
    >r ( a u) ( oct# val)
    BEGIN extract-#b ( a u c|val' f)
    WHILE r> + >r ( #=5, b=-5) ( a u)
    REPEAT ( a u c)
        drop
    BEGIN extract-\/ ( a u c|val' f)
    WHILE r> + >r ( \=1, /= -1) ( a u)
    REPEAT
        drop r> r> ( a u val" oct#)
( val oct#) justify ( a u val')
( val) pitch! ( a u)
    bracket-count @
    note-type ! ( a u) \ for load-fifo
    fetch-habit ( habit!) ( a u) \ habit to coll
    fetch-durat durat! ( a u) \ durat to coll
    true pend ! ( a u) \ load-fifo
    true ; ( a u true)

-----

\ As long as string (a u) empty, read a new line,
\ skipping leading blanks

: read-multiline ( a u -- a' u')
    BEGIN dup 0= ( a u f)
    WHILE 2drop sheetfileID_ ( id)
        pad pad-size read-a-line ( a u f)
        linecount incr not ( a u f)
        IF ." read_multiline error " QUIT THEN
    REPEAT _loading ;

```

```

UNTIL ;
\ -----
\ Partitur-Zeilen. Die Notenzeilen geben flag=-1 zurück,
\ die anderen 1. (-1 bewirkt Aufruf des Generators,
\ nur nach Notenzeile sinnvoll.)

: >fixline ( a u --)      fixline 1 ;
: >setline ( a u --)      setline 1 ;
: >regline ( a u --)      regline 1 ;
: >Aline ( a u --)      man-A! noteline -1 ;
: >Bline ( a u --)      man-B! noteline -1 ;
: >Cline ( a u --)      man-C! noteline -1 ;
: >Dline ( a u --)      man-D! noteline -1 ;
: >title ( a u --)      titleline 1 ;
: >comment ( a u --)    commentline 1 ;
\ -----

\ Zulässige Partiturzeilen-Köpfe.
\ Verwendet in interpreter via extract-token.

: headers$
  ['>fixline c" fix{" ['>setline c" set{"
  ['>regline c" reg{"
  ['>Aline c" A{"      ['>Bline c" B{"
  ['>Cline c" C{"      ['>Dline c" D{"
  ['>title c" {"      ['>comment c" [{" ;

create headers headers$ , , , , , , , , , , 0 ,
\ -----
\ Hier wird die Partitur Zeile für Zeile interpretiert.
\ Flag f ist gewöhnlich true, bei '.' oder vergeblichem
\ read_a_line false. Called by main-loop.

: interpreter ( a u -- a' u' f)
  BEGIN      dup ( a u u)
             drop ( a u)
             period_ IF ." period " false EXIT THEN
  BEGIN      _leading dup 0= ( elapsed?) ( a u f)
  WHILE      2drop sheetfileID_ ( id)
             pad pad-size read-a-line ( a u f)
             linecount incr ( a u f)
             not IF false EXIT THEN ( a u 0)
  REPEAT      over c@ (.) = ( a u f)
             IF true to period_ false ( a u)
             ELSE headers extract-token ( a u token f)
                  not IF [INVALID-HDR] THEN ( a u token)
                  PERFORM ( f) ( a u f=-1|1)
             THEN dup 1 <> ( a u f f')
  UNTIL ;      ( a u f)
\ -----
\ Wenn der Generator keinen Nachschub findet, wird die
\ Generatorschleife verlassen und der Interpreter

```

```

\ aktiviert.

: generator ( --)
  BEGIN
    WHILE      loader ( f) \ load latch, success?
    waver \ load accu, build wave
    REPEAT \ anything to load?
    ; \ back to partiture
  \ -----
  \ H a u p t s c h l e i f e

: main-loop ( --)
  pad 0 ( a 0)
  BEGIN
    interpreter ( f) ( a u f)
  WHILE
    generator ( a u)
  REPEAT ;
\ -----

```

Damit soll es für diesmal genügen. Ein paar aufscheinende Funktionen (z. B. `fetch-habit`, `load-fifo` etc.) wurden nicht besprochen, das würde im Moment zu weit führen.

Elegantes Forth sieht sicherlich anders aus, das ist mir klar, denn das Programm ist nie der Bastelphase entstiegen. Viel Versuch und Irrtum. Aber wenn's dann klingt wie es soll, dann ist die Freude groß. Eine Reihe wichtiger Effekte wird noch vermisst, aber die kommen noch (und erfordern, den Generator aufzuboahren).

Die Registrierung ist ein Riesen-Thema, das beim nächsten Mal angegangen werden soll. Das wird wieder gehörig Raum für die Aufgabenstellung brauchen.

Das gesamte Programm, soweit es am Laufen ist, kann wegen seines Umfangs unmöglich abgedruckt werden. Auch bin ich mit dringender Kosmetik noch gut beschäftigt, ehe fremdes Auge auf das Machwerk fällt.

Link

Eine Hörprobe nebst Partitur und Notenblatt findet sich im Dateibereich der Website der Forth-Gesellschaft unter

<http://www.forth-ev.de/filemgmt/viewcat.php?cid=54>

Top One

Mit MuseScore erstellt – 18-jun-2011 H. Teich

grenzbichler



Abbildung 5: Top One–Notenzeile



Forth–Compiler–Hilfsbefehle als High–Level–Befehle

Willi Stricker

Der Forth–Compiler benötigt verschiedene Hilfsbefehle, die er für den Programm–Ablauf compilieren muss, die der Programmierer aber nicht „sieht.“ Sie werden compiliert anstelle „sichtbarer“ Befehle, die wiederum nicht direkt umgesetzt werden können.

Einfachstes Beispiel sind die Befehle `BRANCH` und `?BRANCH`, die der Compiler für sämtliche Strukturen in Forth einsetzt (`IF...ELSE...THEN`, `BEGIN...UNTIL`, `BEGIN...WHILE...REPEAT`). Der Programmierer benötigt die beiden Befehle normalerweise nicht.

Bei Forth–Systemen werden die im Folgenden beschriebenen Hilfsbefehle in Assembler geschrieben (aus Gründen der Geschwindigkeit). Wenn aber, wie beim `STRIP`–Forth–Prozessor, keine Assembler–Befehle zur Verfügung stehen, müssen sie in High–Level geschrieben werden.

Die Befehle beeinflussen alle den Instruction Pointer (IP), dazu folgen einige Vorbemerkungen:

Bei Aufruf eines High–Level–Befehles (also eines Unterprogramms) liegt immer die Adresse des nachfolgenden Befehls auf dem Return–Stack (Return–Adresse), die ist aber gleich dem jeweils aktuellen Wert des IP.

Man könnte damit die Befehle `IP@` und `IP!` definieren:

```
: IP@ ( -- ip ) R> DUP >R ;
: IP! ( ip -- ) R> DROP >R ;
```

Der Befehl `IP!` wäre dann ein indirekter Sprung.

Man erhält demnach durch den Befehl `R>` den aktuellen IP–Wert, muss aber dafür sorgen, dass eine gültige Return–Adresse zurückgelegt wird, anderenfalls wird beim nachfolgenden Return in die nächsthöhere Programm (Struktur)–Ebene zurückgesprungen.

Anmerkungen zur Schreibweise: Wie in der Forth–Literatur üblich, werden die Compiler–High–Level–Hilfsbefehle in Klammern geschrieben. Bei der Darstellung des Speichers werden das Name–Field und das Link–Field weggelassen.

Befehle `VARIABLE` und `CONSTANT`

Die Variable (`VARIABLE`) benutzt normalerweise die Hilfs–Routine „dovar“:

Anordnung im Speicher: `[dovar]` [Speicherplatz für die Variable]

Funktion der „dovar“-Routine: Sie gibt als Ergebnis die Adresse des nachfolgenden Speicherplatzes aus (– address), deren Inhalt dann z. B. mit dem `@`–Befehl ausgelesen werden kann, außerdem reserviert der Compiler den nachfolgenden Speicherplatz für die Variable. Wichtig: Das Programm wird an der Stelle fortgeführt, an der die Variable aufgerufen wurde (übergeordnete Ebene)!

Daraus folgt die Realisierung der Routine `dovar` in High–Level sehr einfach (address = Return–Adresse des den Befehl (`VAR`) aufrufenden Programms):

```
: (VAR) ( -- address )
\ return stack: ( address -- )
R> ;
```

Die Konstante (`CONSTANT`) benutzt normalerweise die Hilfs–Routine „docon“: Anordnung im Speicher: `[docon]` [konstanter Wert]

Funktion der „docon“-Routine: Sie gibt als Ergebnis den auf dem nachfolgenden Speicherplatz liegenden konstanten Wert aus (– constant). Wichtig auch hier: Das Programm wird an der Stelle fortgeführt, an der die Konstante aufgerufen wurde! (übergeordnete Ebene).

Daraus folgt die Realisierung der Routine „docon“ in High–Level:

```
: (CON) ( -- constant )
\ return stack: ( address -- )
R> @ ;
```

Die Befehle `CREATE` und `CREATE...DOES>`

Der Befehl `CREATE` hat in Forth eine Doppelbedeutung:

1. Er markiert die Anfangsadresse eines beliebig großen Daten–Feldes
2. Er wird benutzt in der Kombination `CREATE...DOES>`

Zu 1. Datenfeld:

Der Hilfsbefehl ist hier identisch mit dem der Variablen: „dovar“, es wird also kein neuer Befehl benötigt.

Anordnung im Speicher:

`[dovar]` [Speicherplatz beliebiger Größe]

Im Unterschied zur Variablen wird hier vom `CREATE`–Befehl kein Speicherplatz reserviert, sondern explizit vom Programmierer z. B. durch den `ALLOT`–Befehl.

Zu 2. Kombination `CREATE` und `DOES>`

Die Kombination aus `CREATE` und `DOES>` kann nur als „Definierender Befehl“ benutzt werden, z. B.

```
: XYZ CREATE ... DOES> ... ;
```

Der Befehl `XYZ` erzeugt eine Gruppe von Befehlen, bei denen der „Create“-Teil ein Feld mit Daten beliebiger Art definiert und der „Does“-Teil ein Programm zur Bearbeitung dieser Daten. Wird ein Befehl des Typs `XYZ` programmiert, dann wird lediglich das zugehörige Feld definiert und das Programm des `Does`–Teils aufgerufen.

Funktion: der `Create`–Befehl muss das Programm des `Does`–Teils aufrufen und gleichzeitig die Adresse des nachfolgenden Feldes übergeben.

Anordnung im Programm:

Create-Teil: [Adresse des Does-Teils] [Datenfeld beliebiger Größe]

Does-Teil: [DOES-Befehl] [Does-Programm]

Demgemäß muss lediglich ein Befehl (DOES) definiert werden, der dem „Does“-Teil die Daten-Adresse des „Create“-Teils übergibt (cadr = Adresse des „Create“-Teils, radr = Return-Adresse des den Create-Teil aufrufenden Befehls):

```
: (DOES) ( -- cadr )
\ return stack: ( cadr radr -- radr )
  R> R> SWAP >R
;
```

Funktion:

Beim Aufruf des Does-Programms liegt die (Daten-)Adresse des Create-Teils auf dem Return-Stack (erster R>-Befehl), darüber die Adresse des aufrufenden Programms (zweiter R>-Befehl). Der SWAP-Befehl vertauscht die Adressen, der >R-Befehl legt die Return-Adresse auf den Return-Stack zurück für den abschließenden Return, so dass nun die Adresse des Create-Teils für das Does-Programm zur Verfügung steht.

Die Befehle DO ... LOOP/+LOOP

Die Loop-Befehle sind, wie der Name besagt, Programmschleifen, die jeweils mit dem Befehl DO eingeleitet und mit dem Befehl LOOP oder mit dem Befehl +LOOP abgeschlossen werden. Es werden jeweils 2 Eingagsparameter erwartet:

Der Startindex (Index) und der Endindex (Limit). Der Compiler benutzt dazu die Hilfsbefehle (DO), (LOOP) und (+LOOP) folgendermaßen:

[(DO)] [Schleifenbefehle beliebiger Anzahl]
[(LOOP/+LOOP)] [start-address]

Der Befehl (DO) markiert den Anfang, der Befehl (LOOP/+LOOP) das Ende der Schleife, die Adresse „start address“ ist die auf den Befehl (DO) folgende Adresse (1. Befehl der Schleife).

Der vom Compiler anstelle des DO-Befehls compilierte Befehl (DO) verschiebt die beiden Parameter vom Parameter-Stack auf den Return-Stack, damit sie innerhalb der Schleife nicht „stören“ aber ständig zugreifbar sind. Auch hierbei ist zu beachten, dass es sich um einen High-Level-Befehl handelt und deswegen der oberste Wert auf dem Return-Stack die Return-Adresse (radr) enthält, die vor dem Rücksprung wieder dort liegen muss:

```
: (DO) ( limit index -- )
\ return-stack: ( radr -- limit index radr )
  SWAP R> SWAP >R SWAP >R >R ;
```

Die Verschiebung erfolgt derart, dass der Index als oberstes Wort auf dem Return-Stack liegt. Er wird als Variable benutzt, die mit jedem Schleifendurchlauf verändert wird.

Anstelle des LOOP-Befehls wird vom Compiler zunächst der Befehl (LOOP) compiliert und anschließend die

Start-Adresse der Schleife. Der (LOOP)-Befehl muss zunächst den Index weiterzählen (inkrementieren), dann prüfen, ob der Index kleiner als der Limit ist. In dem Fall muss er auf die Startadresse zurückspringen (nächster Schleifendurchlauf), anderenfalls die Schleife verlassen, indem er auf den nachfolgenden Befehl springt (die Startadresse überspringt).

```
: (LOOP) ( -- )
\ return stack: ( limit index radr --
\                                     limit index startadr / radr+2 )
  R> R> 1+ DUP R@ < \ index+1 < limit ?
  IF >R @ \ next loop -- return to start
  ELSE DROP R> DROP 2+
  \ loop end -- branch to address+2
  THEN >R ;
```

Anmerkung: In Forth wird die Schleife durchlaufen, solange der Index kleiner als der Limit ist, anderenfalls wird die Schleife abgebrochen!

Die +LOOP unterscheidet sich von der einfachen LOOP dadurch, dass der Index nicht um 1 pro Schleifendurchlauf erhöht wird, sondern um einen programmierten Wert „data“:

```
: (+LOOP) ( data -- )
\ return stack: ( limit index radr --
\                                     limit index startadr / radr+2 )
  R> SWAP DUP R> + SWAP OVER R@ <
  \ index+data < limit ?
  IF >R @ \ next loop - return to start
  ELSE DROP R> DROP 2+
  \ loop end - branch to address+2
  THEN >R ;
```

Forth stellt weitere spezielle Befehle für die Schleifenbearbeitung zur Verfügung. Zwei werden kurz beschrieben: Der Schleifenbefehl I (für Index) greift auf den Index im Return-Stack zu, ohne ihn zu verändern.

```
: I ( -- index )
\ return stack ( limit index radr -- limit index radr )
  R> R@ SWAP >R ;
```

Der Schleifenbefehl LEAVE bewirkt, dass die Schleife beim nachfolgenden LOOP oder +LOOP-Befehl unabhängig vom aktuellen Index verlassen wird. Das wird erreicht, indem der Index gleich dem Limit gesetzt wird.

```
: LEAVE ( -- )
\ return stack: ( limit index radr -- limit limit radr )
  R> R> DROP R@ >R >R ;
```

Anmerkung zu den Befehlen (VAR) und (CON) anstelle von „dovar“ und „docon“:

Die in Assembler geschriebenen Routinen „dovar“ und „docon“ werden direkt von der next-Routine aufgerufen und benutzen deren Word-Pointer (WP), ohne auf den Return-Stack zuzugreifen. Wenn man sie durch die High-Level-Befehle (VAR) und (CON) ersetzt, dann muss ihnen eine „docol“-Routine vorangestellt werden (nicht nötig beim STRIP-Forth-Prozessor)! Die Teil-Programme „docol“, „dovar“ und „docon“ werden als Routinen bezeichnet, da sie keine Forth-Befehle sind.



Wie daraus unschwer zu erkennen ist, lässt sich dieses File nicht als Grundlage zum Portieren verwenden. Es ist aber kein Problem, die Ausgabe anders zu formatieren, damit sie wieder kompiliert werden kann. Es müssen halt die Adressen rausgelassen werden. Bisher waren die Adressen wichtig, denn wenn man was Patchen wollte, musste man ja wissen, wo.

Da diese Ausgrabung nun gelungen ist, ist es mir möglich Vergleiche zwischen FIG-Forth und RSC-Forth zu ziehen. Etliches kann ja beim RSC-Forth entfallen, z.B. die Floppydisk-Routinen und die Bankexecute-Routinen. Damit wird das Ganze dann noch kompakter. Wenn man wirklich was damit anfangen will, dann muss ich mir erstmal ein EPROM erstellen, das keine Fehler hat. Das von mir verwendete hat wohl noch fehlerbehaftete Stellen, immer dann, wenn der Code auf leeren Speicherstellen (FF) landet.

Ergebnis

Ich habe das FIG-Forth und das decompilierte RSC-Forth jeweils in ein PDF gesteckt¹. Damit ist es möglich, besser Vergleiche zu ziehen. Zum Beispiel hat RSC-Forth dreimal so viel INX-Befehle und neunmal so viel DEX-Befehle wie FIG-Forth. Das ist ein klares Zeichen dafür, dass es beim RSC-Forth viel mehr Worte gibt, die Maschinenbefehle enthalten, als beim FIG-Forth, d.h., das FIG-Forth wird einfacher zu adaptieren sein, aber das RSC-Forth läuft schneller.

So, von dieser Reise in die Vergangenheit muss ich mich jetzt erst einmal erholen, und dann denke ich mal darüber nach, was ich wohl am besten als Nächstes tue. Wahrscheinlich wird es der Monitor für den MSP430 sein, denn sowas wird ja auf jeden Fall gebraucht.

Listing

```

1  ( LOADSTART )
2
3  HEX
4  : RAMTOP F000 MEMTOP ;
5  RAMTOP
6  FORGET TASK HEX E400 HERE - ALLOT 4 ALLOT : TASK ;
7
8  : ON -1 SWAP ! ;
9
10 : OFF 0 SWAP ! ;
11
12
13 HEX
14
15 : ZIFFER? ( Wert -- Flag )
16   DUP 3A < SWAP 2F > AND ;
17
18 : HEX? DUP DUP 47 < SWAP 40 > AND SWAP ZIFFER? OR ;
19
20 : ASCII>HEX 30 - DUP 9 > IF 7 - THEN ;
21
22 CODE CLI CLI, NEXT JMP, END-CODE
23 CODE SEI SEI, NEXT JMP, END-CODE
24
25 : HEX>ASCII ( HEX -- ASCII )
26   A /MOD 30 OR SWAP 30 OR 100 * + ;
27
28
29 ( R6511-DISASSEMBLER )
30
31 FORTH DEFINITIONS HEX
32
33 E000 CONSTANT (BLOCK)
34 0130 CONSTANT ADRESS.CTR
35 0132 CONSTANT QUIT.FLAG
36 0134 CONSTANT WORD.PTR
37 0136 CONSTANT DOCOL.FLAG
38
39
40 : OFFSET.DSP ADRESS.CTR @ DUP 1+ ADRESS.CTR ! DUP C@ DUP

```

¹ Ihr findet sie dort:

<http://www.forth-ev.de/filemgmt/index.php>

und dann unter: Vierte Dimension → Listings → Sonstiges



```

41      7F > IF SWAP FF - ELSE 1+ THEN + 0 D. ;
42
43 : OPCode.DSP ( Opcode -- Flag Adr.Mode )
44   ( gibt Opcode aus, uebergibt Flag fuer gueltigen Opcode )
45   ( und Adressing Mode )
46   4 * (BLOCK) + SPACE
47   DUP C@ EMIT DUP 1+ C@ EMIT DUP 2+ C@ DUP EMIT 3F = SWAP
48   3 + C@ DUP 39 > IF 7 - THEN 30 -
49   SPACE ;
50
51 : BYTE.DSP
52   ADDRESS.CTR @ DUP C@ 2 .R 1+ ADDRESS.CTR ! ;
53
54 : DBYTE.DSP
55   ADDRESS.CTR @ DUP @
56   0 <# # # # #S #> TYPE 2+ ADDRESS.CTR ! ;
57
58 ( Adressierungsarten )
59
60 : IMMEDIATE.DSP ." # " BYTE.DSP 3 SPACES ;
61
62 : ABSOLUTE.DSP DBYTE.DSP 2 SPACES ;
63
64 : ZERO-PAGE.DSP BYTE.DSP 4 SPACES ;
65
66 : ACCUM.DSP ." .A " ;
67
68 : IMPLIED.DSP 6 SPACES ;
69
70 : (IND,X).DSP ." (" BYTE.DSP ." ,X)" ;
71
72 : (IND),Y.DSP ." (" BYTE.DSP ." ),Y" ;
73
74 : Z.PAGE,X.DSP BYTE.DSP ." ,X " ;
75
76 : ABS.X.DSP DBYTE.DSP ." ,X" ;
77
78 : ABS.Y.DSP DBYTE.DSP ." ,Y" ;
79
80 : RELATIVE.DSP OFFSET.DSP SPACE ;
81
82 : INDIRECT.DSP ." (" DBYTE.DSP ." )" ;
83
84 : Z.PAGE,Y.DSP BYTE.DSP ." ,Y " ;
85
86 : BIT-ADRESSING.DSP BYTE.DSP ." ," BYTE.DSP SPACE ;
87
88 CASE: #.DSP IMMEDIATE.DSP ABSOLUTE.DSP ZERO-PAGE.DSP
89   ACCUM.DSP IMPLIED.DSP (IND,X).DSP (IND),Y.DSP
90   Z.PAGE,X.DSP ABS.X.DSP ABS.Y.DSP RELATIVE.DSP
91   INDIRECT.DSP Z.PAGE,Y.DSP BIT-ADRESSING.DSP ;
92
93 : DISASSEMBLE ( Adress -- Jmp-/Return-Flag Opcode-valid-Flag )
94   ( disassembliert einen Befehl )
95   DUP SPACE 0 <# # # # #S #> TYPE ( Adresse ausgeben )
96   DUP 1+ ADDRESS.CTR ! C@ DUP ( Jmp-/Return-Flag : ) 2DUP
97   0= SWAP 40 = OR SWAP 4C = OR SWAP 60 = OR SWAP 6C = OR SWAP
98   OPCode.DSP 0 MAX D MIN #.DSP ;
99
100 : DISASS ( Adresse -- )
101   ( disassembliert bis JMP, RTS, RTI oder zu ungueltigem Befehl )
102   ADDRESS.CTR ! CR
103   BEGIN ADDRESS.CTR @ DISASSEMBLE OR ?TERMINAL OR UNTIL ;
104
105 : DIS ( Adresse -- ) ( disassembliert bis zu ungueltigem Befehl )
106   ADDRESS.CTR ! CR

```



```

107     BEGIN ADRESS.CTR @ DISASSEMBLE SWAP DROP ?TERMINAL OR UNTIL ;
108
109 : NDIS ( n Adresse -- )
110   ( disassembliert n Opcodes bis zu ungueltigem Befehl )
111   ADRESS.CTR ! CR
112   BEGIN 1- ADRESS.CTR @ DISASSEMBLE SWAP DROP OVER 0= OR
113     ?TERMINAL OR UNTIL DROP ;
114
115 ( RSC-FORTH Decompiler )
116
117 ( CASE control statement by Charles E. Eaker )
118 ( published in FORTH Dimensions 11/3 page 37 )
119
120 FORTH DEFINITIONS DECIMAL
121
122 : CASE      ?COMP CSP @ !CSP 4 ; IMMEDIATE
123 : OF        4 ?PAIRS
124             COMPILE OVER COMPILE =
125             COMPILE OBRANCH HERE 0 ,
126             COMPILE DROP 5 ; IMMEDIATE
127 : ENDOF     5 ?PAIRS
128             COMPILE BRANCH HERE 0 ,
129             SWAP 2 [COMPILE] ENDIF 4 ; IMMEDIATE
130 : ENDCASE   4 ?PAIRS COMPILE DROP
131   BEGIN SP@ CSP @ = 0=
132     WHILE 2 [COMPILE] ENDIF REPEAT
133     CSP ! ; IMMEDIATE
134
135 ( find run-time addresses of each vocabulary word type )
136
137 ' (LOOP)      @ 2 - CONSTANT  LOOP.ADR
138 ' LIT          @ 2 - CONSTANT  LIT.ADR
139 ' :           @ 2 - @ CONSTANT  DOCOL.ADR
140 ' OBRANCH      @ 2 - CONSTANT  OBRANCH.ADR
141 ' BRANCH       @ 2 - CONSTANT  BRANCH.ADR
142 ' (+LOOP)     @ 2 - CONSTANT  PLOOP.ADR
143 ' (." )       @ 2 - CONSTANT  PDOTQ.ADR
144 -1911         CONSTANT  CONST.ADR
145 ' BASE        @ 2 - @ CONSTANT  USERV.ADR
146 -1900         CONSTANT  VAR.ADR
147 ' (;CODE)     @ 2 - CONSTANT  PSCODE.ADR
148 ' CLIT        @ 2 - CONSTANT  CLIT.ADR
149 -1595         CONSTANT  DOES.ADR
150
151 : HLESS.DSP    HEX DUP 0 D. ." Headerless "
152               CR @ DIS DECIMAL ;
153
154 : N.           ( print a number in decimal and hex )
155   DUP DECIMAL . SPACE HEX 0 ." ( " D. ." H ) " DECIMAL ;
156
157 : PDOTQ.DSP    ( display a compiled text string )
158   WORD.PTR @ 2+ DUP >R DUP
159   C@ + 1- WORD.PTR ! ( update PFA pointer )
160   R> COUNT TYPE ;
161
162 : WORD.DSP     ( given CFA, display the glossary name )
163   CONTEXT @ @
164   BEGIN SWAP DUP ROT DUP PFAPTR CFA ROT =
165     IF DUP NFA 1+ @ -11461 =
166       IF 1 QUIT.FLAG !
167       THEN ID. 1
168     ELSE PFAPTR LFA @ DUP 0=
169       IF DROP DUP -16384 U<
170         IF ." ;C" 1 QUIT.FLAG !
171         ELSE DUP HLESS.DSP
172         THEN 1

```



```

173             ELSE 0
174             THEN
175             THEN
176             UNTIL DROP ;
177
178 : BRANCH.DSP
179   ( get branch offset, calculate the actual branch address, )
180   ( and display it )
181   ." to "
182   WORD.PTR @ 2+ DUP WORD.PTR ! ( update PFA ptr )
183   DUP @ + ( offset + PFA = actual target addr )
184   0 HEX D. DECIMAL ( print it ) ;
185
186 : USERV.DSP ( display a user variable )
187   ." User variable, current value = "
188   WORD.PTR @ 2+ ( calculate PFA )
189   C@ 768 + ( +ORIGIN @ ) ( then user area adress )
190   @ N. ( fetch and print contents )
191   1 QUIT.FLAG ! ( done, set flag ) ;
192
193 : VAR.DSP ( display a variable )
194   ." Variable, current value = "
195   WORD.PTR @ 2+ ( calculate PFA )
196   @ N. ( fetch and print contents )
197   1 QUIT.FLAG ! ( done, set flag ) ;
198
199 : CONST.DSP ( display a compiled constant )
200   ." Constant, value = "
201   WORD.PTR @ 2+ ( calculate PFA )
202   @ N. ( fetch and print contents )
203   1 QUIT.FLAG ! ( done, set flag ) ;
204
205 : DOCOL.DSP ( display : or quit with ;C )
206   DOCOL.FLAG @
207   IF ." ;C" 1 QUIT.FLAG !
208   ELSE ." : " 1 DOCOL.FLAG ! THEN ;
209
210 : DECOMPILE ( PFAPTR -- )
211   BASE @ SWAP
212   ( decompiles word with given PFAPTR )
213   DUP DUP CFA @ SWAP ( contents of CFA )
214   @ = ( if contents of CFA = PFA then it is a primitive )
215   IF ." <primitive> " CR CFA @ HEX DIS DECIMAL
216   ELSE ( otherwise it's high level FORTH so decode it )
217     0 QUIT.FLAG ! 0 DOCOL.FLAG ! ( initialize done flags )
218     CFA WORD.PTR ! ( initialize pseudocode pointer )
219     CR CR ( print some blank lines )
220     BEGIN ( now list the compiled pseudocode )
221       WORD.PTR @ DUP ( fetch current pseudocode pointer )
222       0 HEX D. SPACE DECIMAL ( print value of pointer )
223       @ ( fetch current pseudocode word )
224       CASE ( now decode any special word types )
225       LIT.ADR OF ( compiled literal, print it's value )
226         WORD.PTR @ 2+ DUP WORD.PTR ! @ N. ENDOF
227       DOCOL.ADR OF ( points to the nesting routine )
228         DOCOL.DSP ENDOF
229       OBRANCH.ADR OF ( conditional branch with in-line offset )
230         ." Branch if zero " BRANCH.DSP ENDOF
231       BRANCH.ADR OF ( unconditional branch with in-line offset )
232         ." Branch " BRANCH.DSP ENDOF
233       LOOP.ADR OF ( end of a DO...LOOP structure )
234         ." Loop " BRANCH.DSP ENDOF
235       CLIT.ADR OF WORD.PTR @ DUP 2+ C@ N. 1+ WORD.PTR ! ENDOF
236       DOES.ADR OF DOCOL.FLAG @
237         IF 1 QUIT.FLAG !
238         ELSE ." DOES> - word " 1 DOCOL.FLAG ! THEN ENDOF

```

```

239 PDOTQ.ADR OF          ( display compiled text string )
240      ." Print text: " PDOTQ.DSP ENDOF
241 USERV.ADR OF          ( display a user variable )
242     USERV.DSP ENDOF
243 VAR.ADR OF            ( display a global variable )
244     VAR.DSP ENDOF
245 CONST.ADR OF         ( display a compiled constant )
246     CONST.DSP ENDOF
247 PSCODE.ADR OF        ( display ;CODE and quit )
248     WORD.PTR @ @ WORD.DSP 1 QUIT.FLAG ! ENDOF
249 DUP 63255 =
250     IF 1 QUIT.FLAG !
251     THEN              ( all special word types checked, )
252     DUP WORD.DSP      ( if word did not match any cases )
253                      ( just print it's name )
254     ENDCASE CR        ( done decoding word type )
255     2 WORD.PTR +!     ( update pseudocode pointer )
256     QUIT.FLAG @      ( check if finished flag set or if )
257     ?TERMINAL OR     ( interruption from terminal )
258     UNTIL             ( otherwise display another word )
259     ENDIF CR         ( all done now )
260     BASE ! ;
261
262 : DLI -FIND 0=        ( is input word in dictionary ? )
263     IF 3 SPACES ." ? not in glossary " CR ( no, quit )
264     ELSE DROP DECOMPILE THEN ;
265
266 : VLI
267     CONTEXT @ @
268     BEGIN DUP ID. 2 SPACES PFAPTR LFA @ DUP 0= ?TERMINAL OR
269     UNTIL DROP ;
270
271 : CFIND CONTEXT @ @
272     BEGIN SWAP DUP ROT
273     DUP
274     PFAPTR CFA ROT =
275     IF CR ID. 1
276     ELSE PFAPTR LFA @ DUP 0=
277     IF DROP ." not found " 1
278     ELSE 0
279     THEN
280     THEN ?TERMINAL OR UNTIL DROP ;
281
282 : ADUMP 0 DO DUP I +
283     C@ DUP IF > IF 2 SPACES EMIT ELSE 3 .R THEN LOOP DROP ;
284
285 : DECOMPLIST ( name -- ) CR CR
286     -FIND 0= IF 3 SPACES ." not in glossary " CR
287     ELSE DROP NFA
288     BEGIN DUP CR ID. PFAPTR DUP DECOMPILE CR
289     LFA @ DUP 0= ?TERMINAL OR UNTIL DROP THEN ;
290
291
292
293
294

```



Bootmanager und FAT-Reparatur: Zehnter Fort(h)schritt (putdiskimage)

Fred Behringer

Der vorliegende Teil 10 meiner Artikelserie kann als Fortsetzung von Teil 8 betrachtet werden. In Teil 8 wurde gezeigt, wie man bei einem PC (bei mir konkret mit AMD-K6-2 als CPU) eine 3.5-HD-Diskette als Hex-Abbild ‚an einem Stück‘ im RAM unterbringen kann (**getdiskimage**). Hier nun folgt ein Programm zum Zurückschreiben auf die (oder auf eine andere) Diskette (**putdiskimage**). Das Ganze spielt sich im Real-Mode ab, als Zusammenspiel von BIOS, Assembler und Forth. Es wird kein Int15h, kein DPMI, kein RAM-Management und insbesondere kein Abstecher in den Protected-Mode benötigt. Zentrales Mittel ist der (erweiterte) BIOS-Interrupt 13h (hier in der Schreibversion als 13h/3). Es wird auch kein 32-Bit-Forth-Assembler gebraucht. Der im 16-Bit-Turbo-Forth enthaltene 16-Bit-Assembler genügt — wenn er mit einigen wenigen Ergänzungen versehen wird. Wichtig für die Analyse ist der mit viel Überlegung entwickelte 32-Bit-Dump-Apparat (**dump-32**). Selbstverständlich ist vorgesehen, das Diskimage im RAM als Arbeitsfeld für Disk-Modifikationen, wie z.B. die Aufgaben aus Teil 4, zu verwenden. Die Automatisierung der dazu nötigen Parameter-Eingaben (Spielwiese für High-Level-Forth) soll für später zurückgestellt bleiben. Ich habe einige Dinge aus Teil 8 hier noch einmal aufgeführt. Für das Zurückschreiben des Disketten-Images auf Diskette kommt man damit mit der Eingabe

```
forth include bootma10 [ret]
```

durch, wobei forth.com das 16-Bit-Turbo-Forth-System ist und bootma10.fth das Programm aus dem Listing des vorliegenden Artikels. Will man ein vollständiges Programm (zum Lesen, Verarbeiten und Schreiben) haben, dann gehe man so vor, wie im Abschnitt ‚Schnelleinstieg‘ beschrieben.

Bei den hier entwickelten Mitteln zeigt sich die Stärke von Forth, wo man ganz schnell Dinge gestalten kann, die (im vorliegenden Beispielfall) von Diskcopy (aus dem altherwürdigen DOS) oder Rawwrite (z.B. [JN00]) nicht immer erledigt werden können.

Voraussetzungen

Es mögen die in Teil 8 gemachten Voraussetzungen gelten. Aber auch nur diese. Weitere werden hier nicht benötigt. Am leichtesten tut man sich, wenn man das Listing aus Teil 8 mit dem Listing des vorliegenden Teils 10 zusammenführt und als einheitliches Ganzes betrachtet. Redundante Definitionen können dabei der Einfachheit halber beibehalten werden. Sie stören nicht.

Ich gehe davon aus, dass ab der in der 2Variablen **imgbuf** aufbewahrten RAM-Adresse der von

(**getdiskimage**) sektorweise hexadezimal abgelegte Inhalt einer 3.5-HD-Diskette liegt, die den in Teil 8 [FB1a] gemachten Voraussetzungen genügt. Natürlich — und das ist im vorliegenden Teil volle Absicht — kann das ein ‚intelligent‘ abgeänderter Disketten-Image-Inhalt sein! An sich ist es völlig egal, wo das Disketten-Image für das hier zu entwickelnde putdiskimage herkommt. Es muss halt so aufgebaut sein, dass es beim Zurückschreiben eine sinnvolle Diskette ergibt. Die Übertragung des Disketten-Images aus dem RAM in die ‚reale‘ Diskette ist der eigentliche Gegenstand des diesmaligen Teils 10 meiner Artikelserie. **imgbuf** soll als Punktzahl (32 Bit) eingegeben worden sein! Der Wert 2000000. (Punktzahl!) wäre ein guter Wert für Eigenexperimente.

Turbo-Forth unterscheidet nicht zwischen Groß- und Kleinschreibung. Ich bevorzuge im Vorliegenden die Kleinschreibung.

Sämtlichen Programmteilen liegt die Zahlenbasis hex zugrunde.

Schnelleinstieg

Es sei **forth.com** das 16-Bit-Turbo-Forth-System. Es sei **bootma-8.fth** das Forth-Programm aus Teil 8 der Artikelserie.

Es sei **bootma10.fth** das Forth-Programm des vorliegenden Teils 10.

Man gebe ein:

```
forth include bootma-8 include bootma10 [ret] .
```

Damit steht dann alles zur Verfügung, um

- (1) von einer 3.5-HD-Diskette in Laufwerk a: ein Image ins RAM zu schreiben,
- (2) das Image zu analysieren und gegebenenfalls zu verändern,
- (3) das Image dann auf (dieselbe oder eine andere) Diskette zu schreiben.

Man erreicht das (beispielsweise) durch Eingabe von:

- (1) 2000000. **getdiskimage**
Wert (Punktzahl!) weitgehend beliebig wählbar,
- (2) 2000000. 100. **dump-32**
Anfangs-Screen (auch 100. ist Punktzahl!).
- (3) 2000000. **putdiskimage**
Zurück. Eventuelle Fehlermeldungen beachten.

Achtung

Das Forthwort **dump-32** bezieht sich (zumindest in der Voreinstellung) auf **fs = 0**. Wenn man beim Experimentieren beispielsweise per

```
4711 2000000. cc!-32 2000000. cc@-32
```

etwas Falsches herausbekommt, hatte sich das Segment fs ursprünglich wahrscheinlich auf einen anderen Wert als 0 eingestellt. (Bei mir stellt sich fs gleich nach Beginn auf fs = f000 ein.) Abhilfe schafft die Eingabe 0 fs! vor der eben vorgeschlagenen Überprüfung mit 4711.

Schnelle Plausibilitätsüberprüfung

Um sicherzugehen, dass bei den Eingaben 1 bis 3 ‚alles mit rechten Dingen zugegangen ist,‘

gebe man ein: 100000. 100. dump-32

Bei mir begann dann die erste Zeile des Dumpings rechts mit der Eingangs-Kennung ... VDISK3.3... für die RAM-Disk.

Erklärung: Die zweifach includete Programmeingabe des vorausgegangenen Abschnitts stellt das System auf hex ein (und dabei soll es bleiben!). Die RAM-Adresse 100000. (der Punkt gehört zur Zahl) enthält das erste Byte des ‚Extended Memorys.‘ Dorthin legt das DOS-System bei mir normalerweise eine RAM-Disk (von 30 MB Länge). Eine Eintragung in der config.sys sorgt dafür. Wenn ich das Disketten-Image nach 2000000. lege, wird die RAM-Disk, wie man sich leicht ausrechnet, von dem Disketten-Image, welches aus getdiskimage hervorging, nicht angetastet. Ich kann und konnte also während meiner Experimente die RAM-Disk voll ausnutzen und alle benötigten Dateien in der RAM-Disk vorhalten.

Die Punkte 1 bis 3 auch auseinanderziehbar

Man gebe ein:

```
forth include bootma-8 [ret] .
```

Dann:

(1) 2000000. getdiskimage

(ohne die Nummerierung und mit Abschluss [ret].)

Dann kann man sich das Disketten-Image ansehen, es überprüfen und es gegebenenfalls auch abändern per:

(2) 2000000. 100. dump-32

und Weiterschalten über die Plustaste, wobei die Änderungen per c!-32 etc. vorgenommen werden können. Dann kann man aus dem ganzen Forth-System auch ruhig wieder herausgehen:

(2a) bye

Das Disketten-Image bleibt (zumindest bei dem gewählten höheren Adresswert) erhalten, solange man das DOS-System nicht per Affengriff (oder sogar per Netzschalter) ausschaltet. Dann kann man eingeben:

```
forth include bootma10 [ret] .
```

Nun ist alles fürs Schreiben auf Diskette vorbereitet. Man lege, so man will, eine neue Diskette ins Laufwerk und leite das Schreiben ein durch:

(3) 2000000. putdiskimage

Falsifizierung

Nicht jedem genügt die eben angeführte Plausibilitätsbetrachtung. Gerade bei solchen Übernahmen von ‚Fremdprogrammen‘ wie dem im vorliegenden Artikel kann sich leicht ein Abschreibefehler einschleichen. Eine genauere Überprüfung wäre besser. Natürlich kann man mit noch

so genauen Methoden nicht nachprüfen, ob das vorgeschlagene Verfahren immer zum Ziel führt. Man kann aber das Vertrauen in das generelle Funktionieren des (abgetippten oder sonstwie beschafften) Programms erhöhen, beispielsweise dadurch, dass man die ‚geklonte‘ Diskette wieder einliest, diesmal aber beispielsweise an die Adresse 3000000. (andere Werte tun es natürlich auch):

(1a) 3000000. getdiskimage

Und dann kann man die Bytes ab 2000000. mit denen ab 3000000. vergleichen. Im Listing habe ich dafür das Forth-Wort falsify vorgesehen.

Das Wort falsify ist nur einer von vielen denkbaren Falsifikatoren, aber sicher ein plausibler: Das Image der Ursprungsdiskette wird Byte für Byte mit dem Image der geklonten Diskette verglichen. Bei den Bytes aus dem ersten Image, die sich vom korrespondierenden Byte aus dem zweiten Image unterscheiden, wird ein Aufaddierer (im Listing heißt er falsi) inkrementiert. Liefert falsi nach Beenden des Vorgangs den Wert 0, dann ist das Vertrauen in das richtige Funktionieren (des Klon-Programms) groß. Ganz sicher kann man natürlich nicht sein, denn es könnten ja beispielsweise sowohl im Lese- wie auch im Schreib-Programm die CHS-Reihenfolgen zu CSH vertauscht worden sein (siehe Bemerkungen in Teil 8), was auf den Wert von falsi keinen Einfluss hätte. Sicher ist nur, dass im Gegenbeispielsfall (wenn falsi einen Wert ungleich null liefert) das Klonen nicht erfolgreich war. Außerdem könnte sich ja auch in den Falsifikator falsify ein Fehler (beim Abtippen oder schon beim Entwerfen) eingeschlichen haben. Und das würde das Thema Falsifizierung weiterverkomplizieren: Kann ein Falsifikator (als Aussage gefasst) in die zu falsifizierende Aussage einbezogen werden, ohne dass das Ganze, ähnlich der Menge aller Mengen, die sich selbst nicht enthalten, zu einem Widerspruch in sich führt.

Liegt ein Fehler vor, dann stellt der von falsi gelieferte Wert ein Maß dafür dar, wie ‚stark‘ die geklonte Diskette von der Ursprungsdiskette abweicht. (Diesen Gedanken könnte man natürlich ausbauen: ‚Forth und die Philosophie.‘)

Über ‚Falsifizierung‘ (im Popperschen Sinne) kann man sich per ‚Google — Wikipedia — Falsifizierung‘ Informationen verschaffen.

Falsifikator für was?

Ich versuche es einmal mit folgender Formulierung:

Das Forth-Wort falsify (siehe Listing) ist ein Falsifikator für die Vermutung „Die durch ad1 und ad2 charakterisierten Disketten-Images sind zwei Images ein und derselben Diskette.“

1. Sind die Strings ad1 und ad2 byteweise gleich, dann kann man den Satz vom unzureichenden Grund aus der Entscheidungstheorie anwenden und sagen, es liegt kein Grund vor, an der eben aufgestellten Vermutung zu zweifeln. Sicher sein kann man natürlich nicht: Eine beliebige gleichlaufende Umordnung beider Strings würde ja zum gleichen Ergebnis führen,



und darunter wären auf jeden Fall auch solche, die gar keine Images von Disketten sein können. Beispiel: ‚Null-Images.‘

2. Sicher ist aber eines (Falsifizierungsfall): Ist auch nur ein einziges Paar von Bytes aus den beiden Disk-Images im RAM voneinander verschieden, dann ist unsere Vermutung zu Fall gebracht. Dann kann es sich nicht um zwei Images ein und derselben Diskette handeln.
3. Das Ganze ist natürlich stark abhängig von einer sauberen Definition dessen, was wir unter einem Disketten-Image verstehen wollen.

Kaputte Diskette wegwerfen?

Angenommen, das eine Image stammt von einer als einwandfrei erkannten ‚Original-Diskette.‘ Weiterhin angenommen, wir haben damit per `putdiskimage` eine Kopie-Diskette erstellt. Und schließlich angenommen, wir haben von dieser Kopie-Diskette aus ‚Sicherheitsgründen‘ ebenfalls ein Image (‚zweites Image‘) per `getdiskimage` angefertigt. Der Bytepaar-Vergleich läuft in meinem Programm (kann natürlich auch anders eingerichtet werden) von höheren zu niedrigeren Adressen. Es bot sich an, die Adresse des zuletzt ermittelten Unstimmigkeits-Paars (falls es überhaupt ein solches gibt) in der Bildschirm-Statistik festzuhalten. Das geschieht denn auch (siehe Listing). Damit haben wir ein schnelles Mittel an der Hand, herauszubekommen, ab welchem Byte auf die Image-Werte kein Verlass mehr ist.

Einbettung des Disk-Images in eine Datei im RAM

Inzwischen habe ich mich über Teil 8 hinaus mit dem ‚Handling‘ des Images der Diskette (im RAM, zur Aufbewahrung oder/und zur Weitergabe) beschäftigt und in Teil 9 der Artikelserie [FB1b] Vorschläge unterbreitet.

Programm-Aufbau

Dem in Teil 8 Gesagten lag der Leseteil des Interrupts 13h, also dessen Funktion 2, zugrunde. Hier in Teil 10 ist es der Schreibteil, also die Funktion 3 von Int 13h. Entsprechend stark ausgeprägt ist die Analogie des vorliegenden Programms `putdiskimage` zu `getdiskimage` aus Teil 8.

Hauptlast im vorliegenden Artikel

trägt das Zubringer-Wort (`putdiskimage`), die Klammern gehören zum Forth-Wort, das vollständig in Forth-Assembler geschrieben ist. Das High-Level-Wort `putdiskimage` ruft (`putdiskimage`) auf und nutzt die im Forth-System schon vorhandene Organisationsgewalt aus. Es läuft so ziemlich alles im vorliegenden Teil 10 analog zu dem in Teil 8 Gesagten. Zur Erinnerung darf ich ein paar Erklärungen in Form von Beispielen (mit konkreten Eingabewerten) aufziehen.

2000000. `getdiskimage`

legt ein Disketten-Abbild nach Adresse 2000000. — der Punkt gehört zum Forth-Wort. Das zugrunde gelegte Turbo-Forth ist ein 16-Bit-System. Eine 3.5-HD-Diskette fasst 1.44 Megabyte. Mit (einfachgenauen) 16-Bit-Zahlen für die Adressen komme ich also (auch schon wegen des begrenzten PC-RAM-Bereichs) nicht durch. Punktzahlen, wie die eben genannte, sind doppeltgenau (32 Bit breit). Über mögliche RAM-Organisationen (wie beim Aufbau von Dateisystemen) habe ich hier (immer noch) nicht weiter nachgedacht. Ich bewege mich mit Turbo-Forth in einem DOS-System und agiere nach dem Motto: Mein RAM gehört mir und ich kann damit machen, was ich will. (Übrigens befolgt ja DOS ein solches Motto mit `.com`-Dateien, die bei Aufruf (zunächst einmal) den gesamten PC-RAM-Speicher zur Verfügung gestellt bekommen, auch.) 2000000. ist ein leicht merkbarer ‚runder‘ Wert. Ich habe mit Erfolg auch mit anderen Werten experimentiert. (Der Eingabe-Parameter ist eine Punktzahl, doppeltgenau, also 32 Bit breit. Den Punkt nicht vergessen!)

High-Low-Vertauschung in Forth

Man beachte die bei doppeltgenauen Zahlen (Punktzahlen) üblicherweise vorzufindende Vertauschung von höherwertigem und niederwertigem Anteil! Formt man aus einer Turbo-Forth-Punktzahl (z.B. aus dem Inhalt von `imgbuf`) dann eine 32-Bit-Assembler-(Ganz-)Zahl (wie beispielsweise zum Einlagern in das CPU-Register `eax`), dann muss aus der Punktzahl erst eine Little-Endian-Zahl gemacht werden!

Nachträgliche Veränderungen am Disk-Image

lassen sich mit den über `dump-32` gewonnenen Adress-Informationen per `!-32`, `@-32` etc. gezielt vornehmen. Wegen näherer Erklärungen vergleiche man das in Teil 8 Gesagte. Will man solche Veränderungen per `dump-32` überprüfen, dann achte man darauf, vorher (oder zumindest zwischendurch) `fs = 0` zu setzen (zu erreichen über `0 fs!`), da `dump-32` (allerdings nur im Default-Fall) mit einem nullgesetzten Segment `fs` arbeitet. (Im Real-Mode berücksichtigt das System die Segment-Register `fs` und `gs` nur 16 Bit breit.)

Ich verwende eine 16/32-Bit-Mischung

Will man beispielsweise den Disk-Spuren-Puffer `trackbuf` komfortabel über `dump-32` auslesen (statt einfach über das in Turbo-Forth eingebaute `dump`), dann lese man sich vorher den Abschnitt mit der gleichen Überschrift in Teil 8 und die daran anknüpfenden Abschnitte aufmerksam durch!

Liefert `putdiskimage` des vorliegenden Teils 10 einen Schreibfehler,

dann wird der Schreibversuch für die gerade anstehende Disketten-Spur wiederholt. Nach drei hintereinander aufgetretenen Schreibfehlern wird das Programm abgebrochen. Am Bildschirm wird beim Aussprung dann die

erreichte Seite, die erreichte Spur, die Adresse des zuletzt übertragenen Bytes und die Angabe, ob ein Fehler vorliegt oder nicht, angezeigt.

Abschließendes Experiment mit Nulldiskette

Ich habe eine voll beschriebene DOS-Diskette ins Laufwerk a: gelegt und mein Gesamtsystem aufgerufen:

- (1) `forth include bootma-8 include bootma10 [ret]`
- (2) `2000000. getdiskimage [ret]` → Kein Lesefehler
- (3) `2000000. 3000000. falsify [ret]` → 151b8e unstimmmige Bytepaare
- (4) `3000000. putdiskimage [ret]` → Kein Schreibfehler
- (5) `2000000. getdiskimage [ret]` → Kein Lesefehler
- (6) `2000000. 3000000, falsify [ret]` → Kein unstimmmiges Bytepaar

Interpretation

Alle Bytes, insbesondere auch die ab 2000000. bis 2161f00. und die ab 3000000. bis 3161f00. waren anfangs null. In Punkt 2 wurde ein Disk-Image nach 2000000. gelegt. Das Falsifizieren in Punkt 3 lieferte nicht die Höchstzahl an unstimmmigen Bytepaaren (161f00): Die Diskette enthielt ja auch Nullbytes — und die wurden mit 0 verglichen, was zu stimmigen Bytepaaren führte. In Punkt 4 wurde eine ‚Nulldiskette‘ hergestellt (alle Bytes, aber auch wirklich alle, gleich 0)! Davon wurde in Punkt 5 ein Disk-Image angefertigt und nach 2000000. gelegt. Und das Falsifizieren in Punkt 6 ließ erkennen, dass die Diskette jetzt tatsächlich zur Nulldiskette geworden war.

Listing

```

1  \ *****
2  \ *
3  \ * BOOTMA10.FTH
4  \ *
5  \ * Zutaten fuer FAT-Reparatur und Bootmanager unter *
6  \ * Turbo-FORTH-83
7  \ *
8  \ * Fred Behringer - Forth-Gesellschaft - 30.8.2011 *
9  \ *
10 \ *****
11
12 \ =====
13 \ Auch im vorliegenden Artikel wird wieder darauf ver-
14 \ zichtet, die Programme fuer das ZF-System ebenfalls
15 \ einzurichten. Sie sind nur fuer Turbo-Forth gedacht!
16 \ Neu ist putdiskimage mit Zubringer (putdiskimage).
17 \ Die aus Teil 8 benoetigten Worte werden wiederholt.
18 \ =====
19
20
21 hex \ Alle Eingaben im vorliegenden Listing sind hexadezimal !
22
23 \ 32-Bit-Assembler-Erweiterung
24 \ -----
25 \ Es wurde alles mit einem AMD-K6-2/500 ausprobiert (vergleiche Teil 8).
```

Diskette schreddern

Der ursprüngliche Disketten-Inhalt, oh Schreck, war natürlich weg ;-) War aber nicht weiter schlimm, da das ja sowieso eine Experimentier-Diskette war. Ich hätte in Punkt 5 und 6 mit z.B. 4000000. statt mit 2000000. arbeiten können — und das ursprüngliche Image aus 2000000. anschließend zurückschreiben. Jedenfalls habe ich auf diese Weise gezeigt, wie man eine Diskette inhaltlich ganz schnell und ganz leicht schreddern kann. Forth macht's (auch mir als Amateur) möglich! Dazu brauche ich keine Spezialprogramme aus dem Internet.

Turbo-Forth, und kein ZF?

Auch im vorliegenden Artikel gilt das in Teil 8 im Abschnitt mit der gleichen Überschrift Gesagte. Das Thema ‚Anpassung an ZF‘ verschiebe ich auf später.

Literatur

- [FB98] Behringer, Fred: Real-Mode-32-Bit-Erweiterung für Turbo-Forth. Vierte Dimension 2/1998.
- [FB08] Behringer, Fred: Erster Teil meiner VD-Artikel-Serie. Vierte Dimension 3/2008.
- [FB09] Behringer, Fred: Vierter Teil meiner VD-Artikel-Serie. Vierte Dimension 2/2009.
- [FB1a] Behringer, Fred: Achter Teil meiner VD-Artikel-Serie. Vierte Dimension 1/2011.
- [FB1b] Behringer, Fred: Neunter Teil meiner VD-Artikel-Serie. Vierte Dimension 2/2011.
- [JN00] Newbigin, John: <http://www.chrysocome.net/rawwrite> ... (2000).



```
26
27 only forth also assembler definitions
28
29 : fs:      64 c, ; : gs:      65 c, ; : opsize: 66 c, ; : adrsiz: 67 c, ;
30 : eax opsize: ax ; : ecx opsize: cx ; : edx opsize: dx ; : ebx opsize: bx ;
31 : esp opsize: sp ; : ebp opsize: bp ; : esi opsize: si ; : edi opsize: di ;
32
33
34 \ Freischalten der Adressleitung a20
35 \ -----
36 forth definitions
37
38 \ himem.sys in der config.sys reicht an sich. Ansonsten free-a20 verwenden:
39 code free-a20 ( -- )               cli
40     begin 64 # al in 02 # al and 0= until 0d1 # al mov 64 # al out
41     begin 64 # al in 02 # al and 0= until 0df # al mov 60 # al out
42     begin 64 # al in 02 # al and 0= until sti next end-code
43
44 : a20? ( -- fl ) 0ffff 10 l@ 0. l@ <> 0= ; \ fl=-1 --> a20 gesperrt
45
46
47 \ Segment-Register fs und gs (im PC ab 80486 vorhanden) bearbeiten
48 \ -----
49
50 \ cs ds es benoetigen im vorliegenden Artikel keine Sonderbehandlung.
51 code fs@ ( -- cc ) 0f c, 0a0 c, next end-code \ f-Segment holen: fs push
52 code gs@ ( -- cc ) 0f c, 0a8 c, next end-code \ g-Segment holen: gs push
53 code fs! ( cc -- ) 0f c, 0a1 c, next end-code \ f-Segment setzen: fs pop
54 code gs! ( cc -- ) 0f c, 0a9 c, next end-code \ g-Segment setzen: gs pop
55
56
57 \ Datenverkehr ueber den gesamten 32-Bit-Systembereich
58 \ -----
59 \ Die folgenden Store- und Fetch-Befehle laufen ueber fs. Lineare Adressen
60 \ per 0 fs! - In dump-32 ist fs=0 Default. fs wird dabei 'gerettet'. Beachte
61 \ das in Teil 8 ueber Stack-Kommentare bei Punktzahl-Eingaben Gesagte.
62
63 code c@-32 ( ad. -- c ) \ Unmittelbares Byte c von Adresse ad. zum Stack
64     ebx pop opsize: 0c1 c, 0c3 c, 10 c, ( 10 # ebx rol ) ax ax xor
65     fs: adrsiz: 8a c, 03 c, ( fs:[ebx] al mov ) 1push end-code
66
67 code c!-32 ( c ad. -- ) \ Byte c vom Stack nach Adresse ad. speichern
68     ebx pop opsize: 0c1 c, 0c3 c, 10 c, ( 10 # ebx rol ) ax pop
69     fs: adrsiz: 88 c, 03 c, ( al fs:[ebx] mov ) next end-code
70
71 code cc@-32 ( ad. -- cc ) \ Doppelbyte cc von Adresse ad. zum Stack
72     ebx pop opsize: 0c1 c, 0c3 c, 10 c, ( 10 # ebx rol )
73     fs: adrsiz: 8b c, 03 c, ( fs:[ebx] ax mov ) 1push end-code
74
75 code cc!-32 ( cc ad. -- ) \ Doppelbyte cc nach Adresse ad. speichern
76     ebx pop opsize: 0c1 c, 0c3 c, 10 c, ( 10 # ebx rol ) ax pop
77     fs: adrsiz: 89 c, 03 c, ( ax fs:[ebx] mov ) next end-code
78
79 code @-32 ( ad. -- d ) \ 32-Bit-Wert d von Adresse ad. zum Stack (32-Bit)
80     ebx pop opsize: 0c1 c, 0c3 c, 10 c, ( 10 # ebx rol )
81     opsize: fs: adrsiz: 8b c, 03 c, ( fs:[ebx] eax mov )
82     opsize: 0c1 c, 0c0 c, 10 c, ( 10 # eax rol ) eax push
83     next end-code
84 code !-32 ( d ad. -- ) \ 32-Bit-Wert d in 4 Bytes ab Adresse ad. legen
85     ebx pop opsize: 0c1 c, 0c3 c, 10 c, ( 10 # ebx rol )
86     eax pop opsize: 0c1 c, 0c0 c, 10 c, ( 10 # eax rol )
87     opsize: fs: adrsiz: 89 c, 03 c, ( eax fs:[ebx] mov )
88     next end-code
89
90 \ Absicherung gegen RAM-Bereichs-Ueberschreitung
91 \ -----
```



```

92 \ Man achte auf fs (lineare Adressen erfordern 0 fs!). Dieser RAM-Test
93 \ schleift beliebige Eingaben fuer ad. durch. Im weiter unten stehenden
94 \ Forth-Wort putdiskimage wird vor der Fehlermeldung erst noch 161f00
95 \ hinzuaddiert, um sicherzustellen, dass es sich wirklich um einen als
96 \ Disk-Image plausiblen RAM-Bereich handelt.
97
98 : ramtest ( ad. -- ad./abort )      \ ad. = Punktzahl (doppeltgenau)
99   fs@ -rot                          \ fs   aufbewahren
100   2dup @-32 2swap                    \ (ad.) aufbewahren
101   5a5a5a5a. 2over !-32 2dup @-32 5a5a5a5a. d= -rot
102   a5a5a5a5. 2over !-32 2dup @-32 a5a5a5a5. d= -rot
103   2swap and not >r
104   2swap 2over !-32
105   rot fs!
106   r> cr abort" Bei dieser 32-Bit-Eingabe wird RAMmax ueberschritten!" ;
107
108
109 \ 32-Bit-RAM dumpen (dump-32). Ausfuehrlicherer Kommentar in Teil 8.
110 \ -----
111 code 2tuck ( d1 d2 -- d2 d1 d2 )    \ Zur Vereinfachung von dump-32
112   eax pop ecx pop eax push ecx push eax push next end-code
113 : d0<= ( d -- fl ) 0. d> not ;      \ Zur Vereinfachung von dump-32
114
115 variable ascfilt  ascfilt on        \ Nur ASCII-Zeichen bei dump ? on = ja
116 variable fs-dump-32 0 fs-dump-32 !  \ fs fuer dump-32. Vorgabe 0 ('linear').
117
118 : ?ascii ( n1 -- n2 )                \ Nicht-ASCII-Zeichen --> Punkt ?
119   ascfilt @                          \ true = ja
120   if
121     dup 20 7e between not            \ nicht ASCII ?
122     if drop 2e then                  \ dann Punkt
123   else
124     dup 7 =                          \ Bell
125     over 8 = or                      \ Del
126     over a = or                      \ Linefeed
127     over d = or                      \ cr
128     over 1b = or                     \ esc
129     over Off = or                    \ Bell usw. ?
130     if drop bl then                 \ dann Blank
131   then ;
132
133 : dump^ ( -- )
134   ." 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF" ;
135
136 : dump-32 ( ad. len. -- )           \ Stop: SPACE. Weiter: SPACE. Exit: CR CR
137   cr >r >r >r >r
138   fs@ fs-dump-32 @ fs!              \ Mehr: + . Nochmal zurueck: -
139   base @ hex r> r> r> r> 2over      \ Anzeige in Hex
140   10 mu/mod 2drop >r 2swap r@ 0 d- r>
141   dup 3 * dup 17 > if 1+ then 0a + (cursor) nip 0b swap at dump^
142   (cursor) nip at bold ." \/ " 3c + (cursor) nip at ." V" attoff cr
143   begin
144     begin
145       2dup <# # # # # # # #> type space space \ Adresse anzeigen
146       8 0 do 2dup c@-32 0 <# # # #> type space 1. d+ loop space
147       8 0 do 2dup c@-32 0 <# # # #> type space 1. d+ loop space 10. d-
148       10 0 do 2dup c@-32 ?ascii emit 1. d+ loop cr
149       2swap 10. d- 2tuck
150       d0<= stop? or \ Ende oder Abbruch ?
151     until
152     (cursor) dup 18 = if 0 0 at 0b spaces dump^ then at
153     bold (cursor) -1 0 (frame) ." Weiter: + Zurueck: - Exit: RET"
154     attoff at
155     key
156     dup ascii + = if drop 2swap 2drop 100. 2swap
157     false else

```



```
158         ascii - = if 2swap 2drop 100. 2swap 200. d- dark 0b 0 at dump~ cr
159                 false else
160         2drop 2drop base ! fs! true then then
161         until 0c 0d (frame) cr ;
162
163
164 \ Es folgt der 18d-Sektor-Puffer trackbuf (vergleiche Teil 8 der Serie).
165 \ -----
166
167 2410 allot here          \ Platz fuer mind. 18d Sektoren = 512d * 18d Bytes
168 here 0f and -           \ trackbuf an Paragraphen-anfang
169 2400 -                  \ Anfang des Uebertragungs-Puffers
170 constant trackbuf       \ Liefert Anfangsadresse des Puffers fuer 1 Spur.
171
172 2variable imgbuf        \ Anfangsadresse des Disk-Image-Puffers (Punktzahl).
173 2variable imgbyte       \ Adresse 1. freies Byte nach dem letzten Doppelbyte.
174
175 variable spur           \ Letzte geschriebene Spur
176 variable seite          \ Letzte geschriebene Seite
177 variable flag           \ Fehler-Flag (0/1) fuer (putdiskimage) : 1 = Fehler
178
179 \ Die bis hierher aufgefuehrten Worte sind auch in Teil 8 enthalten. Mehr
180 \ Kommentare siehe dort. Das Listing aus Teil 8 kann auch mit dem hiesigen
181 \ Listing von hier ab zusammengefuehrt werden, ohne etwas zu verlieren.
182
183 \ Das folgende Wort (putdiskimage) ist die Code-Vorstufe zu putdiskimage.
184 \ Es ist analog zu (getdiskimage) aufgebaut, nur dass jetzt 'int 13h/3'
185 \ (Sektoren schreiben) statt 'int 13h/2' (Sektoren lesen) angesprochen wird.
186 \ Ausserdem kehrt sich natuerlich jetzt der Datentransfer um, also Spur fuer
187 \ Spur von imgbuf ueber trackbuf (je 1 Spur) per INT 13h/3 zur Diskette.
188
189 code (putdiskimage) ( -- )
190     ds push             \ ds (push = mit Stack als Zwischenstation)
191     es pop              \ nach es uebertragen.
192     0 # flag #) cmp 0=
193     if                 \ di frei verwendbar.
194         trackbuf        \ Anfang des Puffers fuer Transfer der naechsten Spur
195         # di mov        \ nach di legen.
196         imgbyte #) push \ Hi-Anteil der 2Variablen imgbyte auf den Stack legen.
197         imgbyte 2 + #) push \ Lo-Anteil der 2Variablen imgbyte auf den Stack legen.
198         ebx pop         \ Disk-Image-Zeiger liegt jetzt little-endian in ebx.
199         clc             \ -----
200         1200 do          \ imgbuf>trackbuf: 1200h = 4608d Doppelbytes = 9216 B
201         \ -----
202         \ cx = Zaehler fuer do-loop. cl wird dann neu gesetzt.
203         adrsiz: fs:      \ Auf Segment fs bezogen und mit 32-Bit-Adressbreite.
204         8b c, 03 c,      \ Das entspricht [ebx] ax mov (beachte adrsiz).
205         ax 0 [di] mov    \ di = trackbuf (16 Bit). Keine edi-Erweiterung noetig.
206         \ -----
207         di inc          \ ax uebertraegt ein Doppelbyte. Die Adresse di muss
208         di inc          \ sich also anschliessend um 2 erhoehen: di = di+2.
209         ebx inc         \ ebx muss 1,44 MB ansprechen. Die 16 Bit fuer bx
210         ebx inc         \ reichen nicht. ax fasst 2 Bytes. Also ebx = ebx+2.
211         loop            \ -----
212         ebx push        \ Adresse des ersten freien Bytes nach dem zuletzt
213         imgbyte 2 + #) pop \ gesendeten Doppelbyte. Lo-Anteil davon forthrichtig
214         imgbyte #) pop   \ in 2Variable imgbyte legen, Hi-Anteil entsprechend.
215         then            \ -----
216         3 # di mov      \ Zaehler fuer 3 Disketten-Schreibversuche ansetzen.
217         begin          \ ----- Bis zu 3 Schreibversuche -----
218         0 # flag #) cmp 0< \ ----- Laufwerk initiieren ? -----
219         if
220             dl dl xor    \ dl = 0 = Diskettenlaufwerk a:
221             ah ah xor    \ Funktion 0 von Int 13h aufrufen, also
222             13 int       \ das Disketten-System zuruecksetzen.
223             then        \ ----- Bemerkung hierzu siehe unten -----
```

```

224      \ ----- 1 Spur (18d Sektoren) schreiben -----
225  seite #) dh mov      \ Seitennummer dieses Spurendurchgangs
226  spur  #) ch mov      \ Spurenummer dieses Spurendurchgangs
227      trackbuf        \ 18d Sektoren (= 1 Spur = 9216d = 2400h B) schreiben
228      # bx mov         \ bx = Anfangs-Offset des Spur-Transfer-Puffers
229      1 # cl mov       \ cl = Sektor 1 in der laufenden Spur (18d Sektoren)
230      dl dl xor        \ dl = 0 = Diskettenlaufwerk a:
231      3 # ah mov       \ ah = 3 (Schreibfunktion von Int 13h)
232      12 # al mov      \ al = 12h = 18d Sektoren = 1 Spur auf ch/dh schreiben
233      13 int          \ Schreib-Interrupt aufrufen.
234      u>= if          \ Wenn danach cf = 0 (kein Lesefehler), dann Aussprung
235      di di xor        \ aus der Schleife per di = 0 vorbereiten und
236  0 # flag #) mov     \ flag = 0 (kein Fehler) setzen.
237      else           \ Wenn jedoch cf <> 0 (Lesefehler),
238      di dec          \ dann Zaehler fuer 3 Schreibversuche dekrementieren
239  1 # flag #) mov     \ und flag = 1 (Fehler) setzen.
240      then
241      0 # di cmp 0=    \ Aussprung aus der 3er-Schleife mit flag = 0 oder 1.
242      until          \ -----
243  next end-code
244      \ -- Bemerkung zu int 13h/0 (Laufwerk initiieren) --
245      \ Das Disketten-Laufwerk braucht nicht fuer jede Spur
246      \ neu initiiert zu werden (vergl. Bemerkung in Teil 8).
247
248  : printparams ( -- ) \ Ausdruck von flag-Meldung imgbuf imgbyte seite spur
249      cr cr
250      ." =====" cr
251      flag @ 1 = if ." Fehler beim Beschreiben der Diskette" cr then
252      flag @ 0 = if ." Die Diskette wurde ohne Schreibfehler erstellt" cr then
253      ." -----" cr
254      ." Erstes geschrieben. Byte v. RAM-Adresse : " imgbuf 2@ ud. cr
255      ." Letztes geschrieben. Byte v. RAM-Adresse : " imgbyte 2@ 1. d- ud. cr
256      ." Letzte Seite; oder erste, die Fehler hat : " seite @ u. cr
257      ." Letzte Spur ; oder erste, die Fehler hat : " spur @ u. cr
258      ." -----" cr
259      ." Falls Sp/S = 0/0 : Wirklich Diskette im Laufwerk?" cr
260      ." =====" cr cr ;
261
262  : putdiskimage ( imgbuf -- )
263      0 fs!           \ Disk-Image im RAM auf f-Segment = 0 beziehen.
264      161f00. d+      \ Laenge der Diskette (in Bytes) hinzuaddieren.
265      ramtest         \ Ist das vielleicht gar kein Disk-Image (Platz)?
266      161f00. d-      \ Wieder zum Anfang des Disk-Image-Puffers gehen.
267      2dup            \ Zunaechst lege ich den
268      imgbuf 2!       \ Anfang des Disk-Image-Puffers in die 2Variable imgbuf
269      imgbyte 2!      \ und dann (zunaechst auch) in die 2Variable imgbyte.
270      0 spur !        \ Letzte geschriebene Spur zunaechst auf 0 setzen.
271      0 seite !       \ Letzte geschriebene Seite zunaechst auf 0 setzen.
272      0 flag !        \ Fehler-Flag zunaechst auf 0 setzen.
273      begin
274      (putdiskimage)
275      seite @ 0 =     \ Spuren und Seiten beginnen bei 0. Seiten: 0/1.
276      if             \ Wenn bei einer bestimmten Spur die Seite auf 0 steht,
277      1 seite !      \ dann Seite auf 1 schalten und Spur beibehalten.
278      else           \ Ansonsten steht die Seite auf 1.
279      0 seite !      \ Dann Seite wieder auf 0 setzen
280      spur 1+!      \ und Spur (es gibt 50h davon) weiterschalten.
281      then
282      spur @ 50 =    \ Verwendbare Spuren: 0-4f. Wenn 50h erreicht ist,
283      flag @ 0<> or  \ oder auch schon vorher, wenn naemlich ein Fehler
284      until         \ aufgetreten ist, dann begin-until-Schleife verlassen.
285      seite @ 1 =    \ Spuren- und Seiten-Fortschaltung rueckgaengig machen.
286      if            \ if-then hier analog zu if-then eben.
287      0 seite !
288      else
289      1 seite !

```



```
290      spur 1-!
291      then
292      printparams ;    \ Bildschirm-Ausgabe der (Fehler-)Parameter.
293
294      \ Das folgende Wort falsify, mit dem Zubringer-Wort (falsify), steigert das
295      \ Vertrauen in das Gelungensein einer Disketten-Kopieraktion (Klonen).
296
297
298      2variable ad1      \ Anfangsadresse des 1. Disketten-Images (Punktzahl)
299      2variable ad2      \ Anfangsadresse des 2. Disketten-Images (Punktzahl)
300      2variable ad3      \ 1. Fehladresse (Punktzahl) ab ad2, falls ueberhaupt
301      2variable falsi    \ Anzahl der unstimmmigen Bytepaare aus ad1 und ad2,
302                          \ bezogen auf ad2, Punktzahl.
303
304                          \ Es wird vorausgesetzt, dass ad2 > ad1+161f00h.
305
306                          \ Das Folgende ist nur ein zagherfter Versuch. Versierte
307                          \ Programmierer wuerden das sicher geschickter angehen.
308                          \ Im Grunde genommen berechne ich die 'effektiven'
309                          \ Adressen explizit, um mich in der 32-Bit-Adresslogik
310                          \ innerhalb einer 16-Bit-Umgebung nicht zu verlieren.
311
312      code (falsify) ( -- )
313          ad1      #) push \ Hi-Anteil der 2Variablen ad1 auf den Stack legen
314          ad1 2 + #) push \ Lo-Anteil der 2Variablen ad1 auf den Stack legen
315          ebx pop  \ Adress-Zeiger aus ad1 jetzt little-endian in ebx
316          ad2      #) push \ Hi-Anteil der 2Variablen ad2 auf den Stack legen
317          ad2 2 + #) push \ Lo-Anteil der 2Variablen ad2 auf den Stack legen
318          edx pop  \ Adress-Zeiger aus ad2 jetzt little-endian in edx
319          opsize: b9 c, \ Zaehler ecx fuer Bytevergleiche wird jetzt little
320          1f00 , 0016 , \ endian mit 161f00 (Anzahl der Disk-Bytes) geladen.
321          begin \ -----
322              ecx dec \ Naechster Schritt: Zaehler ecx um 1 vermindern
323              ebx push \ ebx auf Stack zwischenspeichern
324              ecx bx add \ ebx := ebx+ecx
325          fs: adsize:
326          8a c, 03 c, \ fs:[ebx] al mov (al = Byte an effektiver Adresse [ebx])
327          edx bx mov \ ebx := edx
328          ecx bx add \ ebx := ebx+ecx
329          fs: adsize:
330          8a c, 23 c, \ fs:[ebx] ah mov (ah = Byte an effektiver Adresse [ebx])
331          ah al cmp \ al <> ah ?
332          0<> if \ Wenn ja, dann
333              ebx push \ ebx auf den Stack legen
334          ad3 2 + #) pop \ Neuen Lo-Anteil der 2Variablen ad3 vom Stack holen
335          ad3      #) pop \ Neuen Hi-Anteil der 2Variablen ad3 vom Stack holen
336          falsi      #) push \ Hi-Anteil der 2Variablen falsi auf den Stack legen
337          falsi 2 + #) push \ Lo-Anteil der 2Variablen falsi auf den Stack legen
338          ebx pop  \ Inhalt von falsi little endian nach ebx legen.
339          ebx inc  \ ebx um 1 erhoeihen.
340          ebx push \ Den erhoehten Wert auf den Stack legen.
341          falsi 2 + #) pop \ Den Lo-Anteil von falsi vom Stack holen
342          falsi      #) pop \ Den Hi-Anteil von falsi vom Stack holen
343          then
344              ebx pop \ ebx fuer naechsten Schritt vom Stack zurueckholen
345          eax ax xor \ eax := 0
346          opsize:
347          eax cx cmp \ cmp = 0 ?
348              0= \ Aussprung nach Abarbeiten von 161f00h Bytes (= 1 Disk)
349          until \ -----
350          next end-code
351
352      : printfalsi ( -- ) \ Ausgabe der Parameter von falsify auf den Bildschirm
353                          \ Die Bytepaare werden von hohen nach niedrigen Adressen
354                          \ hin abgefragt. Die Adressen der unstimmmigen Bytes des
355                          \ 2. Disketten-Images (soweit solche vorhanden sind)
```

```

356          \ werden voruebergehend aufbewahrt. Was uebrig bleibt,
357          \ ist gegebenenfalls die erste solche Adresse.
358
359          . " ===== "                                cr cr
360          . " 1. Disketten-Image ab RAM-Adresse: "      ad1 2@ ud. cr
361          . " 2. Disketten-Image ab RAM-Adresse: "      ad2 2@ ud. cr
362          falsi 2@ 0. d= if ." Die Disk-Images waren byteweise gleich" cr
363          else ." Die Disketten-Images waren verschieden" cr
364          ." Anzahl der unstimmigen Byte-Paare: "      falsi 2@ ud. cr
365          ." 1rst unstimmiges Byte im 2. Image: "      ad3 2@ ud. cr
366          then
367          . " ===== "                                cr cr ;
368
369 : falsify ( ad1. ad2. -- ) \ ad1. ad2. Punktzahlen
370   0 fs!          \ Disk-Image im RAM auf f-Segment = 0 beziehen.
371   ad2 2!         \ Anfangsadresse des 2. Images in die 2Variable ad2
372   ad1 2!         \ Anfangsadresse des 1. Images in die 2Variable ad1
373   0. falsi 2!    \ Anfangswert fuer Anzahl der unpaarigen Bytes
374   0. ad3 2!     \ Ruecksetzen fuer erneuten Aufruf
375   (falsify)      \ Bytepaare vergleichen und Parameter sammeln
376   printfalsi ;  \ Bildschirmausgabe der Parameter
377
378
379 \ =====
380
381

```

Xchars im Microcontroller

Bernd Paysan

Matthias Trute fragte im IRC, was er denn tun müsse, um das Xchar-Wordset in seinem amForth zu implementieren. Ich nutze die freie Seite in der VD, das mal kurz zu erklären. Das Xchar-Wordset ist ein Modul des Forth200x-Standards, mit dem ASCII-kompatible Erweiterungen von Zeichensätzen, auch solche mit variablen Zeichenlängen, verarbeitet werden können. Das ist vor allem das populäre UTF-8, aber auch GBK (VR China), Big5 (Taiwan) oder JIS X 0213 (Japan) haben variable Länge und sind ASCII-kompatibel.

Einleitung

Die Motivation zu der Xchar-Erweiterung kommt aus der Anforderung der ISO, zu Forth doch bitte auch ein Internationalization-Konzept hinzuzufügen. Xchars lösen nur einen Teil davon, nämlich die Zeichensätze — ANS-Forth selbst war ein reines ASCII-System. Anders als etwa Python, das zwischen Strings und Byte-Arrays unterscheidet, und Strings bei der Ein/Ausgabe recodieren muss, wollte ich keine solch gravierenden Eingriffe in das Forth-System vornehmen. Xchar ist also so gebaut, dass man mit wenig Aufwand ein vorhandenes System fit für UTF-8 oder andere Encodings mit variabler Länge machen kann.

So ein Standard-Text ist nicht unbedingt leicht zu lesen, weil Standards immer viel mit Politik zu tun haben, und man das dann so formuliert, dass auch verschiedene Implementierungsoptionen möglich sind. In der Praxis kommen diese Optionen selten in Frage. So definiert Xchar ein „pchar,“ in den meisten Implementierungen nennt man das einfach „Byte.“ Ein Xchar (extended character) besteht halt aus einem oder mehreren Bytes.

String-Befehle selbst funktionieren ohne Änderungen mit Xchars, weil Strings nach wie vor einfach Byte-Arrays sind. Natürlich muss man darauf achten, dass alles 8-Bit-clean ist, auch das Wörterbuch. Dann hat man schon ein System, das weitgehend funktioniert, wenn man ihm UTF-8-Quelltext vorlegt. Gforth 0.6.2, die letzte Version ohne Xchars, schluckt das, ohne zu murren. Nur der Zeileneditor hakt, wenn man Xchars eingibt, und mit dem Cursor darüber hinweglaufen will — weil der eben nichts über die variable Zeichenlänge weiß. amForth, das hat Matthias gleich ausprobiert, verschluckt sich im Wörterbuch, wenn man UTF-8-Zeichen benutzt. Aber das ist sicher ein Problem, das leicht zu beheben ist. Damit ist das Forth-System dann schon mal 8-Bit-clean, das ist die halbe Miete.

Was das Xchar-Wordset nicht macht: Es unterstützt nicht das Umwandeln von einem Encoding in ein anderes. Man kann also nicht mal ein UTF-8-Terminal, mal ein Latin-1-Terminal verwenden, und sich darauf verlassen, dass das dann alles geht. Wenn man solche Features haben möchte, kann man die außerhalb des Standards implementieren, wie es einem gerade gefällt.



Minimaler Befehlssatz

Man muss nicht viel implementieren, um das Xchar-Wordset auf einen Controller zu bringen. Die Erweiterungen (XCHAR EXT) sind nicht Pflicht, und auch nicht nötig, um einfache Operationen auszuführen. Viel mehr wird man auf dem Controller auch nicht machen, nur XCHAR- ist interessant. Ich definiere die Wörter so, dass man sie auf einem Controller einfach implementieren kann.

X-SIZE (xc-addr u1 — u2)

Liefert die Länge des ersten Xchars im String — maximal u1. Kann man sehr einfach implementieren:

```
: x-size ( xc-addr u1 -- u2 )
  >r dup xc@+ drop - r> min ;
```

XC-SIZE (xchar — u)

Liefert die Länge des Zeichens im Speicher. Auch das sehr einfach zu implementieren:

```
Create xc-buf 4 allot
: xc-size ( xchar -- u ) xc-buf xc!+ xc-buf - ;
```

XC@+ (xc-addr1 — xc-addr2 xchar)

Liest ein Xchar aus dem String, analog zu COUNT. Für UTF-8 kann man sich an die Referenz-Implementierung halten:

```
: xc@+ ( xc-addr -- xc-addr' u )
  count dup $80 u< IF EXIT THEN
  $7F and $40 >r
  BEGIN dup r@ and WHILE r@ xor
    6 lshift r> 5 lshift >r >r count
    $3F and r> or
  REPEAT r> drop ;
```

XC!+ (xchar xc-addr1 — xc-addr2)

Speichert ein Xchar in einem String, und gibt die nächste unbenutzte Adresse zurück — also das Gegenstück zu XC@+. Auch hier der Verweis auf die Referenz-Implementierung für UTF-8:

```
: xc!+ ( xchar xc-addr -- xc-addr' )
  over $80 u< IF tuck c! char+ EXIT THEN
  >r 0 swap $3F
  BEGIN 2dup u> WHILE
    2/ >r dup $3F and $80 or
    swap 6 rshift r>
  REPEAT $7F xor 2* or r>
  BEGIN over $80 u< 0= WHILE
    tuck c! char+ REPEAT nip ;
```

XC!+? (xchar xc-addr1 u1 — xc-addr2 u2 flag)

Diese Variante von XC!+ schützt vor Buffer-Overflows. Man kann das mit XC-SIZE leicht implementieren:

```
: xc!+? ( xchar xc-addr1 u1 -- xc-addr2 u2 flag )
  >r
  over xc-size r@ u>
  IF nip r> false EXIT THEN
  r> over + >r xc!+ r> over - true ;
```

XC, (xchar —)

Legt ein Xchar im Dictionary ab. Auch das leicht zu implementieren:

```
: xc, ( xchar -- ) here xc!+ dp ! ;
```

XCHAR+ (xc-addr1 — xc-addr2)

Geht zum nächsten Xchar, ebenfalls trivial zu implementieren:

```
: xchar+ ( xc-addr1 -- xc-addr2 ) xc@+ drop ;
```

Es gibt da noch ein XCHAR-, das nicht ganz so leicht zu implementieren ist, aber sehr nützlich für den Zeileneditor. Bei UTF-8 sucht man rückwärts nach etwas, was entweder ASCII oder größer \$C0 ist.

XKEY (— xchar)

Liest ein Xchar vom Terminal. Auch hier ist die Implementierung Encoding-abhängig.

```
: xkey ( -- xchar )
  key dup $80 u< IF EXIT THEN
  $7F and $40 >r
  BEGIN dup r@ and WHILE r@ xor
    6 lshift r> 5 lshift >r >r key
    $3F and r> or
  REPEAT r> drop ;
```

XKEY? (— flag)

Liefert true zurück, wenn ein komplettes Xchar im Input-Buffer des Terminals angelangt ist. Das heißt, man muss Buffering implementieren. Ich überlasse das als Übung dem Leser, und überlege mir, ob man XKEY? auf dem Controller wirklich braucht.

XEMIT (xchar —)

Gibt ein Xchar auf dem Terminal aus. Das kann man jetzt wieder direkt implementieren:

```
: xemit ( xchar -- )
  xc-buf xc!+ xc-buf tuck - type ;
```

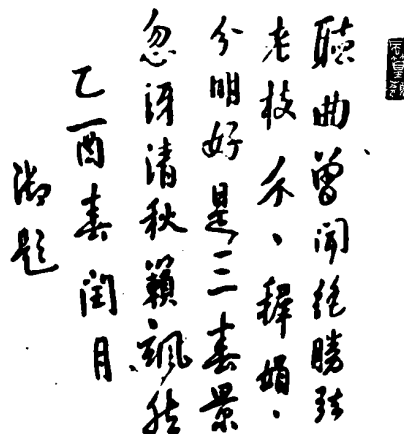


Abbildung 1: Das Thema Schreibrichtung ist nichts für Microcontroller

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) – 46 22 14 91 (p)

bernd.paysan@gmx.de

Treffen: Jeden 4. Donnerstag im Monat um 19:00, im Sommer (Mai–September) im Chilli Asia, Dachauer Str. 151, im Winter im Sloveija Grill, Dachauer Str. 147, 80335 München.

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,

Ulrich Hoffmann

Sicherheitskritische

Tel.: (0 43 51) – 71 22 17 (p)

Systeme

Fax: – 71 22 16

Forth-Vertrieb

Ingenieurbüro

volksFORTH

Klaus Kohl-Schöpe

ultraFORTH

Tel.: (0 70 44) – 90 87 89 (p)

RTX / FG / Super8

KK-FORTH

Homecomputer-Forth

Carsten Strotmann

6502 (Commodore, Atari,

cstrotm@forth-ev.de

Apple 2)

Z80 (Amstrad, CP/M)

M68K (Atari ST, Apple

Mac)

Termine

Mittwochs ab 20:00 Uhr

Forth-Chat IRC #forth-ev

23. Sep–25. Sep EuroForth

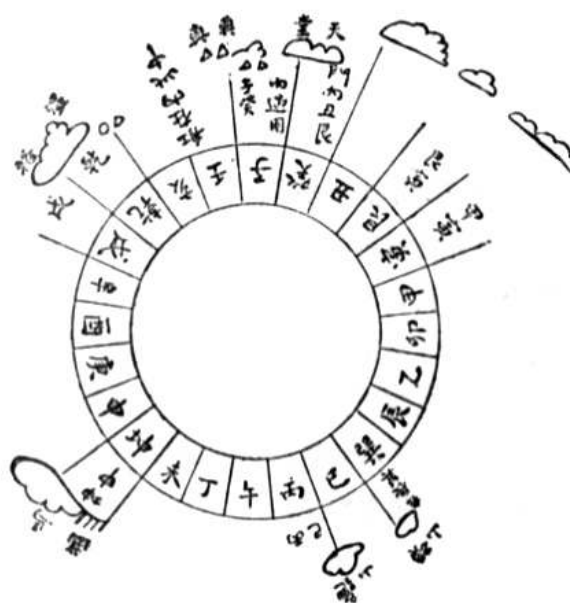
www.complang.tuwien.ac.at/anton/euroforth/ef11

05. Nov Brandenburger Linux-Infotag (BLIT)

blit.org

12. Nov–13. Nov OpenRheinRuhr

openrheinruhr.de



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

Hinweise zu den Angaben nach den Telefonnummern:

Q = Anrufbeantworter

p = privat, außerhalb typischer Arbeitszeiten

g = geschäftlich

Die Adressen des Büros der Forth-Gesellschaft e.V. und der VD finden Sie im Impressum des Heftes.



EuroForth 2011
27th EuroForth Conference
TU Wien, Vienna, Austria
September, 23rd to 25th, 2011



EuroForth is an annual conference on the Forth programming language, stack machines, and related topics, and has been held since 1985. The 27th EuroForth will be held in Vienna, Austria. The conference will be preceded by a Forth 200x standards meeting.

- July 11: Deadline for draft papers (academic stream)
- August 11: Registration deadline (later registration uncertain)
- August 17: Notification of acceptance of academic stream papers
- September 14: Deadline for camera-ready paper submission (academic and industrial stream)
- September 21-23: Forth200x meeting
- September 23-25: EuroForth 2011 conference

Information on earlier conferences can be found at the EuroForth home page. This year's EuroForth will be organized by Ewa Vesely and Anton Ertl.

Accommodation will be in the 4-star Hotel „Erzherzog Rainer“, a limited number of reservations have been made that will expire after August 11.

Links

- Registration Form and Invitation:
<http://www.complang.tuwien.ac.at/anton/euroforth/ef11/registration.pdf>
- Travel Information: <http://www.complang.tuwien.ac.at/anton/euroforth/ef11/travel.html>
- Call for Papers <http://www.complang.tuwien.ac.at/anton/euroforth/ef11/cfp.html>
- EuroForth 2009 Home page <http://www.complang.tuwien.ac.at/anton/euroforth/ef11/>
- EuroForth Home <http://www.complang.tuwien.ac.at/anton/euroforth/index.html>



Hotel „Erzherzog Rainer“, Wien (Photos: <http://www.schick-hotels.com/hotels-wien-fotos-rainer.de.htm>)