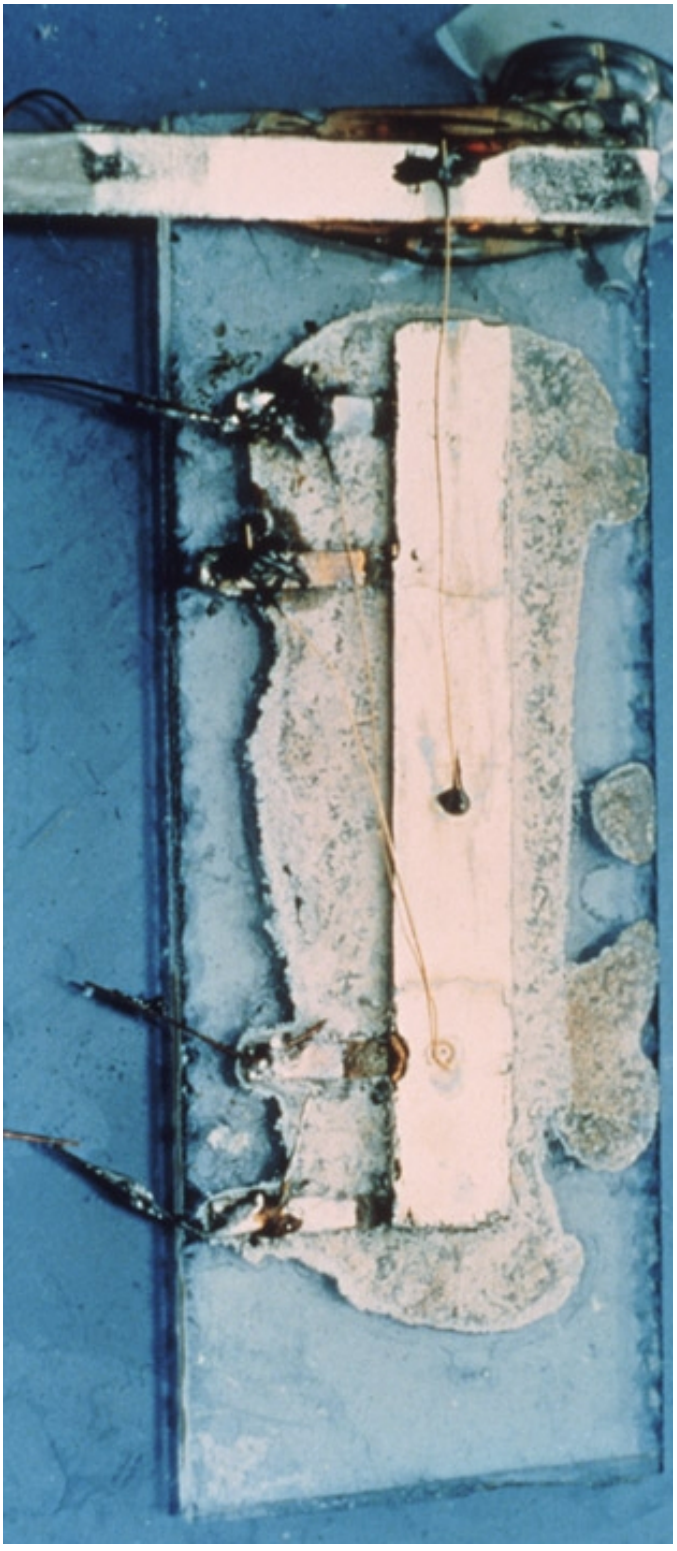
Vierte Dimension

1/2015

Das Forth–Magazin

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

In dieser Ausgabe:



Dienstleistungen und Produkte fördernder Mitglieder des Vereins

tematik GmbH
Technische
Informatik

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

Seit 2001 entwickeln und vertreiben wir unter dem Markennamen "Servonaut" Baugruppen für den Funktionsmodellbau wie Fahrtregler, Lichtanlagen, Soundmodule und Funkmodule. Unsere Module werden vorwiegend in LKW-Modellen im Maßstab 1:14 bzw. 1:16 eingesetzt, aber auch in Baumaschinen wie Baggern, Radladern etc. Wir entwickeln mit eigenen Werkzeugen in Forth für die Freescale-Prozessoren 68HC08, S08, Coldfire sowie Atmel AVR.

LEGO RCX–Verleih

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

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

Inhalt:

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

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

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

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

KIMA Echtzeitsysteme GmbH

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

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

FORTecH Software GmbH
Entwicklungsbüro Dr.–Ing. Egmont Woitzel

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

Ingenieurbüro
Klaus Kohl–Schöpe

Tel.: **(0 82 66)–36 09 862**
Prof.–Hamp–Str. 5
D–87745 Eppishausen

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

Impressum

Name der Zeitschrift

Vierte Dimension

Herausgeberin

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

Redaktion & Layout

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

Anzeigenverwaltung

Büro der Herausgeberin

Redaktionsschluss

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

Erscheinungsweise

1 Ausgabe / Quartal

Einzelpreis

4,00€ + Porto u. Verpackung

Liebe Leser,

und wieder beginnt ein neuer Jahrgang Forth–Magazin. Wir zählen `hex 1F` also das einundreißigste Jahr :-)

Dieses Heft konzentriert sich darauf, wie Forth geeignet auf Microcontrollern programmiert werden kann. Ein Problem, dem viele von uns auf Kleinst–Mikrocontrollern begegnen, ist der effiziente Umgang mit dem verfügbaren Speicher. Selbst ein kleines Forth–System bringt dabei einen gewissen Bodensatz mit, der in fertigen Applikationen nicht mehr benötigt wird, etwa `DUMP`, der outer Interpreter oder der Assembler.

À propos *Assembler*. Wie man auf den Assembler im Zielsystem verzichten kann, ist eine Frage, die gleich aus drei Blickwinkeln diskutiert wird: Zunächst ist da Brad Rodriguez englischsprachiger Artikel, der beschreibt, wie man den Assembler transient laden kann und ihn so nach getaner Arbeit wieder aus dem Zielsystem entfernt. Zweitens erreicht uns die Nachricht von Manfred Mahlow, dass er seine Entwicklungsumgebung `e4thcom` aktualisiert hat. Sie unterstützt jetzt Cross–Assembler und erlaubt so das Definieren von `Code`–Worten, die auf dem Host–System in Maschinencode übersetzt werden und dann im Komma–Code in das Zielsystem übertragen werden.

À propos *Komma–Code*. Dies ist eine Erfindung von Albert Nijhof und Willem Ouwerkerk, die sie im Zusammenhang mit ihrem `noForth` vorgestellt haben. `Code`–Worte werden zunächst mit geladenem Assembler im Zielsystem erzeugt, dann aus dem Programmspeicher ausgelesen und in Form von Bytes, die mit dem Forth–Wort `,` ins Wörterbuch geschrieben werden können, dargestellt. Diesen Komma–Code kann man dann ohne Assembler ins Zielsystem laden und voilá: ein Zielsystem mit Applikation aber ohne Assembler.

À propos *noForth*. Seit Anfang 2015 gibt es eine Version dieses Forth–Systems für MSP430, die mit einem Target–Compiler und vollem Quellcode kommt. Wir hatten die Gelegenheit, Albert und Willem ein paar Fragen rund um `noForth` zu stellen und so die Besonderheiten dieses Systems zu ergründen. Es stecken neben Komma–Code noch eine *Menge* gute Ideen in `noForth`. Danke Albert und Willem, dass Ihr sie mit uns teilt.

À propos *MSP430*. Dieser kleine Mikrocontroller zieht mehr und mehr Aufmerksamkeit auf sich. Matthias Trute hat begonnen, sein `amForth` zu portieren und berichtet uns über sein Experiment. `amForth` setzt Recognizer ein, um die Verarbeitung von Programmtext noch besser konfigurieren zu können.

À propos *Recognizer*. Matthias zeigt uns in einem weiteren Artikel über Recognizer, wohin die Reise mit Recognizern führen kann. Anton Ertl schlug vor, dabei sogar ganz auf das SEARCH-ORDER–Wordset zuverzichten, das im ANS-Forth-94–Standard definiert wurde und sich auch noch in Forth-2012 findet.

À propos *Standard*. Wer es noch nicht mitbekommen hat: Es gibt seit dem 10. November 2014 den Forth-2012–Standard, der viele Dinge beschreibt, die in kommerziellen und freien Forth–Systemen üblich sind. Bernd Paysan hat im letzten Heft 2014-3/4 schon kurz vorgestellt, was es dabei Neues gibt.

Nun viel Freude beim Lesen dieser Ausgabe.

Ulrich Hoffmann

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
`http://fossil.forth-ev.de/vd-2015-01`

Atmega Tipps – mit Vorgeschichte

Über Atmega fuses, Launchpad ttyACM0, und einen sehr preiswerten USB-Tx-Rx-Umsetzer.

Ich beschäftige mich nicht jeden Tag mit embedded programming. Das führt zu Erinnerungslücken und in der Folge dazu, dass ich versuche, mir lesbare Notizen zu machen. Quasi sollen das Briefe an mein zukünftiges ich sein. Daraus hier einige Tipps mit ihren jeweiligen Vorgeschichten.

Nach langer Zeit konnte ich mich wieder mit amforth von Matthias Trute, hier in der Version 5.6, auseinandersetzen. Das gibt es inzwischen auch für den MSP430G2xxx von Texas Instruments (Launchpad). Zu Hause ist amforth auf den verschiedenen Atmega–8Bit–MCUs. Der Umzug zum MSP ist noch in vollem Gange. Das Readme zeigt einige Lücken und To–Do's auf. Zuerst wollte ich sowohl alte Kenntnisse auffrischen, als auch den aktuellen Stand kennenlernen.

Amforth kommt als Sourcefiles. Fast alle WORDS in amforth sind in Assembler geschrieben. Bevor man also eine Hex–Datei in eine MCU brennen kann, muss man sie von einem Assembler kompilieren lassen.

Vor einiger Zeit hat ein STK500 (STK = Starter–Kit) von Atmel den Weg zu mir gefunden. Das besondere daran: Es kann Hochvoltprogramming. Damit kann man 'verfuste' AtmegaXXX-MCUs retten. Fuses sind zwei (oder drei) Bytes in einer Atmega–MCU, mit denen man bestimmte Eigenschaften ein- und ausschaltet. Unter anderem sind dies Taktquelle (Quarz oder äußerer Oszillator oder interner), Taktteiler (1 oder 8), Startverhalten (timing), caveat: Schreibschutz!

Ein Beispiel: Angenommen, man hat die Fuse so gesetzt, dass die MCU den äußeren Takt eines Oszillators verwendet. In einem Entwicklungsboard mit Oszillator läuft alles gut. Jetzt baut man die MCU in eine Schaltung ein und vergisst dabei, der Schaltung einen Oszillator zu spendieren. Dann läuft die MCU nicht an und wirkt wie tot. Das Gleiche gilt, falls man Oszillator (schwingt von selbst) und Quarz verwechselt. Wer jetzt sagt: „Wem passiert denn so etwas!", der bemühe eine Suchmaschine zu 'fuses' und 'atmega'. Es passiert erstaunlich oft!

Eine andere Quelle für falsche Fuses sind Nachlässigkeiten oder Fehler beim Lesen von Datenblättern. Das gleiche Bit/Byte kann bei unterschiedlichen MCUs schon mal eine andere Bedeutung haben. Manche Fusebytes können im normalen SPI–Modus gesetzt, aber nicht zurückgesetzt werden. Dann hilft nur ein Entwicklungsboard, ein Programmer oder eine selbst gebaute Schaltung evtl. im Steckbrett, die die Fusebytes mit Hochvoltprogramming zurücksetzen können. In der Regel wird man die Fusebytes auf 'sichere' Werte setzen. D.h.: interner Takt, geteilt durch 8 = 1 MHZ und lange Startzeiten. Wenn die MCU damit ansprechbar ist, weiß man, dass sie noch 'lebt' und kann gezielt die richtigen Fusebytes für sein Projekt setzen. Ich verwende zum Brennen (programmieren) der Fusebits unter Linux das Programm *avrdude*. Sichere Fusebits kann man entweder im Datenblatt zu der einzelnen MCU oder u.A. hier nachlesen:

`PocketMagic - Where Technology meets magic.`
`http://www.pocketmagic.net/`
`2013/03/how-to-set-the-avr-fusebits/`

Recht beliebt ist auch der FUSEbit Calculator von Mark Hämmerling.
`http://www.engbedded.com/fusecalc`

TIPP 1: Setze sichere Fusebytes (Hochvoltmodus) !

Jetzt, da ich wieder sicher war, dass ich lebendige MCUs hatte und sie auch programmieren konnte, wollte ich die aktuelle Version (5.6) von amforth kompilieren und brennen. Beim amforth ist das so organisiert, dass in einem Makefile in den zur Ziel–MCU passenden *inc– und *asm–Dateien Werte gesetzt werden. In der Hauptsache sind das der MCU–Typ, und die Taktrate, unter der die MCU später laufen soll, sowie die Baudrate der seriellen Schnittstelle. Der Aufruf von `make clean` und `make install` stößt dann alles Notwendige an und im Erfolgsfall bekommt man eine MCU, die in der Zielschaltung das tut, was sie soll, wozu oft eben auch die Kommunikation via serieller RX–TX Schnittstelle gehört. Nun gut. Bei mir brach der Assembler mit einer Fehlermeldung ab, weil der Algorithmus zur Bestimmung der Zeitwerte/Teiler zur Baudrate bei dem ausgewählten Takt eine zu große Fehlerbreite meldete. Hier war ein Wert von einem Promille eingestellt. In der MCU wird später der Zeitwert für die high–low–Pegel im Verhältnis zur MCU–Frequenz ausgerechnet. Dabei kommt es zwangsläufig zu Rundungsfehlern. Darauf wird schon beim Kompilieren des Hexfiles geachtet. Das ist von Matthias sehr klug gemacht und verführte mich ein wenig zum Spielen, mit Baudrate, Fehlerabweichung und Taktfrequenz. Mein erster Versuch bestand darin, die Fehlerbreite einfach hoch zu setzen und mich an den Wert heranzutasten, der von dem Algorithmus gerade noch akzeptiert wurde und im Entwicklungsboard funktionierte. Das gelang erstaunlich gut. Heutige USB–RS232–Umsetzer sind recht fehlertolerant. Ein Beispiel: Bei einer MCU–Frequenz von 12MHZ, 38400Baud, wird ein Fehler von 2,4‰ beim Kompilieren akzeptiert und führt dennoch zu einer funktionierenden Kommunikation. Bei gleicher MCU–Frequenz und 19600Baud gilt dies für einen Fehlerwert von 1‰. Bei einer MCU–Frequenz von 1MHZ und einer Fehlerbreite von 1‰ ist die höchstmögliche Baudrate 4800. Soweit so gut. Das STK500 hat einen einstellbaren Baudratenoszillator, der eine eingesetzte MCU mit einem Takt von 3,686 (4)MHZ und 'geraden' Teilen davon versorgen kann. Das habe ich auch einmal ausprobiert und war sehr verblüfft. Ein einfacher Atmega168, der mit einer MCU–Frequenz von 3,686 MHZ läuft, bewältigt erfolgreich eine Baudrate von 230400!!! Wenn es also auf eine schnelle Kommunikation ankommt, lohnt sich auf jeden Fall die Anschaffung eines Baudratenquarzes! Analoges gilt für Uhrenquarze. Hier gibt es allerdings externe RTC (Realtime clock) Bausteine.

Tipp 2: Verwende, wenn nötig, einen angepassten Quarz!

Wie erwähnt, wird zur Zeit amforth auf das Launchpad von Texas Instruments portiert. Dort ist die MCU ein MSP430G2xxx. In gewisser Hinsicht ist das Launchpad etwas, von dem wir früher geträumt hatten: Programmer

und USB-serielle Schnittstelle in einem. Es gibt nur eine Verbindung zum PC, ein einfaches mini–USB Kabel. Die Launchpad–Platine ist zweiteilig und könnte an einer aufgedruckten Linie getrennt werden. Der Teil mit dem Aufdruck `MSP-EXP430G2` enthält den Steckplatz für die MCU (MSP430Gxxx) und ein wenig Peripherie: Taster, LEDs, und herausgeführte Portpins. Der andere Teil, Aufdruck `EMULATION`, sorgt für die Umsetzung von USB auf serielle Rx–Tx–Kommunikation und die Stromversorgung. Der Umsetzer meldet sich im Linux via USB als ACM0 von Texas Instruments, Inc., eZ430 Development Tool, an. Die beiden Platinenteile sind mit einem Jumperfeld aus fünf Jumpern miteinander verbunden.

Das Programmieren des Launchpads gelang mit `mspdebug rf2500 „prog amforth.hex“` auf Anhieb. Doch konnten unter Linux sowohl *minicom*, als auch *picocom* sowie *screen* und *cutecom* das Device `/dev/ttyACM0` zwar öffnen, froren aber ein, anstatt mit dem Launchpad zu kommunizieren. Nach dem Beenden dieser Programme dauerte es noch einige lange Sekunden, ehe der dazugehörige Prozess terminierte.

In den üblichen verdächtigen Gruppen `dialout` und `plugdev` war ich eingetragen. Letztendlich stellte sich heraus, dass ein Beitritt in die Gruppe `tty` half. Ebenso wichtig ist es, nach einem Programmieren mittels `mspdebug`, die Verbindung zum PC zu trennen - also das USB-Kabel ziehen - und wieder aufzubauen - USB-Kabel einstecken, einige Sekunden warten.

Tipp 3: Bei Versuchen mit `ttyACM0` sei Mitglied in `tty`, `plugdev` und `dialout`! Das Ziehen des USB–Steckers kann hilfreich sein.

Wie erwähnt, enthält das Launchpad einen USB–Rx–Tx–Umsetzer. Diesen kann man unabhängig vom Rest der Platine benutzen. Dicht bei der USB–Buchse finden sich zwei Lötaugen, die 5V USB–Spannung führen. Dort einen Lötpin angebracht und schon kann man eine kleine Schaltung mit 5V versorgen. 3,3V sind an einem der VCC-Pins auf der Platine abgreifbar. Nun noch die Jumper entfernen, die die Rx– und Tx–Signale (Platinenaufdruck als Koordinatenangabe: Tx=Tx:Tx und Rx=Rx:Rx, Wer die Platine vor sich hat, versteht das!) abgreifen und jede beliebige MCU, die über Rx–Tx–Pins verfügt, kann angesprochen werden. Bei mir ist es zur Zeit ein Atmega32 in einem Pentabug.

Zu dieser EMULATION eine Besonderheit. Die Chip–Software–Kombination auf dem Platinenteil ist sehr intelligent, es ist ihr in weitem Rahmen egal, mit welcher Baudrate sie vom PC aus angesprochen wird, sie stellt sich darauf ein und leitet die empfangenen Signale mit einer Baudrate von 9600 an das Target weiter. Also: nicht verwirren lassen, wenn eine MCU mit 9600 Baud scheinbar auf 230400 hört!

Tipp 4: Die EMUALTION-Platine eines Launchpads ist ein preiswerter, allfälliger USB-Rx-Tx-Umsetzer!

Martin

Das Titelbild, Kilby, Noyce, und die Integrierte Schaltung

> Jack St. Clair Kilby (* 8. November 1923 in Jefferson City, Missouri; + 20. Juni 2005 in Dallas, Texas) war ein US-amerikanischer Ingenieur. Er gilt zusammen mit Robert Noyce als Erfinder der integrierten Schaltung, wofür er den Nobelpreis für Physik erhielt, und wird als „Vater des Mikrochips“ bezeichnet. ... 1958 begann er seine Arbeit bei Texas Instruments, wo er als Neueinsteiger keinen Sommerurlaub hatte. Er hatte das Labor für sich allein und seine Überlegungen. Dies brachte ihm Zeit, um sich mit der *Tyranny of numbers* zu beschäftigen, worunter man damals im Computerdesign das Problem verstand, dass neue Designs immer mehr Komponenten aufwiesen, die sich immer schwieriger verdrahten ließen. Er kam zu dem Schluss, dass eine Lösung durch die Verwendung von Halbleitern möglich sei. Am 24. Juli 1958 beschrieb er in seinem Labortagebuch erstmals seine Idee, Transistoren, Widerstände und Kondensatoren zu einem Bauteil zusammenzufügen. Am 12. September 1958 präsentierte er ein erstes Exemplar einer funktionierenden Schaltung auf einem Halbleiter. Nicht viel mehr als ein Stück Germanium mit einigen Kabeln auf einem Stück Glas, etwa so groß wie eine Büroklammer, war alles was zuerst zu sehen war. ...

Bei TI fing also alles an, und da wir hier Forth für deren jüngere MCUs abhandeln, bot es sich an, gedanklich mal ganz an den Anfang zurück zu gehen. Wer es nachvollziehen möchte ist eingeladen zum mitsurfen. *mk*

Zitat aus: `http://de.wikipedia.org/wiki/Jack_Kilby`

Titelbild: First Integrated Circuit `http://www.ti.com/corp/docs/kilbyctr/downloadphotos.shtml`

Link: `http://www.ti.com/corp/docs/kilbyctr/kilby.shtml`

e4thcom - Ein Terminal für eingebettete Forth-Systeme

Manfred Mahlow hat im Januar dieses Jahres eine erweiterte Auflage seines Terminals herausgebracht. Das Terminal kann Quellcode bedingt und unbedingt in das Zielsystem hochladen. Es läuft unter Linux 32–Bit–Systemen, X86, und Raspberry/Raspbian. Die jüngste Version e4thcom 0.4.4 bietet eine Reihe von Änderungen und Ergänzungen:

- Für Raspberry/Raspbian wurde das ausführbare Image dazugegeben.
- Die noForth–Unterstützung wurde ergänzt. Es versteht sich also nun mit 4e4th, amforth, mecrisp und noForth.
- In der e4thcom–Quelle wurde der zielsystemspezifische Code in kleine Dateien ausgelagert, die Forth–Source–Code enthalten. Damit können Forth–Kundige leicht weitere Zielsysteme anpassen.
- Ein Cross–Assembler–Interface wurde eingebaut.
- Der MSP430–Cross–Assembler des noForth ist bereits als gebrauchsfertiges Beispiel enthalten.
- Eine Datei mit allen Register- und Bit–Namen für die MSP340G22553–MCU ist nun gebrauchsfertig enthalten.

mk

`http://www.forth-ev.de/wiki/doku.php/en:projects:e4thcom`

The Attila Book

Attila ist eine neue (2012), kleine und schnelle Implementation der Programmiersprache Forth. Es wurde als Experimentierfeld für Computersprachendesign und deren Implementationen auf eingebetteten Systemen gemacht, und speziell auf Sensor–Netze zugeschnitten. Es wird cross–compiliert, wodurch sehr kleine Zielsysteme erzeugt werden können.

Die aufgeräumte Webseite des Autors ist gleichzeitig die Referenz für Attila, ein umfangreiches Handbuch zum Forth und zum Crosscompiler. Der Kern von Attila wurde in C verfasst.

Der Autor beschreibt im besten Bachelorarbeitsstil seine Motivation und die Geschichte hinter Attila. Dabei wird deutlich, warum er Forth benutzt, für vernetzte Sensoren, und als Zugang zu den inzwischen „allgegenwärtigen verteilten Systemen“, wie er sich ausdrückt. Klingt nach einer spannenden Lektüre. *mk*

Quelle: The Attila Book, Simon Dobson, School of Computer Science, University of St Andrews, UK.

`http://www.threaded-interpreter.org/manual/book/attila.html"`

Zen of LaunchPad

Noch mehr Lesestoff hat Chen–Hansen Ting soeben bereitgestellt. Zusammen mit seiner jüngsten 430eForth–Version für das TI MSP430-LaunchPad ist auch das 430eforth–Manual erschienen, frisch überarbeitet und betitelt *Zen of LaunchPad*.

Auf 157 Seiten wird das ganze eForth genau erklärt. Warum es so klein werden konnte - grad mal 4K im Flash–Speicher nimmt es ein - und wie jedes Wort gemacht ist, mitsamt dem Code-Listing, Glossar. Er begründet, warum für so eine MCU wie den MSL430G553 eine einzige überschaubare Assembler–Quelldatei genügt, um das Forth zu erzeugen, und ein einfaches Werkzeug - die 430eForth-IDE[1] - um den Chip zu flashen, getreu seiner Philosophie des „Zen of Forth“. Ein Werk, das man sich auch gedruckt zulegen sollte!

Wie bei Ting üblich, bekommt man das Ganze fast geschenkt, gegen eine minimale Gebühr, die sich in diesem Fall aber besonders lohnt. Denn hier bekommt man exemplarisch vorgeführt, wie man einen *Compiler für eine Sprache* baut, reduziert auf das Allerwesentlichste - „back to the roots“. *mk*

`http://www.offete.com/index.html`

amforth 5.6, and porting it to MSP430

I just released a new version of amforth. For the Atmega controllers it fixes a few regressions introduced with 5.6. The *MSP430 port* made a few huge steps forward. Most notably are an almost complete double cell number support (thanks to Martin for some assembly words), and many many small fixes almost everywhere. That work's not yet finished however, so please see the MSP430 still as beta. Common for both are a few more words from the Forth 2012 standard: `NAME>x`. *Matthias*

Quelle: Amforth-devel@lists.sourceforge.net vom 01.02.2015

Fortsetzung auf Seite 23

[1] Eine Spezielversion der 4e4th-IDE von Dirk Brühl für das eForth

A Transient MSP430 Forth Assembler

B. J. Rodriguez

This describes a "transient" assembler for the MSP430 CamelForth. The basic idea is that the assembler is loaded into high application memory. That assembler is then used to add CODE words to the low end of application memory. When all CODE words have been defined, the assembler is deleted and its memory space recovered. That freed memory space is then available to compile more of the application. The specific example here is the MSP430G2553. Under MSP430 CamelForth*, there is 8K bytes of application memory (flash ROM) available from $C000 to $DFFF. (The CamelForth kernel occupies the remaining 8K bytes from $E000 to $FFFF.)*

The Assembler (asm430.fth)

The assembler needs to be as small as possible, so it is a stripped-down „Forth-style“ (postfix) assembler with minimal error checking. The assembler is a standalone program; it can be separated from the „transient dictionary“ code (by removing references to `| ALT` and `PRIMARY`) and compiled as a normal part of the application.

Registers are PC, SP, SR, and R4 through R15. The assembler also recognizes these aliases for several MSP430 registers, corresponding to the ITC CamelForth model:

RSP (Return Stack Pointer) is SP

PSP (Parameter Stack Pointer) is R4

IP (Interpreter Pointer) is R5

W is R6

TOS (Top Of Stack) is R7

INDEX is R8

LIMIT is R9

X is R10

Y is R11

Q is R12

T is R13

Addressing modes are listed in tabel 1:

The assembler does not automatically detect the "short form" immediate values (using the constant generator). Instead, you must explicitly use the operands `8# 4# 2# 1# 0#` and `-1#` (having no space between the number and the # character).

Two-operand instructions take the form *source destination opcode* . The following opcodes are recognized; note that the trailing comma is part of the opcode:

```
MOV, MOV.B,
ADD, ADD.B, ADDC, ADDC.B,
SUBC, SUBC.B, SUB, SUB.B,
CMP, CMP.B,
DADD, DADD.B,
BIT, BIT.B, BIC, BIC.B, BIS, BIS.B,
XOR, XOR.B, AND, AND.B,
```

For example, to add 4 to register R15, using the short form constant, you would write `4# R15 ADD,`

One-operand instructions take the form *destination opcode* . The following opcodes are recognized:

```
RRC, RRC.B, SWPB, RRA, RRA.B, SXT,
PUSH, PUSH.B, CALL,
```

To call a specified address you use the immediate addressing mode. For example, to call absolute address $1234, you would write `HEX 1234 # CALL,`

Zero-operand instructions are `RET,` and `RETI,` .

Jump instructions take the form *address opcode* . The following opcodes are recognized:

```
JNE, JNZ, JEQ, JZ, JNC, JLO,
JC, JHS, JN, JGE, JL, JMP,
```

Note that on the MSP430, `JLO,` is the same as `JNC,` and `JHS,` is the same as `JC,`

Emulated instructions are not implemented. They're not difficult; they just take up scarce program space. It is the responsibility of the programmer to use the "actual" MSP430 instruction instead of using the emulated instruction; for example, instead of `R6 POP` you must write `SP @+ R6 MOV,`

Structured conditionals (Forth style) are implemented, as follows:

```
cond IF, ... THEN,
cond IF, ... ELSE, ... THEN,
BEGIN, ... cond UNTIL,
BEGIN, ... cond WHILE, ... REPEAT,
```

Note that the assembler conditionals have a trailing comma; this distinguishes them from the similar high-level Forth conditionals. The conditions that can be used are

```
Z NZ C (or HS) NC (or LO) P L GE NEVER
```

`P` is "positive" which is the opposite of `N` (negative). Because these conditionals assemble the "opposite" jump instruction, there is no `N` condition (because there is no JP instruction). Similarly, `NEVER` assembles an unconditional jump instruction; its main use is to create an infinite loop with `BEGIN, ... NEVER UNTIL,`

Macros can be easily implemented as Forth words. Register names are merely constants which are placed on

Notation	Description
reg	register mode
index reg)	indexed mode, equivalent to TI assembler index(reg)
reg 0)	indirect register mode, equivalent to @reg (we can't use the symbol @)
reg @+	autoincrement mode, equivalent to @reg+
addr &	absolute mode, equivalent to &addr
value #	immediate mode, equivalent to #n
index PC)	symbolic mode (PC relative addressing)

Tabelle 1: Addressing modes

the stack; addressing modes merely modify those constants (some addressing modes, such as `#` and `&,` also provide the register). Opcodes are also simple Forth words that consume parameters from the stack. One macro is defined in the assembler, the indirect-threaded `NEXT,`:

```
: NEXT, IP @+ W MOV, W @+ PC MOV, ;
```

Defining Assembly-Language Words in the Forth dictionary is done with `CODE, PROC,` and `ENDCODE`:

```
CODE name ...assembler code... END-CODE
PROC name ...assembler code... END-CODE
```

The difference is that for a `CODE` word, the assembler code is executed when "name" is used. For a `PROC` word, "name" merely leaves the address of the assembler code on the stack. This is useful for writing subroutines. The general rule is: if it ends with `RET,` or `RETI,` it should be a `PROC`. If it's a `CODE` word, it should end with `NEXT,`.

The Alternate Dictionary (altdict.fth)

CamelForth uses the simplest of Forth dictionary structures: a single, singly-linked, list of words. It does not implement any of the ANS extensions to support multiple word lists. So how do we add the assembler to the dictionary, and later discard it? On a RAM-based system this would be easy. We could reset the Dictionary Pointer (DP) to a high address, compile the assembler there, and then set the DP back to the low address where we want our application to be compiled. When through with the assembler, we could then "patch" the dictionary links to remove the assembler from the linked list. But in a flash-ROM based system like the *MSP430G2553*, a link can only be written once. Links can't be patched. The solution involves keeping an alternate dictionary pointer (`ADP`), and also an alternate LATEST pointer (`ALATEST`). Recall that `DP` – or `IDP` in split RAM/ROM systems like the '*G2553* – points to the next free memory location, while `LATEST` points to the header of the last word added to the dictionary (the head of the linked list). `IDP` controls where new code is compiled; `LATEST` controls where Forth words are searched. What we want to build is this: The words of the application are located at the low end of the user ROM (shown here as $C0xx), and link back directly to the "common" words. (As shown here, the last common word is `PRIMARY` -this will be explained shortly.) `LATEST` points to the last word of the application. `IDP` points to the next available location in "low ROM." The assembler is located at the high end of the user ROM space (shown here starting at $D400). It also links back to the common words. `ALATEST` points to the last word in the "alternate" dictionary, and `ADP` points to the next available location in "high ROM." (User ROM ends at $DFFF.) Now, CamelForth doesn't know anything about alternate dictionary pointers, so we add three new words to control searching and compiling:

|	(–) exchange `LATEST` and `ALATEST` (this is the vertical bar or "pipe" symbol)
ALT	(–) exchange `IDP` and `ADP`
PRIMARY	(–) exchange as required to ensure that `LATEST` and `IDP` point to the "primary" dictionary (the lower dictionary address).

Not shown in the diagram (fig. 1) are the two new variables, `ADP` and `ALATEST`, which are implemented as CamelForth USER variables.

Adding the assembler to the Alternate Dictionary

To begin adding words to the alternate dictionary, you simply set `ALATEST` equal to `LATEST`, set `ADP` equal to the desired high address, which must be on a flash page boundary, switch to the alternate pointers with `|` and `ALT`, and add a marker word, in this example `MARKER DISCARD`

By setting `ADP` to a flash page boundary, we ensure that the marker word begins at that boundary, and that it will later erase all the way back to that boundary – that is, it will erase the entire assembler if properly used. The assembler can then be compiled. When it is finished, typing `| ALT` (or `ALT |`) will switch back to the primary dictionary. Or you could use the word `PRIMARY`, which does a "smart" swap. `PRIMARY` assumes that the alternate dictionary is at a higher address than the primary dictionary. If it sees that `IDP>ADP`, it knows to swap the dictionary pointers. Similarly, if it sees that `LATEST>ALATEST`, it knows to swap those pointers. Of course, this is only safe to use after `ADP` and `ALATEST` have been initialized.

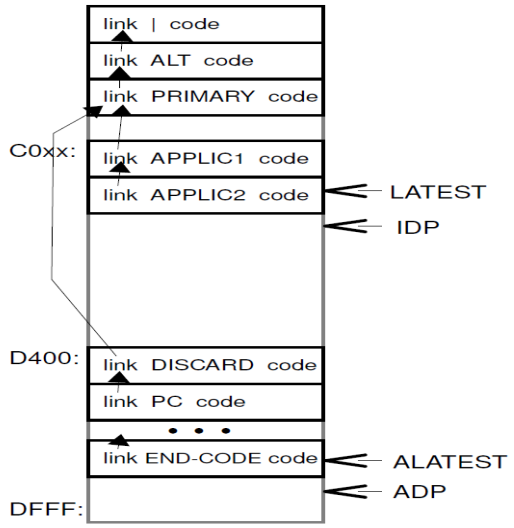


Abbildung 1: Linking the Alternate Dictionary

Using the Alternate words

There are three useful combinations:

1. Primary `DP` and primary `LATEST` – the "normal" Forth mode, used to add application code to low ROM. The assembler is invisible.
2. Alternate `DP` and alternate `LATEST` – used to add new definitions to the Alternate Dictionary (as in the above example, when adding the assembler to the dictionary).
3. Primary `DP` and alternate `LATEST` – makes the Alternate Dictionary visible in the search order, but adds new code to the application in low ROM. This is how you use the assembler.

You can't just type `| CODE name ... END-CODE |` however, because as soon as you type `|` new forth headers will be linked into the alternate dictionary. So `CODE` and `PROC` are defined to first create a header (in the primary dictionary), and then switch to the alternate dictionary, allowing access to the assembler. Similarly `END-CODE` is defined to switch back to the primary dictionary. Obviously `CODE` and `PROC` must themselves reside in the primary dictionary, so they can be seen before the switch. So to create an assembler definition you simply type

```
CODE name ...assembler code... END-CODE
PROC name ...assembler code... END-CODE
```

Accessing application words when assembling

Because of the way the dictionaries are linked (see above), when using the assembler you also have access to all the words of the CamelForth kernel...but you do not have access to any words defined in your application. Such as variables! The way to handle this is to temporarily "flip" the search order, just long enough to access the application word. This is why `|` has such a short name. Here's an example:

```
VARIABLE TICKS
PROC TIMER-IRPT
... 1# | TICKS | & ADD, ...
RETI, END-CODE
```

Bracketing the word `TICKS` with vertical bars ensures that it is found (in the primary dictionary). Note that the "closing" bar must precede the addressing mode & , because & is in the assembler (in the alternate dictionary).

Equates

The alternate dictionary allows a useful function, normally only available in metacompilers: the EQU (equate) directive. An `EQU` is like a `CONSTANT`, except that it occupies no space in the application dictionary. It is assembled the same as a numeric literal. This is particularly valuable in assembler programming, because the *MSP430* has many special function registers that we would really prefer to program by name. Such as, say, `TA1CTL`. Defining CONSTANTs for all the MSP430 registers would take a lot of space, and if referenced only in assembler code, those Forth words will only be used at compile time. It's not sufficient to simply define a CONSTANT in the alternate dictionary. Because if you use that symbolic name in high-level Forth code, the address of the CONSTANT word will be compiled. When the assembler is removed (and the alternate dictionary ROM erased), that CONSTANT will no longer exist, and code which uses it will crash! So use `EQU` instead:

```
: EQU ( n -- )
  <BUILDS I, IMMEDIATE
  DOES> I@ POSTPONE LITERAL ;
```

This is simple enough to explain. `n EQU name` creates a new word "name" which stores the value n. When "name" is executed, the value `n` is fetched and the action of `LITERAL` is performed. If compiling a word, `n` is compiled as a literal value (e.g. `lit, n`). If interpreting, `n` is left on the stack. The address of the word "name" isn't compiled anywhere. You can put equates in either dictionary, but they're most useful in the alternate dictionary:

```
PRIMARY
ALT | ( put equates in alternate dictionary )
HEX
180 EQU TA1CTL
182 EQU TA1CCTL0
184 EQU TA1CCTL1
186 EQU TA1CCTL2
...
PRIMARY
```

Remember that `ALT` causes the new words to be compiled in high ROM, and `|` causes the new words to be linked into the alternate dictionary list. Both are required. Since these equates reside in the alternate dictionary, they are visible when using the assembler:

```
PROC TA1IFG_IRPT
  1# TA1CTL & BIC, ... RETI,
END-CODE
```

On the other hand, this means that such equates are not normally visible to high-level Forth code. Once again, a quick search order swap is required:

```
: INIT_TA1 0 | TA1CTL | ! ... ;
```

Bracketing TA1CTL with vertical bars ensures that it will be found, and the literal value compiled. (For this reason, | is an IMMEDIATE word.)

Cleaning Up

When you have finished writing all your CODE and PROC words, and finished using the equates you've defined, you can remove the assembler and erase the flash ROM it used, so that the application can grow upward into that ROM space. This is done with

```
| ALT DP @ DISCARD DP ! | ALT
```

What this does is switch to the alternate dictionary with | ALT , and then execute DISCARD. The "marker" word DISCARD will back up the LATEST pointer to the word preceding itself, back up the IDP to the starting address of DISCARD, and then erase itself and everything following from the dictionary (up to the end of user ROM). This basically erases the entire alternate dictionary. Then the second | ALT returns to the primary dictionary, whose pointers have been untouched. Unfortunately, a marker word will also roll back the RAM dictionary pointer, DP. And we don't want to do that, because we've certainly created RAM data structures while we've been using the assembler and writing I/O words. So before DISCARD we must save the RAM DP on the stack, and after DISCARD we restore it.

Load Sequence

First load altdict.fth. Then load asm430.fth. The "cleanup" sequence should be added into your application file, at the point where the assembler and equates are no longer needed.

Future Work

The alternate dictionary is a somewhat inelegant solution, constrained by the capabilities of the "standard" CamelForth kernel. One simple kernel change that would simplify this would be to allow two word lists in the search order, along the lines of fig-Forth's CURRENT and CONTEXT. That would eliminate the need for a lot of search order swapping.

But for now, this has allowed me to add assembler code to my application, with only a small overhead in the final code. (To be precise, the words ADP ALATEST | ALT PRIMARY CODE and PROC.)

Links

http://www.camelforth.com/news.php

Listings

MSP430 resident assembler

```
1   ( MSP430 resident assembler )
2   ( B. Rodriguez  17 nov 2012 )
3   ( Note: assumes a 16-bit cell size )
4
5   ( Register/address mode storage
6   ( w---rrrrdissrrrr  where
7   (   rrrr = register # [appears twice]
8   (   d    = destination address mode, 0=Rn, 1=x[Rn] , 0 if illegal
9   (   i    = 1 if illegal destination mode
10  (   ss   = source address mode, 00=Rn, 01=x[Rn], 10=@Rn, 11=@Rn+
11  ( if w=1 then an index word follows, first src, then dst
12
13  ( Example:
14  ( 0404 CONSTANT R4   8494 CONSTANT (R4)
15  ( 0464 CONSTANT @R4  0474 CONSTANT @R4+
16
17  PRIMARY ALT |   ( put assembler in alternate dictionary )
18  HEX
19
20  0000 CONSTANT PC                                    8070 CONSTANT #
21  0101 CONSTANT SP
22  0202 CONSTANT SR  8292 CONSTANT &     0262 CONSTANT 4#   0272 CONSTANT 8#
23  0343 CONSTANT 0#  0353 CONSTANT 1#    0363 CONSTANT 2#   0373 CONSTANT -1#
24
25  0404 CONSTANT R4   0505 CONSTANT R5   0606 CONSTANT R6   0707 CONSTANT R7
26  0808 CONSTANT R8   0909 CONSTANT R9   0A0A CONSTANT R10  0B0B CONSTANT R11
27  0C0C CONSTANT R12  0D0D CONSTANT R13  0E0E CONSTANT R14  0F0F CONSTANT R15
28
29  ( Addressing mode modifiers for PC, SP, R4-R16 )
30  : )   8090 + ;
31  : 0)  0060 + ;  ( because the @ symbol can't be redefined! )
32  : @+  0070 + ;
33
```

```
34
35  ( 2OP format:  src dst opcode )
36  : INDEXED? ( mode -- mode f )   DUP ( 0<) 8000 AND ;
37  : DESTOK? ( mode -- mode )   DUP 40 AND ABORT" Illegal dest.adrsg.mode" ;
38  : ,2OP ( rd rs opc -- )   SWAP 0F30 AND OR  SWAP 8F AND OR I, ;
39  : 2OP   <BUILDS I, DOES>    ( ? rs ? rd -- )
40      @ >R    ( save opcode )
41      DESTOK?
42      INDEXED? IF ( ? rs xd rd )
43          ROT INDEXED? IF ( xs xd rd rs ) R> ,2OP SWAP I, I,
44          ELSE ( xd rd rs )               R> ,2OP I,
45          THEN
46      ELSE ( ? rs rd )
47          SWAP INDEXED? IF ( xs rd rs )   R> ,2OP I,
48          ELSE ( rd rs )                  R> ,2OP
49          THEN
50      THEN
51  ;
52
53  4000 2OP MOV,    4040 2OP MOV.B,
54  5000 2OP ADD,    5040 2OP ADD.B,
55  6000 2OP ADDC,   6040 2OP ADDC.B,
56  7000 2OP SUBC,   7040 2OP SUBC.B,
57  8000 2OP SUB,    8040 2OP SUB.B,
58  9000 2OP CMP,    9040 2OP CMP.B,
59  A000 2OP DADD,   A040 2OP DADD.B,
60  B000 2OP BIT,    B040 2OP BIT.B,
61  C000 2OP BIC,    C040 2OP BIC.B,
62  D000 2OP BIS,    D040 2OP BIS.B,
63  E000 2OP XOR,    E040 2OP XOR.B,
64  F000 2OP AND,    F040 2OP AND.B,
65
66  ( 1op format: src opcode )
67  : ,1OP ( rs opc -- )   SWAP 3F AND OR I, ;
68  : 1OP   <BUILDS I, DOES>   ( ? rs -- )
69      @ >R    ( save opcode )
70      INDEXED? IF ( xs rs -- )   R> ,1OP I,
71      ELSE ( rs -- )             R> ,1OP
72      THEN
73  ;
74
75  1000 1OP RRC,    1040 1OP RRC.B,
76  1080 1OP SWPB,
77  1100 1OP RRA,    1140 1OP RRA.B,
78  1180 1OP SXT,
79  1200 1OP PUSH,   1240 1OP PUSH.B,
80  1280 1OP CALL,
81
82  : 0OP   <BUILDS I, DOES> @ I, ;
83
84  1300 0OP RETI,
85  4130 0OP RET,    ( emulated instruction: SP @+ PC MOV, )
86
87  : !JMP ( dstadr jmpadr opc -- )
88      OVER 2SWAP  ( -- opc jmpadr dstadr jmpadr )
89      2 + - 2/     ( compute jump offset  -- opc jmpadr ofs )
90      DUP -1FF 1FF WITHIN 0= ABORT" Jump offset out of range."
91      3FF AND  ROT OR SWAP I!
92  ;
93  : ,JMP ( dstadr opc -- )   SWAP IHERE ROT !JMP ( CELL) 2 IALLOT ;
94  : JMPOP   <BUILDS I, DOES> @ ,JMP ;
95
96  2000 JMPOP JNE,   2000 JMPOP JNZ,    2000 CONSTANT Z
97  2400 JMPOP JEQ,   2400 JMPOP JZ,     2400 CONSTANT NZ
98  2800 JMPOP JNC,   2800 JMPOP JLO,    2800 CONSTANT C   2800 CONSTANT HS
99  2C00 JMPOP JC,    2C00 JMPOP JHS,    2C00 CONSTANT NC  2C00 CONSTANT LO
100 3000 JMPOP JN,                       3000 CONSTANT P
101 3400 JMPOP JGE,                      3400 CONSTANT L
102 3800 JMPOP JL,                       3800 CONSTANT GE
103 3C00 JMPOP JMP,                      3C00 CONSTANT NEVER
104
105 : IF, ( cond -- cond adr )   IHERE ( CELL) 2 IALLOT ;
106 : THEN, ( cond adr -- )   IHERE SWAP ROT !JMP ;
107 : ELSE, ( cond adr -- cond' adr' )   NEVER IF, 2SWAP THEN, ;
108
109 : BEGIN, ( -- adr )   IHERE ;
```

```
: UNTIL, ( adr cond -- )  ,JMP ;
: WHILE, ( adr1 cond -- adr1 cond adr2 )  IF, ;
: REPEAT, ( adr1 cond adr2 -- )   ROT JMP,  THEN, ;

( extensions for CamelForth ITC model )

0101 CONSTANT RSP
0404 CONSTANT PSP  0505 CONSTANT IP   0606 CONSTANT W    0707 CONSTANT TOS
0808 CONSTANT INDEX 0909 CONSTANT LIMIT 0A0A CONSTANT X  0B0B CONSTANT Y
0C0C CONSTANT Q    0D0D CONSTANT T

: NEXT,  IP @+ W MOV,  W @+ PC MOV,  ;

( these words are for use in alternate dictionary )

: EQU ( n -- )   <BUILDS I, IMMEDIATE DOES> I@ POSTPONE LITERAL ;
: END-CODE   POSTPONE | ;

PRIMARY   ( these words go in primary dictionary )

: CODE   HEADER  IHERE CELL+ ,CF  POSTPONE | ;
: PROC   HEADER  [ ' UINIT @ ] LITERAL ( DOROM) ,CF  POSTPONE | ;

( alternate 1778 BYTES as compiled on 17 aug 2013 )
( ADP before = D412, after = DB04 )
```

Alternate Dictionary support for MSP430 CamelForth

```
( Alternate Dictionary support for MSP430 CamelForth

( Alternate DP and LATEST
( Space is reserved in the MSP430 CamelForth kernel for user variables
( 24, 26, 28, and 30.  We use the first two of these here.

DECIMAL
26 USER ADP         ( alternate dictionary pointer
28 USER ALATEST     ( alternate "latest" link

( Words to swap dictionary order
( Note: these must be used with care, and normally in pairs

: | ( -- )   ( swap search order )
   LATEST @ ALATEST @ LATEST ! ALATEST !
; IMMEDIATE     ( so we can use it inside colon defs )

: ALT ( -- )   ( swap ROM dictionary pointers )
   IDP @ ADP @ IDP ! ADP ! ;

( In case we lose track of the pointers:
: PRIMARY ( -- )   ( ensure lower-address pointers are active )
    LATEST @ ALATEST @ 2DUP U> IF  LATEST ! ALATEST !  ELSE 2DROP THEN
    IDP @ ADP @  2DUP U> IF  IDP ! ADP !  ELSE 2DROP THEN ;

LATEST @ ALATEST !  ( ensure that these words appear in both search orders )
HEX D400 ADP !      ( start alternate dictionary at D400 )
( ADP must be on a 512-byte boundary. Note that Forth kernel starts at E000 )

( Typing DISCARD with the alternate dictionary & search selected will erase the
( alternate dictionary space, and return the alternate pointers to their
( original state

| ALT
MARKER DISCARD
| ALT

( Since MARKER erases Flash from the saved location to IHERE, it is safe to use
( in both the primary and alternate dictionary. HOWEVER, both DP and LATEST must
( be in the same space -- primary or alternate -- when MARKER is executed!

( For an assembler in alternate space, CODE should be defined in the primary
( space so that new CODE words are added to that dictionary.
( CODE and END-CODE may switch search order.
```

Waduzitdo in Forth

Jürgen Pintaske (Sammlung), Dirk Brühl (Forth) und Michael Kalus (Übersetzung)

frei nach Brian Connors, 2001

Am Anfang war der Personal Computer. Und der war gut, wenn auch nicht viel mehr als ein Spielzeug mit Suchtpotenzial, wenn man es genauer betrachtete. Und dann war da dieser Kerl namens Larry. Der hatte Freunde, die ihn immer fragten: "Und, was kann der?" Das ist heute[1] eine einfach zu beantwortende Frage, aber 1978, als man da nur diese Kisten mit blinkenden Lichtern sah, und die Freunde dachten, ein Komputer sei doch etwas, das ein Zimmer ausfüllt, und die kommunalen Steuern hochtreibt, da war das eine sehr lästige Frage. Und obendrein schauten die Freunde ja auch skeptisch: "Waaaaas? Wieviel hast du dafür hingelegt!?" In dieser Lage beschloss dieser Kerl, also Larry, etwas mit der Kiste zu machen, das jeder gleich versteht, und erfand eine Computersprache für die, die keine Ahnung von Computern hatten. Er nannte sie "Waduzitdo"[2] und schrieb lustige kleine Quizprogramme für seine Freunde damit.

Eine Sprache für die, die keine Ahnung von Computern haben.

Diese Sprache war sowas wie PILOT[3], nur viel kleiner, und stammte womöglich davon ab, also von dieser ziemlich hässlichen Mischung aus BASIC und Assembler, die wegen beträchtlicher Unverständlichkeit längst versunken ist, die es aber bestimmt noch in einem Retrocomputing-Museum gibt. Sie ist nicht Turingvollständig und ohne jeden brauchbaren Standard. Und obwohl es Konditionale und Sprung-Anweisungen gibt, macht man eigentlich nichts mit den Daten, die man da rein füttert. Soviel zum Waduzitdo.

Waduzitdo in Forth, Level 0

Diese Fassung ist, wie die originale Sprache[4] auch, nachträglich dokumentiert worden von irgendeinem Professor aus irgendwo. Wir haben das eingesammelt und geben das hier wieder. Die Sprache selbst ist, wie gesagt, PILOT, einem einfachen Assembler sehr ähnlich, hat Opcodes und Datenfelder. Und so geht's:

Grundkonzept

Offensichtlich haben wir nun viel mehr Speicherplatz zum Spielen frei als Larry 1978, daher geht es etwas anders zu. Wir könnten so nett sein und das Ganze stapelbasiert anlegen, aber das wäre zu schön, zu elegant und zu einfach. Deshalb machen wir etliche Register die in Variablen gehalten werden, als virtuelle Maschine über der virtuellen Forth-Maschine sozusagen: Einen Akkumulator für die Eingabe und ein Übereinstimmungs-Register - `INPUT` und `MATCH`. Der Akkumulator hält einen Wert, das ein Zeichen ist, und das andere Register hält auch einen Wert, und der ist entweder *wahr* oder *falsch*. Und noch so ein paar Register - siehe Quellcode. Damit wäre die Rückwartskompatibilität zu den alten Puzzels von Larry hergestellt.

Syntax

Alle Anweisungen haben die Form

```
modifier opcode: data
```

Data können auch leer sein, wenn es nichts einzugeben gibt. Der Doppelpunkt ist Pflicht. Aber es ist verboten, das Semikolon zu benutzen. "Ja, aber..." wendet ihr ein. Nein! Es ist mir wurscht, dass ihr das möchtet, ihr dürft nicht! Punkt.

Opcodes

T	type – Zeigt Data auf der Konsole an.
A	accept – Akzeptiert eine Antwort, und schiebt die in den Akkumulator.
M	match – Macht den Vergleich zwischen Data und Akku und setzt das Übereinstimmungs-Register auf *wahr*, wenn Überreinstimmung herrscht, sonst auf *falsch*.
J	jump – Wenn Data 0 ist oder nicht da, springe zurück zur letzten accept-Anweisung. Sonst springe vorwärts entsprechend dem **label*.
S	stop – Stop das Programm.

Modifiers

Die sind eigentlich sowas wie ein Teil des Opcodes. Sie sagen dem Interpreter, was die nachfolgende Zeile bedeutet.

[1] 2015 - für die, die das noch nach der Apokalypse lesen sollten.

[2] engl.: "What does it do?" - Na? Klickts? ;-)

[3] Programmed Instruction, Learning, or Teaching (PILOT)

[4] 'WADUZITDO: How To Write a Language in 256 Words or Less' has been the title of an article from BYTE magazine, September 1978 issue (pp. 166-175), by Larry Kheriarty, describing a very minimalistic language to show off 'what a computer can do', intended to catch attention from even a complete computer illiterate.

*	label – Markiert eine Zeile, die angesprungen werden kann.
Y	if_yes – Führt die Zeile nur aus, wenn im Übereinstimmungs-Register ein *wahr* steht.
N	if_no – Führt die Zeile nur aus, wenn im Übereinstimmungs-Register ein *falsch* steht.

Unterschiede zum Original

Waduzitdo in Forth ist natürlich anpassungsfähiger als das Original, weil die Begrenzungen auf eine bestimmte Maschine nicht da sind, es ist also, anders als beim frühen Personal Computer, auf verschiedenen Plattformen lauffähig, bzw einfach anpassbar. Zum Beispiel spezifiziert Waduzido.f die Akkumulator-Variable explizit, die im Original ja inerhalb der CPU steckte, damals also hardwaremäßig vorgegeben war. Äußerlich entspricht es dem Original.

Und es eignet sich so natürlich auch bestens dazu, auf all diese Fragen zu den *heutigen Kistchen mit den blinkenden Lichtern*[5] etwas entgegnen zu können, taugt also für all diese hübschen neuen Entwicklungsboards von MCUs, die gerade massenweise herauskommen, jedenfalls solange sie irgendein Forth drauf haben.

Links

http://waduzitdo.org/

http://www.reocities.com/ResearchTriangle/Station/2266/tarpit/waduzitdo/wdzref.html

http://en.wikipedia.org/wiki/PILOT

Listings

```
1   \ Waduzitdo
2
3   \ Written for win32forth by Dirk Brühl in 2014
4   \ here adopted for gforth by mk 1/2015
5
6
7   \ anew waduzido.f \ <-- do that in win32forth
8
9   : ascii  postpone [char] ;  immediate \ <-- to be compatible with gforth
10
11
12  : -- bye ; \ gforth debug helper
13
14  decimal
15
16   variable EoP    \ End of Program
17   variable Match  \ true or false
18   variable Accpt  \ adress for return jump
19   Variable Input
20   variable Textlength
21   variable YES
22   variable NO
23   variable Command
24   variable End    \ to end the program with <CTRL>+B
25
26  : YES? ( -- flg )  \ checks if command has to be executed
27     YES @ Match @ and
28     NO  @ Match @ 0= and
29     or
30     YES @ NO @ or 0= or
31  ;
32
33  : Type!   ( start -- n )
34          Textlength off
35          EoP @ swap do i c@ dup YES? if emit else drop then 1 Textlength +!
36                        $A = if leave then loop
37          Textlength @
38  ;
39
40  : Accept! ( start -- n )
41     Accpt ! cr
42     Key dup 2 = if -1 End ! drop BL then
43     $DF and dup Input C! emit cr 1
44  ;
45
```

[5] *http://www.reocities.com/connorbd/blinkenlights.html*

```
: Match!  ( start -- n )
   c@ Input c@ =
   Command C@ ascii M =
    if  Match @ else 0 then or Match !
   1
;

: Jump!   ( start -- n )
   YES? if Accpt @ swap - else drop 1 then
;

: Stop!   ( start -- n )
   drop -1
;

: CheckCommand  ( i char -- n )
     dup >r                       \  save Command for Match-Checking
        case
        ascii T of Type!   endof  \  displays data on the console
        ascii A of Accept! endof  \  accepts one line of input into the accumulator
        ascii M of Match!  endof  \  compares data to the accumulator and sets match flag to TRUE if true, FALSE if not tru
        ascii J of Jump!   endof  \  If data is 0 or nonexistent, jump back to the last accept statement.
                                  \  not implemented: Otherwise, jump forward by data * markers.
        ascii S of Stop!   endof  \  Stop program.
        nip 0 swap endcase
     r> Command C!                \  save Command for Match-Checking
;

\ needs slurp.f  \ win32forth needs it, gforth has it.

: Waduzido ( $ len -- )
    Here >R  \ save address for redeeming memory space
    Slurp-file cr cr End off
    bounds over EoP !
           do i c@ ascii : =
                    if   i 2 - c@ ascii Y = if YES on else YES off then
                         i 2 - c@ ascii N = if NO  on else NO  off then
                         i 1+ i 1- c@ CheckCommand dup 0= if 1 then
                    else 1
                    then dup End @ or -1 =  if leave then +loop 2drop
    r> DP !  \ redeem memory space
;
```

```
\ Slurp.f

Decimal

anew Slurp.f


: Slurp-file ( a n -- address len )
        r/o open-file abort" Couldn't open file"  >r
        here                                          \ buffer address
        r@ file-size abort" Couldn't get filesize"    \ length
                     abort" File too large"           \ maximum 64kB
        dup allot                                     \ allocate space
        2dup
        r@ read-file abort" Couldn't read file" drop  \ read  file
        r> close-file abort" Couldn't close file"     \ close file
        ;


: spit-file ( bufferadr u nameadr n -- )
  r/w create-file throw dup ( fid) >r
      write-file  throw r>
      close-file  throw ;

\ s" my.in"  slurp-file  s" my.out" spit-file


\s

```

```
\ Waduzitdo: BIRTHDAY LIST

T:IT IS BIRTHDAY LIST TIME.

T:THE PURPOSE OF THIS PROGRAM IS TO

T:DETERMINE WHAT GIFTS ARE ACCEPTABLE.

T:TYPE THE CODE LETTER ASSOCIATED WITH

T:THE POTENTIAL GIFT IDEA...

T: A HOME APPLIANCE

T: B SOMETHING BORING

T: C ITEM OF CLOTHING

T: D SOMETHING DECORATIVE FOR THE HOUSE

T: G GARBAGE DISPOSAL

T: M MY OWN COMPUTER

A:

M:A

YT:UNACCEPTABLE.

M:B

YT:NO WAY .

M:C

YT:ACCEPTABLE IF NOT UGLY.

M:D

YT:OKAY IF CHOSEN WITH GOOD TASTE

YT:SO AS NOT TO BE TACKY.

M:G

YT:YEAH !

M:M

YT:THE LAST THING IN THE WORLD

YT: I WOULD EVER WANT.

NM:A

NM:B

NM:C

NM:D

NM:G

NM:M

NT:CANT YOU READ FOOL? THAT IS NOT

NT: ONE OF THE CHOICES.

NT:TRY A or B or C or D or G OR M

J:O

S:
```

Abbildung 1: Das Keyboard ist noch immer da...

Was ist das Blinkenlights?

Herr Doctor Otto Von Fraktur-Englisch

Ach, die Dummkopfen are always asking me zat qvestion. Das Blinkenlights ist a Word for ze tings on der front of der computer. Vat? You say you have keine Blinkenlights? Vell, get off your Tuschi und go into ze Netzverkroom so you can see some Blinkenlights!

Back in the olden days zere vere many many lights on ze front of ze Computers. Zey vould flash so zat you could see how schnelle ze Computer vould run, but zese days, ach, you can nicht find a Light auf ze Computer zat vill blink slow enough to see it blink. You must go into ze Netzwerkroom to see Blinkenlights on ze Ethernetkontrolen und Rackmountserveren. So sad...

Aber now zer is ze LaunchPad, and zer are sie wieder, die Blinkenlights. And so can man say again, and for ever und ever this wird true sein:

Das Computermachine ist nicht fuer gefingerpoken und mittengrabben. Ist easy schappen der shpringenverk, blowenfusen unt poppencorken mit spitzensparken. Ist night fuer gewerken bei die Dummkopfen. Das rubbernecken Sichtseeren keepen das cotton-pickenen Hands in die Pockets muss; relaxen und watschen die Blinkenlichten.

Für eine alternative Bedeutung von *Blinkenlights* siehe `https://de.wikipedia.org/wiki/Projekt_Blinkenlights` oder `http://blinkenlights.net`. Welche Rolle spielt Forth dabei?

Neulich beim Stammtisch – Paralleles Rechnen

Martin Bitter

Trotz Terminproblemen fand der Rhein-Ruhr-Stammtisch statt[1] Carsten hatte sich ein P1602-DK02 parallela-Board bestellt und berichtete davon. Michael fragte laut, wann denn paralleles Programmieren Sinn mache. Obwohl ich weder einen Parallelrechner besitze, noch jemals einen von nahem gesehen habe, hub ich an (... wie der Blinde von der Farbe!): Sinn macht es, wenn es einen Zeitvorteil bringt.

Erste theoretische Überlegung

Dazu eignen sich nicht alle Aufgaben, sondern nur solche, deren Teilschritte sich unabhängig voneinander bearbeiten lassen. Ein Beispiel sei das bekannte Apfelmännchen[2]. Dabei sollen letztendlich für jedes Bildschirmpixel Farbwerte berechnet werden. Jedes Pixel steht für ein Koordinatenpaar in der Mandelbrotmenge, eine Kombination aus Real- und Imaginärzahlen, also komplexe Zahlen, und kann unabhängig von den Werten seines Nachbarn berechnet werden. Abhängig von der eingestellten Rechentiefe und den jeweiligen Koordinaten wird ein und dieselbe Berechnung wiederholt durchgeführt, bis eine Abbruchbedingung erreicht wird. Das Ergebnis einer Berechnung geht in die Eingangsparameter der jeweils nächsten Berechnung ein. Mit einem herkömmlichen seriell arbeitenden Rechner wird ein Pixel nach dem anderen berechnet. Bei einer Bildgröße von 1280x1024 müssen die Werte für 1.310.720 Pixel berechnet werden. Das ist sehr rechenintensiv und führt zu langen Berechnungszeiten. Bei einem 'idealen' parallel rechnenden Computer berechnet jeder Knoten nur die Werte für ein Pixel, so dass das gesamte Bild in einer wesentlich kürzeren Zeit fertig ist. Im Idealfall - jede Berechnung benötigt die gleiche Zeit - wäre der Parallelrechner knapp 1,3 Millionen mal schneller fertig. Das ist doch was!

Praktisch gesehen

In der wirklichen Welt ist es nicht ganz so schön (schnell).

Zum einen sind nicht alle Rechnungen in der gleichen Zeitdauer beendet. Beim Apfelmännchen gibt es Koordinaten (Pixel), deren Werte in nur einem Durchgang berechnet werden können, andere benötigen die jeweils vorgegebene Maximalzeit. Das Apfelmännchenbild ist fertig, wenn das letzte Pixel berechnet ist. Bei einer eingestellten Rechentiefe von z.B. 50 Maximaldurchgängen pro Pixel wird also auf das 'langsamste' Ergebnis gewartet. Leider lässt sich kaum vorhersehen, welches Pixel ein 'schnelles' und welches ein 'langsames' Pixel ist. Worstcase: Wenn, was zwar möglich, aber sehr unwahrscheinlich ist, von den knapp 1,3 Millionen Pixeln alle bis auf eines in einem Durchgang berechnet wären, und dieses eine Pixel 50 Durchgänge bräuchte, so betrüge die parallele Rechenzeit 50 Durchläufe, während ein serieller Rechner 1,3 Millionen + 50 Durchläufe benötigte. Dies drückt die 'parallele' Geschwindigkeit etwas. Bestcase: Alle Pixel benötigen 50 Durchläufe. Das führt zu einer 'seriellen' Berechnungszeit von 50 * 1,3 Millionen Durchgängen und zu einer parallelen Berechnungszeit von immer noch 50 Durchgängen. Hier ist der Vorteil des Parallelrechners am größten.

Abbildung 1: Wir trafen uns übrigens im Unperfekthaus in Essen ...

Zum anderen ist ein Parallelrechner mit knapp 1,3 Millionen Rechenknoten nicht ganz billig und wird deshalb eher für ernsthafte Berechnungen und nicht für das Apfelmännchen eingesetzt. Übliche Rechner haben deutlich weniger Rechenknoten. Das oben erwähnte parallela-Board gibt es in einer 16- und einer 64-Knoten Version. Damit kann man die geforderte Rechenzeit auf ein 1/16 bzw. 1/64 kürzen.

Nun kommen aber Verteilungsprobleme ins Spiel: Teilt man beim Apfelmännchen den Bildschirm in 16 (64) Kacheln auf, die parallel berechnet werden? Oder ist es besser, jedes 16te Pixel einem Rechenknoten zuzuweisen? Letzteres verteilt in der Regel die Rechenlast besser, weil so eine Anhäufung von 'langwierigen' Pixeln gleichmäßiger verteilt wird. Die Frage ist, wie verteilt man die Rechenlast so, dass die Anzahl der Durchläufe für jeden Rechenknoten annähernd gleich ist? Was geschieht mit Rechenknoten, die schon fertig sind? Sollen die anderen Knoten fremde Rechnungen (Pixel) bekommen, wenn sie schon fertig sind? Wie entscheidet man das? Gibt es dafür einen Algorithmus? Wieviel Overhead handelt man sich mit der Bearbeitung dieser Fragen ein? Bei Parallelrechnern gibt es für diese Verwaltungsaufgaben oft spezielle Prozessoren. Die gleichen Anforderungen ergeben

[1] Termine: unter www.forth-ev.de in die Suchmaschine „ruhrpott“ eingeben.

[2] Mandelbrotmenge; Algorithmus und Forthprogramme dazu werden hier als bekannt vorausgesetzt

[3] beowulf, wulfware

sich, will man 'verteiltes' Rechnen verwirklichen. In diesem Fall kommen noch Probleme des Netztransfers hinzu[3].

Michael war mit dieser Erklärung zufrieden und meinte abschließend: Schreib's für die Vierte Dimension auf! Das ist hiermit geschehen. Ein Schmankerl gibt es noch: Carsten wird zur Tagung 2015 sein parallela-Board mitbringen. Das hat 16 Rechenknoten (RISC) und eine FPGA für die Verwaltungsprozeduren. Wenn sich Interessenten mit Vorwissen finden, kann ein Workshop dazu stattfinden. Ein mögliches Ziel: Apfelmännchen! Mich würde es freuen.

Im Nachgang zum Stammtisch per email erinnerte Bernd Paysan daran, dass heutzutage in den meisten Rechnern Mehrkern-CPUs sitzen, und das fast alle *GPUs massive Parallelrechnungen* ausführen - wenn nicht in deinem PC, dann sicherlich in dem deines Sohnes! (Oder dem der Tochter natürlich; der Sätza).

Links

https://www.parallella.org/board/

http://www.phy.duke.edu/~rgb/Beowulf/wulfware.php

http://nuclear.mutantstargoat.com/articles/sdr_fract/

AmForth auf dem TI MSP430

Matthias Trute

Amforth ist entstanden, um auf den Atmegas der Firma Atmel zu laufen. Andere Microcontroller waren nie ein Thema. Es gab und gibt schließlich genug andere Forths für diese, da gibt es keinen Grund in fremden Gefilden zu wildern.

Wer ist schuld?

Warum ich doch anfing, mich für andere Microcontroller zu interessieren, ist eindeutig die Schuld des Forth e.V. Matthias Koch hatte sein Mecrisp für den TI MSP430 und, später, einige ARMs vorgestellt und hoffte auf Mitentwickler. Michael Kalus griff in die Kiste und so fanden passende Boards den Weg auf meinem Basteltisch. Das Launchpad war dankenswerterweise mit einem Forth vorinstalliert: 4€4TH, ein Camelforth–Abkömmling. Da war der Start einfach: USB Port einstöpseln und das passende Terminal starten. Erster Eindruck: Rasend schnell. Zweiter Eindruck: Irgendwie komisch. Wie die Götter schon vom Olymp herabriefen „Kennst Du ein Forth, kennst Du ... ein Forth".

```
4E4th R0.34 Apr 28 2012|110001110 Wiping
words S2? S2 GREEN ....  ok
dp . 522 ok
```

Was man hier nicht sieht, sind seltsame Zeichen am Ende einer jeden Zeile. Software Flowcontrol mit XON/XOFF, ok, das war einfach. Befehle wie `: test 0 do i . loop ;` funktionierten, `: test 0 ?do i . loop ;` erst mal nicht. Dass die Programmausgabe direkt nach dem letzten Zeichen der Eingabe anfängt, verwirrt mich immer noch. Ich bin Linux-Nutzer (neben anderen Unixen), da habe ich so einige Erwartungen an einen Kommandoprompt. Jedenfalls mehr als dass der Cursor blinkt.

```
amforth 5.6 MSP430G2553 8000 kHz
Dez 13 2014 15:08:07 master
> words
2literal s>d spaces space
... ok
> dp .
2895  ok
>
```

Es gab keine großen Dinge, die störten, sondern nur ein paar Kleinigkeiten. Aber ich wollte ja mecrisp ausprobieren. Die fertigen Hexfiles auf den Controller zu zaubern, ist wie bei amforth: obskures Programm installieren (mspdebug bzw lm4flash), eine noch obskurere Kommandozeile füttern (mspdebug rf2500 "prog forth-mecrisp-2553-with-basisdefinitions-launchpad.hex") und schon war es da, das neue Forth. Nun ja, fast.

```
Mecrisp 1.1 for G2553 by Matthias Koch
                                        ok.
                                            ok.
                                                ok.
```

Offensichtlich ist das Zeilenende anders, Matthias ist wohl Mac–User. Warum nur wollen alle Forths subtil unterschiedliche Terminaleinstellungen haben: Zeilenende mal CRLF, mal nur LF (mecrisp). Oder XON/XOFF Flowcontrol, mal ja (4€4th) mal nein (amforth).

Sowas sollte trivial anpassbar sein, da die Sourcen ja verfügbar sind. Die Toolchain für alle Systeme ist, sagen wir, spröde. Aber nicht gemeckert sondern geklotzt. Die erste Hürde, die Hexfiles auf die Controller zu transferieren, ist ja schon genommen, also folgt Schritt zwei: Die Quellen, so wie sie sind, einmal selbst compilieren.

Die MSP430 Quellen von Matthias Koch wurden mit „Alfred Arnolds *great* macro assembler AS" übersetzt, den ich nicht zum Laufen bekam. Schade. Die Camelforth/4€4th Quellen erforderten eine Installation der Texas Instruments IDE unter Windows, zu der Zeit war meine Windows VM gerade mit Plattenplatzproblemen befrachtet und für so ein riesiges Paket nicht benutzbar, der Weg war versperrt. Die ARM Quellen nutzen die GCC Toolchain, davon gibt es für die ARM's nur wenige Dutzend. Hier war eine namens `gcc-arm-embedded` aus einem sepraten Repository erforderlich. Sowas mag

ich eher nicht, das ist spätestens beim nächsten Update nicht mehr benutzbar.

Also noch etwas mit den Systemen gespielt, wie sie sind. Triviale Programme liefen. Ein Test mit komplexeren Quellen scheiterte, zum Teil schon beim Upload. Beim Amforth habe ich ein relativ komplexes Tool, dass mir Forth– Quelltexte nach Bedarf zusammenstellen kann und zum Controller schickt. Das funktioniert aber nicht mit anderen Systemen. Weder der Multitasker funktionierte, noch die I2C Bibliothek. Am Ende habe ich ein eher schmales Fazit gezogen: Die neuen Controller und ihre Forth' sind toll, können aber nichts, was Amforth und Atmegas nicht auch konnten und so wirklich beitragen zum Mecrisp konnte ich auch nichts, da ich keine Ahnung vom MSP430 hatte. Die ARM Variante war mir ohnehin zu groß, ARM hiess für mich, dass da Linux drauf läuft und damit gforth. Dass das falsch ist, habe ich erst viel später gelernt.

Darüber gingen ein oder zwei Jahre hin, in denen die Platinchen in der Kiste lagen und warteten.

Im Frühjar 2014 sind dann zwei Dinge geschehen, die den Stein wieder ins Rollen bringen sollten. Zum einen habe ich einen Assembler namens `naken_asm` entdeckt, der, neben anderen, auch Atmel Atmegas unterstützt. Da mich die umständliche Nutzung des Atmel Assemblers noch nie so recht begeisterte, fing ich an, den `naken_asm` auszuprobieren. Zu meinem Bedauern stellt Amforth wohl doch sehr hohe Anforderungen und die Atmegas sind wohl doch ungewöhnlicher als gedacht, jedenfalls war das Experiment nicht von Erfolg gekrönt.

Nebenbei habe ich in der Beispielliste des `naken_asm` Camelforth von Brad Rodriguez (wieder–) gefunden. Das kannte ich noch von den 4€4th Experimenten und aus der Portierung der 1–wire Anbindung zu Amforth. Also habe ich die alten Platinen wieder herausgesucht und Camelforth installiert, diesmal komplett aus den Quellen. Da das auf Anhieb funktionierte, fing ich an, die Quellen zu lesen. Dabei tat sich eine faustdicke Überraschung auf: Camelforth ist wie Amforth ein Indirect Threaded Forth und die Quellen lesen sich exakt genauso. Da passt kein Blatt Papier dazwischen.

Zum Vergleich zuerst eine (ältere) AVR Variante

```
VE_CHAR:
    .dw $ff04
    .db "char"
    .dw VE_HEAD
    .set VE_HEAD = VE_CHAR
XT_CHAR:
    .dw DO_COLON
PFA_CHAR:
    .dw XT_BL
    .dw XT_WORD
    .dw XT_COUNT
    .dw XT_DROP
    .dw XT_CFETCH
    .dw XT_EXIT
```

und jetzt die Camelforth Fassung

```
HEADER(CHARR,4,"char",DOCOLON)
    DW BLANK
    DW WORDD
    DW ONEPLUS
    DW CFETCH
    DW EXIT
```

Die Unterschiede sind augenscheinlich, andererseits auch wieder praktisch irrelevant. Die Label sind es, die unterschiedlich sind. Und dass `COUNT DROP` das gleiche wie `1+` ist, dürfte keinen überraschen.

Zwei Forth's, die soviel Gemeinsamkeiten aufwiesen, da musste doch was gehen. Wie wäre es, ein Forth zu haben, das mit identischem Sourcecode auf verschiedenen Controllertypen läuft? Mit C kein Problem, nutze ich aber nicht. Also musste ein Weg gefunden werden, den beiden Assemblern den gleichen Quelltext schmackhaft zu machen. Das ging recht schnell mit überschaubaren ifdef's und passenden Makros. Daneben mussten natürlich die Label angepasst werden. So wurde aus `CHARR` ein `XT_CHAR`, mehr zu tippen, hat dafür jedoch keine Kollisionen mit den Schlüsselworten des Assemblers.

```
.if cpu_msp430==1
    HEADER(XT_CHAR,4,"char",DOCOLON)
.endif

.if cpu_avr8==1
VE_CHAR:
    .dw $ff04
    .db "char"
    .dw VE_HEAD
    .set VE_HEAD = VE_CHAR
XT_CHAR:
    .dw DO_COLON
PFA_CHAR:
.endif
    .dw XT_BL
    .dw XT_WORD
    .dw XT_COUNT
    .dw XT_DROP
    .dw XT_CFETCH
    .dw XT_EXIT
```

Leider ist der Atmels AVR Assembler nicht smart genug, um Makros wie `HEADER` bereitzustellen, damit wäre der Quellcode noch einfacher.

Der obige Assemblertext kann sicher ohne Probleme als Forthcode gelesen werden:

```
: char ( addr -- c )
  bl word count drop c@ ;
```

Viele Kleine Schritte

Nachdem die Vorabtests hinreichend erfolgreich waren, ging ich die Projekt generalstabsmäßig an. Da Camelforth unter GPLv3 veröffentlicht wurde und die GPLv2, unter der Amforth stand, nicht dazu passt, musste Amforth ebenfalls unter GPLv3 gestellt werden. Dabei kam

es leider zu einer einigermaßen hässlichen Auseiandersetzung mit einem Amforth Nutzer. Er nutzt Amforth, um in seinem Brot–und–Butter–Geschäft für einen Kunden ein System zu entwickeln. Details hat er nie genannt. Den Wechsel der Lizenz wollte er nicht akzeptieren und protestierte sehr energisch, da er (zusätzliche) Probleme befürchtet.

Die Diskussion kreiste am Ende um zwei Dinge: Zum einen, ob sein bisheriges Vorgehen überhaupt mit der GPL konform geht (es gibt da Zweifel) und ob die GPL–Versionen v2 und v3 da Unterschiede haben. Zum anderen ging es um das Schicksal seiner Beiträge zu Amforth. Es waren nur wenige, es wäre aber schade, auf sie zu verzichten. Am Ende hat er einen Fork gestartet und zugleich erlaubt, seine bisherigen Beiträge auch unter der neuen Lizenz im Amforth zu belassen. Die Diskussionen haben einiges an Zeit und Mühe gekostet, sowas macht wahrlich keinen Spaß.

Nachdem die juristischen Probleme gelöst waren, war wieder Zeit für die erfreulichen Seiten des Lebens. Die Zusammenführung von zwei etwa gleich großen Quellsystemen ist ein ziemliches Unterfangen. Dabei kann man sehr viel falsch machen ud hat am Ende einen Haufen Gordischer Knoten.

Aus diesem Grund habe ich mich entschlossen, die beiden Quelltexte zunächst nur in einem gemeinsamen Verzeichnisbaum zusammen zu kopieren und den Buildprozess zu vereinheitlichen. Später dann sollten Schritt für Schritt die Gemeinsamkeiten zusammenwachsen. Dabei sollte jede Änderung so klein sein, dass sie nie die Lauffähigkeit der beiden Systeme beeinträchtigen konnte. Endziel war ein System, bei dem die Highlevel–Worte aus der gleichen Quelle stammen sollten, lediglich ein paar wenige Worte sollten spezifisch für einen Controller existieren.

Die Quellen vom Camelforth habe ich außerdem weitgehend an den Stil von Amforth angepasst. Jede Wortdefinition kam in eine eigene Datei und die werden über Include-Listen zu einem Ganzen zusammengesetzt. Daneben wurde das Build-System so erweitert, dass es mit den MSP430 typischen Programmen umgehen kann. Da ich hierfür ANT benutze, sind einige XML Files involviert:

```
appl/
  launchpad430/
     build.xml (includes ../msp-build.xml)
  arduino/
     build.xml (includes ../avr8-build.xml)
  common-build.xml
  msp430-build.xml
  avr8-build.xml
avr8/
  words/
  dict/
  Atmel/
  decives/
msp430/
  words/
  TI/
  devices/
common/
  words/
lib/
```

Das File `build.xml` des Launchpad430 sieht hierbei wie folgt aus:

```
<project name="launchpad430"
     basedir="."
     default="Help">
 <import file="../msp430-build.xml"/>
 <target name="launchpad430.hex"
  depends="git-info, build-info">
  <nakenasm
   projectname="launchpad430"
   mcu="msp430g2553"/>
 </target>
 <target name="launchpad430"
   depends="launchpad430.hex">
   <mspdebug
     mcu="msp430g2553"
     projectname="launchpad430"/>
    </target>
</project>
```

Freunde weniger innvovativer Tools finden ein Makefile, das in etwa das Gleiche macht. Man muss nur auf ein paar Zusatzinformationen im generiertem System verzichten, wie etwa den Zeitpunkt des Assemblierens oder den Namen der Quelltextversion (master oder trunk). Diese Angaben sind eigentlich nur für Nutzer interessant, die eine feinere Angabe als die Versionsnummer benötigen.

Bis hierher wurde noch kein Code geändert, die beiden Forth's waren komplett unabhängig. Einzig die Übersetzung erfolgte in einem einheitlichen Rahmen. Damit konnte der absehbar lange Weg der Harmonisierung anfangen. Den Anfang machten Trivialworte wie `2SWAP` und `CR`. Mit zunehmender Erfahrung kamen dann anspruchsvollere Dinge wie die Compilerworte von `AHEAD` bis `WHILE` hinzu. Diese waren insofern anspruchsvoll, als dass sie intern auf Worten wie `<MARK` und `<RESOLVE` aufsetzen, die ihrerseits für jedes Forth-Kernsystem anders aussehen. Bei jeder Änderung wurde geprüft, ob die beiden Systeme weiterhin funktionsfähig blieben. Selbst bei `CR`.

Der große Vorteil dieser Strategie ist, dass man nahezu immer funktionsfähige Systeme hat. Bugs können nur durch die jeweils aktuellen Änderungen entstanden sein.

Nachteil ist, dass man oft temporäre Implementierungen hat, die man am Ende doch entfernt. Auch gab es manche Dopplungen wie `FIND` und `FIND-NAME`, die lediglich unterschiedliche Stringformate benutzen. Das löste sich erst, als alle Worte, die sich mit *counted strings* beschäftigen, aus dem Assemblerquelltext entfernt werden konnten. Wer sie benötigt, kann die Definitionen aus Forth–Quellen natürlich nachladen.

Das Beste vom Brot

Wenn zwei Forthsysteme zusammenkommen, bleibt es nicht aus, dass es Dinge gibt, die das eine *so*, das andere komplett *anders* macht. So auch Amforth und Camelforth. Waren `IF` und Co noch einfach zu vereinen, gab es bei `DO` und `LOOP` klar unterschiedliche Ansätze. Dies äußerte sich darin, wie die Sprünge von `LEAVE` und `?DO` zum Schleifenende organisiert wurden. Bei Camelforth werden diese Sprünge über einen separaten Stack zur Compilezeit gesammelt und auch dann aufgelöst. Amforth hat den Sprung immer erwartet und zur Laufzeit die Adresse zusätzlich mit auf den Returnstack gelegt.

Diese beiden Ansätze konnten nicht vereint werden, hier galt es eine Entscheidung zu treffen: Welche Strategie sollte das zukünftige System nutzen? Am Ende gewann die Camelforth–Lösung. Im AVR Code wurde enstprechend geändert und ein Platz für den `LEAVE`–Stack gefunden: Zwischen Datenstack und Returnstack. Am Ende gab es sogar noch ein kleines Bonbon, da auf eine spezielle Runtime für `?DO` verzichtet werden konnte. Das hatte aber eher damit zu tun, dass ich den Code nach vielen Jahren überhaupt wieder näher angeschaut hatte.

Amforth macht nun nicht alles schlechter, so gibt es auch viele Dinge, die beibehalten werden. Für den Nutzer am auffälligsten ist das Verhalten des Interpreters. Das ist beim MSP430 nun genauso aus wie beim AVR. Kein Wunder, die zugrundeliegenden Worte `ACCEPT` und `QUIT` sind gemeinsamer Code. Damit ist auch die komfortable Amforth–Shell für alle Controller gleichermaßen einsetzbar: Registernamen, Kommandohistorie, File–Includes, Flowcontrol zum Controller etc pp.

Am Ende blieben von Camelforth viele Assemblerworte und auch sonstig nahe am Controller hängende Befehle. Im Allgemeinen ist Amforth auf dem MSP430 ein Forth geworden, wie es auf dem AVR schon war und ist. Camelforth ist faktisch unsichtbar geworden.

```
amforth 5.6 MSP430G2553 8000 kHz
Dez 13 2014 15:08:07 master
> $32 #32 + .
82  ok
> environment show-wordlist
version forth-name cpu  ok
>

amforth 5.6 ATmega328P 12000 kHz
Dez 13 2014 15:01:17 master
> $32 #32 + .
82  ok
> environment show-wordlist
/user mcu-info cpu version
forth-name /hold /pad wordlists  ok
>
```

Unterschiede

Die beiden Controllertypen sind schon sehr unterschiedlich. So bewirkt die 16–bit Architektur des MSP430, dass der Programmcode für die Assemblerworte deutlich kürzer ist. Das ist gerade beim Basissystem auffällig. So belegt der MSP430 Kern weniger als 8KB Programmspeicher, wo der entsprechende AVR Code knapp drüber liegt.

Daneben hat der MSP430 eine andere Speicherarchitektur. Da liegt das Dictionary im gleichen Adressraum wie das RAM und Worte wie `(I)TYPE` müssen keine Rücksicht mehr nemen, wo die Daten nun liegen. Ebenso kann man in RAM compilieren, was gerade für Textzwecke unwahrscheinlich viel schneller geht. Nur hat der MSP430 auf dem Launchpad gerade mal 512 Bytes RAM, da hat jeder Atmega von Rang ein Vielfaches (bis zu 16KB). Da geht vieles einfach nicht, etwa ein BLOCK Handling.

Interessant ist, dass der MSP430 immer relative Sprünge absolviert, eine Eigenheit, die die Forth–VM übernommen hat und auch die Ursache für die unterschiedlichen Implementierungen von `<RESOLVE` und Co ist. Da wird es spannend zu erfahren, wie man die Grenzen der 16–bit Welt sprengen kann. Auf jeden Fall anders als beim Atmega.

Amforth hat zudem das Design, dass viele Konfigurationswerte in einem Speicherbereich liegen, der relativ oft geschrieben wird aber trotzdem einen Neustart ohne weitere Maßnahmen übersteht. Beim Atmega ist der eingebaute EEPROM ideal geeignet, das MSP430 Pendant Info–Flash ist da nicht ganz adäquat, da ist ein Konstrukt wie `SAVE` um einen Stand abzuspeichern nicht zu vermeiden.

Der wohl folgenreichste Unterschied besteht darin, wie man einzelne Flashzellen neu beschreiben kann. Amforth hat Worte, die das auf einzelnen Adressen erlauben. Unter der Haube wird jedoch immer segmentweise gearbeitet. Dabei wird das betreffende Segment gegebenenfalls komplett gelöscht und anschließend mit den neuen Daten gefüllt. Die Atmegas haben eine Segmentgröße zwischen 16 und 64 bytes und verfügen darüberhinaus über einen speziellen Buffer. Die MSP430 haben im Codeflash eine Segmentgröße von 512 bytes und müssen ihr normales RAM benutzen. Bei einer RAM Ausstattung von 512 bytes ist das schlicht nicht machbar. Ergo ist der Flash im MSP430 wirklich write–once und damit sind alle nachträglichen Änderungen, die beim Atmega problemlos funktionieren, tabu. Prominentes Opfer ist `DOES>`. Das ist im Zusammenspiel mit `CREATE` nicht mehr einsetzbar.

Aus der Sequenz `CREATE ... DOES> .... ;` wird damit wie in alten Zeiten `<BUILDS ... DOES> ... ;`. Auf dem Atmega würde diese Strategie zumindest einen Flash-Erase Zyklus ersparen, der bislang zwingend ist, ob das aber als das Beste vom Brot durchgeht? Nicht einmal mehr gforth kennt `<BUILDS` noch.

Fazit und Ausblick

Inzwischen sind viele Basisfunktionen und Module wie Recognizer, Wortlisten, Environment Queries, Exceptions und einige weniger aufregende Dinge vereint. Dabei

habe ich auch sehr von der Arbeit anderer profitiert. So sind z.B. `CODE`, um mit einem Tool wie Commacode zusammezuarbeiten, und `1MS` (1 Millisekunde warten) vom 4€4th übernommen. Ein paar Dinge wie `DEFER` und `VALUE` harren noch der Umsetzung. Insgesamt sind mehr als 130 Worte gemeinsamer Code, etwa 60 in Assembler geschrieben und 60 weitere mehr oder weniger hardwarenah in Forth gestaltet. Der Rest, noch einmal 30 Worte, könnte noch zusammenwachsen. Ein weiteres großes Feld ist die Forth–Bibliothek. Sie ist bislang auf den Atmega zugeschnitten. Da wird es sicher noch die eine oder andere Anpassung geben. Auch gibt es noch etliche kleine Lücken und Bugs, die bereinigt werden müssen.

Nicht geändert wurden grundlegende Dinge, wie die Forth–VM selbst und die Dictionarystruktur. Bis jetzt wurde auch kein einziges Assemblerwort geändert, alle blieben, wie sie waren. Die meisten sind auch noch im Dictionary verankert.

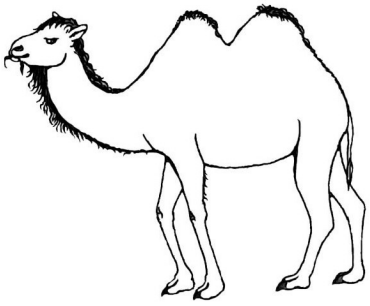
Amforth für den TI MSP430 ist ein spannendes Experiment geworden. Es führt, zumindest mir, sehr deutlich vor Augen, wo ein Forth für Mikrocontroller plattformspezifischen und wo gemeinsamen Code haben kann, selbst wenn man Assembler benutzt, was gemeinhin als der Inbegriff für plattform–spezifischen Code gilt. Bernd hat nur noch mit dem Kopf geschüttelt, wenn wir uns darüber im IRC ausgetauscht haben. Soll er ;)

Der Schwerpunkt von Amforth wird einstweilen die Atmel–Welt bleiben. Dazu ist einfach auch die schiere Anzahl der Testsysteme mitverantwortlich: Ich habe vielleicht ein Dutzend verschiedene Atmegas in allen möglichen Varianten zur Verfügung, aber nur genau ein MSP430 System, und das ist auch eher knapp motorisiert. Ein MSP430 mit FRAM ist ein weiteres interessantes Ziel, die Hardwareansteuerung hat das 4€4th–Team schon ausgeknobelt. Auf den Schultern von Riesen zu sein war nie schöner...

Und der ARM? Die erwähnte GCC Toolchain ist inzwischen ins Standardrepository von Ubuntu gewandert, das Make läuft durch, ein *native code* generierendes Forth mit viel RAM und CPU Leistung ist dann aber doch eine andere Liga.

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

Meldungen — Fortsetzung von Seite 7



Quelle: `http://www.bestcoloringpagesforkids.com/`

Dallas 1-Wire Drivers BSD Licensed — Juni 2014

Brad hat sich entschlossen, seinen Treiber unter BSD herauszubringen. Bitte beachten, wenn ihr den Treiber benutzt. Brad machte dabei noch mal deutich, dass nur der Source–Code für seinen Treiber BSD geworden ist, CamelForth[a] selbst aber unverändert unter GPL steht. Das betrifft also auch 4e4th, welches ja ein Ableger davon ist. mk

MSP430 CamelForth 0.5a Released — April 2014

In dem Zusammenhang finde ich es wichtig, darauf hinzuweisen, dass CamelForth jetzt für Mike Kohn's naken_asm cross–assembler angepasst wurde. Nur das Entwicklungstool hat sich damit geändert. Der MSP430–Code ist identisch zu dem, der vorher mit der IAR–Workbench erzielt worden ist. Es wurde sonst nichts zum Kernel hinzugefügt. Zukünftige Versionen des CamelForth werden nur noch damit gemacht werden. Da der Assembler sowohl in Linux als auch Mac–OS und Windows–Versionen läuft, sollte es also nun gelingen, von dieser elenden Bindung an undurchsichtige und monströse integrierte Entwicklungsumgebungen loszukommen. Bin gespannt, ob es auch 4e4th gelingt nachzuziehen. Und wenn sich dann auch noch Ting mit eForth dazu entschließen könnte, das wäre fein! mk

[a] Brad R.: CamelForth is a Forth implementation for embedded microprocessors. It is compatible with ANS Forth. It was originally developed as an educational project for The Computer Journal, but has since become popular for embedded systems programming.

noForth — ein Interview

Albert Nijhof und Willem Ouwerkerk

Wir haben die Macher von *noForth* — Albert Nijhof und Willem Ouwerkerk — getroffen, im Internet. Und wir durften ihnen ein paar Fragen zu noForth stellen.

VD: Albert und Willem. Danke, dass Ihr noForth gemacht habt, und danke, dass Ihr uns die Gelegenheit gebt, Euch ein paar Fragen zu stellen. Ich verstehe, dass das *no* in noForth für Nijhof und Ouwerkerk steht. Warum habt Ihr das noForth für den MSP430 überhaupt gemacht, wo es doch schon gute Forth–Systeme für diesen Mikrocontroller gibt, wie etwa Camelforth und sein Abkömmling 4e4th?[1]

NO: Zunächst wollen wir uns dafür bedanken, dass ihr uns in der VD Platz für einen Artikel über noForth einräumt.

Auf der Tagung in den Niederlanden[2] wurden wir mit 4e4th bekannt. Willem versuchte, mit dem 4e4th zu arbeiten, und merkte, dass es nicht stabil war. Wir dachten über Verbesserungen nach, kamen dann aber zu der Überzeugung, dass der Aufbau von 4e4th viel zu festgezurrt war, als dass man da noch etwas hätte verändern können. Außerdem wurden wir das handycap nicht los, dass 4e4th vollständig in Assembler formuliert ist.

Mittlerweile waren wir immer stärker von der Eleganz des MSP430 beeindruckt. Wir bauten einen Assembler (in Forth und im Forth–Stil), einen Disassembler, einen Simulator für MSP430–Code — und wir beschlossen, ein ganz neues Forth für den MSP430 zu konstruieren, „from scratch“, unter Einsatz eines Metacompilers und ganz unter Verwendung eigener Ideen. Warum? Weil uns die Beschäftigung damit großen Spaß machte.

VD: Seit Januar 2015 habt Ihr den Meta–Compiler, den, mit dem Ihr noForth generiert, als Open Source (GPL) veröffentlicht. Warum habt Ihr Euch entschlossen, einen Metacompiler statt eines Assembler–Listings zu verwenden. Ist das nicht zu komplizierte Schwarze Magie?

NO: Die Metacompilation ist durchaus keine Schwarze Kunst. Zugegeben, man kann etwas Kompliziertes daraus machen, man kann sie aber auch recht einfach halten. Wir haben das Letztere versucht.

Was noForth betrifft, benötigen wir:

1. Target–Code, Code, der noForth mit seinen eigenen Worten beschreibt.
2. Meta–Assembler oder Cross–Assembler im Host–Forth.
3. Meta–Code, mit den Hostforth–Definitionen der „roten“ Worte im Target–Code. Rote Worte sind die Worte im Target, die während des Metacompilierens in Aktion treten, im Gegensatz zu den blauen Worten, die compiliert werden (und also beim Metacompilieren passiv bleiben).
4. Meta–Interpreter. Der Meta–Interpreter liest den Target–Code, führt die roten Worte aus und compiliert die blauen.

Alles verläuft in Forth. Man kann alles nach Belieben gestalten, denn alles ist sebstgemacht. Wir halten das für einen großen Vorteil.

Auf unserer Website liegt die Datei „Wie noForth aufgebaut ist“.pdf (engl.) Darin wird in groben Zügen die Metacompilation von noForth erklärt.

VD: Die aktuelle Version von noForth kommt in zwei unterschiedlichen Ausrichtungen: C: kompakt und V mit Vokabularen. Warum habt Ihr diese beiden Versionen aufgeteilt. Wäre nicht „conditional compilation“ angemessener?

NO: Das könnte man sehr wohl tun, aber der Targetcode enthält bereits alles, was für die „conditional compilation“ in Verbindung mit den verschiedenen Platinen nötig ist, und wir wollen den Code gern lesbar halten. Übrigens waren die Unterschiede zwischen C und V anfangs größer. Die Methoden, um noForth C kompakter zu machen, haben wir später auch in noForth V verwendet.

VD: Als ich mir noForth angesehen habe, stellte ich fest, dass es ein gehashtes Wörterbuch statt einer einfach–verketteten Liste verwendet. Ist das nicht zu kompliziert für eingebettete Systeme? Welche Vorteile seht Ihr für eine solche Art von Wörterbuch–Struktur in diesem Kontext.

NO: Die Verwendung von Hash–Strängen ist an sich recht einfach. Für 8 Stränge kostet das 8 Zellen im RAM, mit Adressen von 8 Anfangsworten anstelle von nur einem Wort. `FIND` bestimmt zuerst die Strang–Nummer:

```
count  first-character  xor  7  and
( bei 8 Strängen in noForth )
```

und durchsucht dann diesen Strang, der eine verkettete Liste ist. Die Ermittlung der Strangnummer kostet sehr wenig Zeit. Bei 8 Strängen liegt die Zahl der Worte im Mittel bei 1/8 der Gesamtzahl. Das beschleunigt das Suchen und das ist vor allem von Belang, wenn man Programme ohne Handshake lädt.

noForth V kennt Wortlisten und man könnte vielleicht erwarten, dass jede neue Wortliste ein neues Strang–System benötigt. Doch das stimmt nicht. Die Wortlisten

[1] Und inzwischen weitere: mecrisp von Matthias Koch, und eforth von Ting.

[2] 2102, Beukenhof in Biezenmortel (NL)

haben mit den Strängen nichts zu tun. Die Worte einer Wortliste liegen nicht in einer eigenen Liste. Man kann am Header eines Wortes sehen, zu welchem Vokabular es gehört. Das bedeutet, dass `FIND` für einen Suchauftrag immer nur einen einzigen Strang ein einziges Mal durchsuchen muss, auch wenn beispielsweise 6 Wortlisten in der Search–Order stehen. Vor allem Zahlen im Inputstrom profitieren davon.

VD: Der Forth–2012–Standard wurde gerade veröffentlicht. Wie nah ist noForth am Standard? Welche Auswirkungen hat das Kompilieren ins Flash auf das Standard–Verhalten? Was macht Ihr mit `CREATE DOES>`?

NO: Die Unterschiede zwischen den Standards von 1994 und denen von 2012 sind für kleine embedded Systeme wahrscheinlich kaum oder überhaupt nicht relevant. Wir haben versucht, uns, so gut es ging, an den Standard zu halten, aber für ein ROM–System sind ein paar Abweichungen unvermeidlich. Wir beschreiben die in der Datei `"http://home.hccnet.nl/anij/nof/ram&rom.pdf`

Dann das Problem mit `DOES>` in ROM–Systemen:

```
: constant ( x -- ) create , does> @ ;
hex 20 constant bl
```

Worin besteht das Problem?

`CREATE` schreibt die DOCREATE–Adresse in die CFA von `BL` und `DOES>` überschreibt das mit der DOCON–Adresse, der DOES–Routine für `CONSTANT` (@), und das geht natürlich in ROM nicht.

Die noForth–Lösung ist ganz einfach:

`CREATE` setzt nichts in die CFA des Wortes, das es erzeugt. Dadurch kann `DOES>` seine Adresse in die CFA schreiben. Problem gelöst.

Oder doch nicht? Was, wenn da kein DOES> folgt?

```
   create logo  s" noForth" m, align
\ 'noForth' kommt direkt hinter der
\ leeren CFA im ROM, ohne count.
```

Und führt man nun das Wort `LOGO` aus, funktioniert es dennoch!

```
logo 7 type cr <enter> noForth
```

Wie kommt das? Also muss wohl doch die DOCREATE–Adresse in der CFA zu stehen? Nein! In einer noch nicht beschriebenen ROM–Adresse des MSP430 sind tatsächlich alle Bits gesetzt. Und auch dieses Wort wird über `FIND` aufgefunden. Aber wenn `FIND` ein Wort mit FFFF in der CFA findet, dann setzt `FIND` da das DOCREATE nachträglich ein. Das ist alles.

VD: Welche Worte in noForth gefallen Euch am besten?

NO: Am liebsten sind uns nicht so sehr bestimmte Worte, als vielmehr einige unserer Ideen, von denen wir annehmen, dass sie in Forth nicht allgemein verwendet werden. Aber es kann auch sein, dass das garnicht zutrifft, und wir wieder mal verschiedene Räder neu erfunden haben.

Solche Ideen werde dann zu Worten. Hier seht ihr einige davon:

- `NOFORTH` — shield + marker
- `FLYER` — Ringpuffer
- `GNIRTS` — effiziente RAM–Verwendung
- wordlist als Eigenschaft im Header von Worten
- `ALLOT` als Trigger für ein DOES>, das ins RAM weist
- `APP` — mit button
- Präfixe als Input–Tools — `HX DM BN DN CH`

VD: Wollt ihr auch Bibliotheken für die verschiedenen Peripherie-Module machen, wie *1wire* oder *SPI* oder *I2C* ?

NO: Nein, nur einfache beispielhafte Programme. Der Plan ist, für jede interne Hardware ein Beispiel zu machen, besonders für häufig benutzte Module. Vielleicht gelingt es sogar, ein noForth–Wiki aufzubauen, wo dann Code–Beispiele versammelt sind, und auch andere dazu beitragen können.

VD: Wie handhabt ihr die Interrupts?

NO: Ein Beispiel sagt da mehr als viele Worte — Beispiele findet ihr auf unserer Website.

(Im Kern ist es so, dass die Adresse eines assemblierten Codeteils, ein CODE-Wort — „body“ — in den Interruptvector geschrieben wird:

```
[’] my-ISR >body FFF4 vec!
```

Das Geheimnis liegt im Wort `vec!` , welches das macht. Denn eigentlich kann man das flash dort garnicht so ohne weiteres überschreiben. Aber das würde jetzt den Rahmen sprengen und ist dann schon ein eigener Beitrag. mk)

VD: Was schätzt ihr, wie sich noForth inzwischen verbreitet hat?

NO: Die Webseite hat keine Zähler. Und der Vertrieb ist auch nicht unser Interesse, wir brauchten es für uns selbst, haben es aber von vorn herein auch frei gegeben. Wir glauben, dass es ein sehr solides System geworden ist.

VD: Gab es Rückmeldungen über Anwendungen, die Leute mit noForth gemacht haben?

NO: Ich habe es dafür benützt, einen Hexapod mit zwanzig Servomotoren zu bauen. Da ist eine käufliche MSP430F149–Platine drin, dazu kam ein EEPROM am I2C–Bus und ein Bluetooth–Interface. Und ein Bekannter baut damit gerade ein kabelloses Sensoren–Netzwerk auf. Wir fordern niemanden auf, sein Projekt zu veröffentlichen.

VD: Wie sind Eure Pläne — Portierung auf ARM, Multitasking, TCP/IP?

NO: Das nächste noForth machen wir für die große FRAM–MCU, den MSP430FR5969. Und wir möchten weitere Hardware–Beispiele machen, populäre Module. Es gibt schon einen handlichen RS232–WiFi–Umsetzer,

nRF24L01 und sowas[3]. Rund zwanzig davon wurden inzwsichen schon ausgetestet. Wir glauben, dass ein Forth für einen Microcontroller nur populär werden kann, wenn es genügend Anwendungs–Beispiele dafür gibt. Um Forth auf so einen ARM zu bringen, muss man eigentlich nur ein Problem lösen, den wirklich sehr uneleganten Befehlssatz implementieren. Lieber machen wir weitere Experimentier–Bausätze, um mit den verschiedenen MSP–Platinen arbeiten zu können.

VD: Wir hätten da noch einige technische Fragen...

NO: Ohne die 'tools' (WORDS .S DUMP SEE) belegt noForth 8K, die C–Version etwas weniger, die V–version etwas mehr. Es ist *indirect threading* implementiert.

Wir versuchen noForth durch die Art der Implementierung klein zu halten, nicht so sehr durch Verminderung der Wortmenge. Zum Beispiel:

```
: ... ... 7 u< if 3 * else 8 - then ;
```

Das kostet bei klassischer Compilierung 14 Speicherzellen.

```
| lit |  7 |   u< | zbr | addr | lit |    3 |
|  *  | br | addr | lit |    8 |   - | exit |
```

Im noForth sind es nur 9 Zellen:

```
|  #7 | u< | zbr&addr | #3 | * |
|  br&addr | #8 |   - |   exit |
```

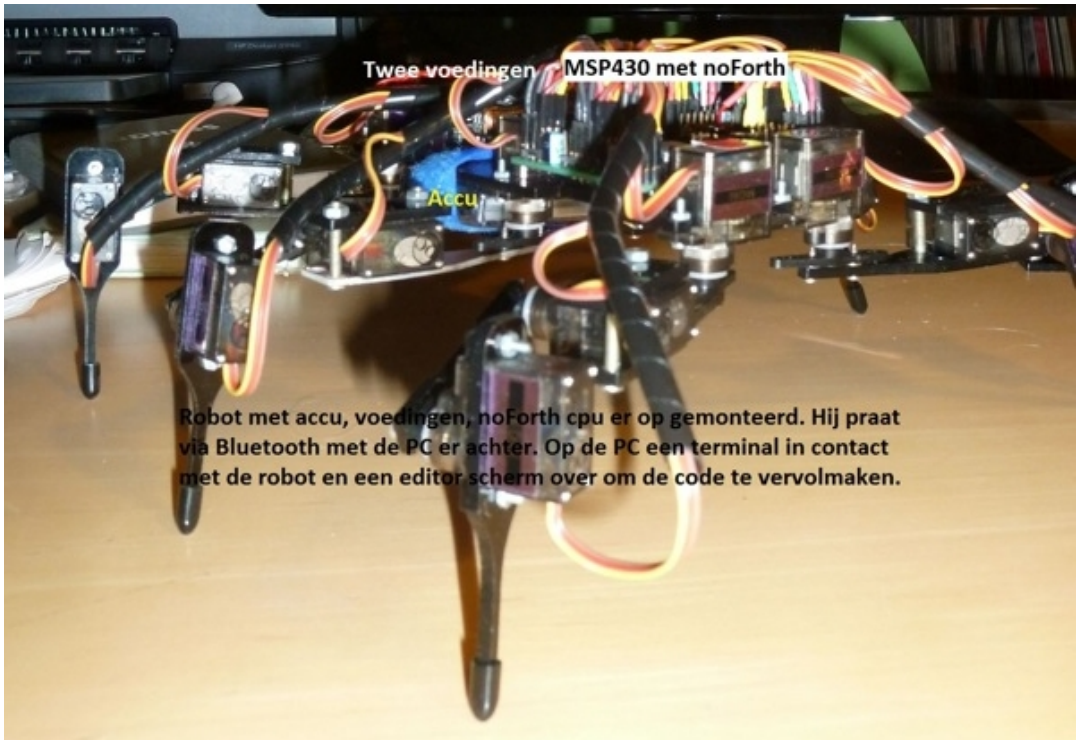
Die bedingten Verzweigungen und die meisten Zahlen kosten nur eine Zelle statt zwei. Die Platzersparnis wird dabei nicht nur im Kern sondern genauso auch in den später noch hinzugeladenen Programmen erzielt.

Was die Anzahl der Worte angeht, haben wir ein gutes Gleichgewicht zwischen spartanisch und vollständig erreicht. Es war uns zu irritierend, wenn ein neues Programm geladen werden soll, erst zu analysieren, was fehlt, und nachgeladen werden muss. Außerdem, wenn jemand das wirklich will, weniger Wörter im noForth–Kern, kann er ja jetzt selbst seinen Kern metacompilieren, und weglassen, was er für die Anwendung oder den Kern nicht mehr benötigt. Der Effekt ist aber vermutlich nicht groß. Eigentlich machen wir den Schritt in Richtung Kompaktheit mit den "Werkzeugen": Die kann man weglassen oder dazuladen.

VD: Albert und Willem, danke für das interessante Gespräch. Alles Gute für Euch und für noForth.
uho,fb,mk

noForth

http://home.hccnet.nl/anij/nof/noforth.html



[3] Interessenten nehmen bitte über die noForth–Webseite Kontakt auf zu Albert und Willem, um mehr zu erfahren.

Commacode

Ulrich Hoffmann

In seinem Artikel *A Transient MSP430 Forth Assembler* auf Seite 8 zeigt Brad Rodriguez, wie man einen transienten Assembler realisiert, der nur zeitweise im Speicher des Forth-Systems vorliegt und wieder entfernt wird, wenn er nicht mehr benötigt wird. Damit kann man wertvollen Platz für das Anwendungsprogramm schaffen, was insbesondere wichtig ist, wenn der (Flash-)Programmspeicher ohnehin klein ist.

noForth [1], ein interaktives Forth–System für die MSP-430–Familie von Albert Nijhof und Willem Ouwerkerk, geht mit `COMMACODE` [2] einen anderen Weg: Eine `CODE`–Definition wird in gleichbedeutende Code–Bytes, genannte *Komma–Code*, übersetzt, die später ohne Asssembler geladen werden können. Die Code–Bytes werden dann mit dem Forth-Wort `,` (Komma) in den Speicher geschrieben, was Pate für den Namen `COMMACODE` stand.

Commacode

Albert Nijhof und Willem Ouwerkerk schreiben zu `COMMACODE`:

> Den Assembler zu laden, nur um ein oder zwei Worte Deines Programms in Assembler zu schreiben, verbraucht relativ viel Platz.
>
> `COMMACODE` produziert *Komma–Code* für das zuletzt definierte Wort und macht es einfach, ein low–level Forth–Wort ohne Benutzung eines aktiven Assemblers zu kompilieren.

`COMMACODE` ist wie folgt definiert:

```
 1 \ For noForth C&V
 2
 3 : COMMACODE ( -- )
 4   created lfa>n count hx 1F and
 5   cr ." code " 2dup type
 6   3 spaces
 7   + aligned >body
 8   chere over - 2/ 0
 9   do @+ u. ." , "
10     dup 7 and 0=
11     if cr else space then
12   loop drop ." end-code " cr
13 ;
```

Es ermittelt zunächst den Namen des zuletzt definierten Wortes (Zeile 4) und gibt ihn als

```
Code Name
```

gefolgt von drei Zwischenräumen aus (Zeile 5 und 6). Zeile 7 bestimmt die Anfangsadresse des Maschinencodes. Da es sich um das zuletzt definierte Wort handelt, endet es an der Adresse `chere`. Zeile 8 berechnet, wie viele 16-Bit-Worte der Maschinencode also umfasst und legt diese Anzahl mit einer 0 als Schleifenparameter auf den Stack. Die `do-loop`-Schleife (Zeile 9-12) gibt nun jedes einzelne 16-Bit-Wort w in der Form w `,` aus. Zeile 10 und 11 bestimmen, ob dabei eine neue Zeile begonnen werden soll. Zum Abschluss gibt Zeile 12 noch `end-code` aus und die `Code`-Definition ist komplett.

Wie man's benutzt

Wie wird `COMMACODE` nun eingesetzt? Albert und Willem nennen folgende vier Schritte:

1. Den Assembler in noForth laden.
2. `COMMACODE` laden. (Den Code oben auswählen, kopieren und in Dein noForth–Terminal einfügen.)
3. Definiere Dein Wort in Assembler.
4. `COMMACODE` ausführen.

Beispiel

Für das Wort `2!`

```
1 code 2! ( lo hi a -- )
2  sp )+ tos ) mov   sp )+ 2 tos x) mov
3  sp )+ tos mov   next end-code
4 commacode
```

gibt `COMMACODE` in Zeile 4 den Komma–Code

```
code 2!   44B7 ,  0 , 44B7 ,
2 ,  4437 ,  4F00 ,  end-code
```

aus. Die Ausgabe kann dann statt der Assembler-Definition in das Programm übernommen werden. Weder der Assembler, noch `COMMACODE` selber sind später nötig, um den Komma-Code zu laden.

Soll nicht nur eine `CODE`–Definition zur Zeit in Komma-Code transformiert werden, sondern ganze `CODE`-Wort–Folgen, dann kann auch der Cross-Assembler `aux430asm.f` [3] eingesetzt werden, der Komma-Code produziert.

Aber Vorsicht. Nicht jede Assembler-Definition kann in richtigen Komma-Code überführt werden. Enthält die Assembler-Definition adress-abhängige Teile, zum Beispiel die Adresse einer Variablen `TEMPO`, dann sollte man die ausgerechnete `TEMPO`-Adressse (die im Komma-Code auftaucht) in „`TEMPO`“ zurückändern. Hier gilt also, dass man wissen muss, was man tut. `COMMACODE` ist aber auf alle Fälle ein nützliches, kleines Werkzeug in ressourcebeschränkten Umgebungen.

Links

[1] noForth: `http://home.hccnet.nl/anij/nof/noforth.html`
[2] COMMACODE: `http://home.hccnet.nl/anij/nof/commacode.html`
[3] aux430asm.f: `http://home.hccnet.nl/anij/nof/aux430asm.f`

Suchen durch Erkennen

Matthias Trute

Leser dieser Zeitschrift werden sicher wissen, dass Recognizer eines meiner Steckenpferde sind. Dass sie es einmal für Wert befunden würden, als Erweiterung von Forth angesehen zu werden, ist verrückt genug. Jetzt kommen noch weit tiefer greifende Ziele auf. Und das schon in der ersten öffentlichen Runde: Das gesamte SEARCH–ORDER–Wordset könnte komplett entfallen.

Wie konnte es dazu kommen?

Recognizer sind entstanden, um ein paar zusätzliche Literale, konkret Gleitkommazahlen, nachträglich in den Forth–Textinterpreter einbauen zu können. Bereits früh wurde erkennbar, dass man die auch für andere Sachen gebrauchen kann. Für Forth 2012 kam die Idee zu spät, aber das Lenkungsgremium hat sich entschieden weiterzumachen und hat auch eine lange Liste vorgelegt, was man alles noch zu tun gedenkt. Beim Forth 2012 hat sich ein Mechanismus namens Request For Discussion (RFD) eingebürgert, über den man Änderungsvorschläge einbringen kann. Das hat in der Vergangenheit ganz gut geklappt und soll wohl so weiterlaufen.

Diskussion des RFD

Erste Fassungen des RFD für Recognizer entstanden im späten Frühjahr 2014. Während man beim Code schreiben sich auf den Compiler verlassen kann und auch das eine oder andere ausprobiert, ist ein (potentieller) Standardtext aus einem anderen Holz geschnitzt. Hier muss man an alles denken und unmissverständlich beschreiben. Auch sollte man begründen, warum die Idee überhaupt wert ist, näher angesehen zu werden. Dabei halfen mir Bernd und Anton mit Rat und Tat. Da die beiden beim Forth 2012 aktiv mitarbeiten, ist wohl auch ihnen zu verdanken, dass die Recognizer schon im Herbst auf die Wunschliste des nächsten Forth gekommen sind, lange bevor ich den Text fertig hatte und öffentlich zugänglich machte.

So ein Text wird nie fertig. Irgendwas findet man immer, das noch besser formuliert werden sollte[1]. Am Ende wurden es 9 Seiten Text, von denen 5 den Kern des Ganzen beschreiben: Änderungen am Textinterpreter, 5 neue Worte und die Ergänzung eines bereits bestehenden Wortes (`MARKER`). Dazu eine Referenzimplementierung, die praktischerweise mit einer älteren gforth–Variante entstanden ist. Der Rest sind Testcases, Beispiele und die Geschichte, wie das Ganze entstanden ist.

RFD's müssen durch zwei Fegefeuer: Die Usenetgruppe `comp.lang.forth` und die Forth200x Mailinglist. Wer schon mal im Usenet die Postings in `comp.lang.forth` verfolgt hat, wird wissen, dass es dort eher robust zur Sache gehen kann. Da ist das deutlich ruhigere Temperament in der Forth200x Mailinglist für einen Neuling, der ich auf diesem Gebiet nunmal bin, passender.

[1] Nicht dass das mit Artikeln in der VD anders wäre.

Der Probelauf im November 2014 lief wie erhofft: Konstruktive Kommentare gemischt mit nebulösen Gurumeinungen, der Kopf blieb aber fest auf dem Hals. In der Vorweihnachtszeit gab es weniger Muße, da blieb das Thema erst mal liegen. Zum Jahreswechsel dann der Sprung ins kalte Wasser. Das Feedback von der Mailingliste war ermutigend, da wollte ich dann doch wissen, was der Rest der Meute meint. Deswegen, und weil es kaum Änderungen am Vorschlag gab, erschien der gleiche Text noch einmal. Klar, ein paar Leute sind in beiden Welten aktiv, aber es kamen neue Teilnehmer und auch neue Ideen. Wider Erwarten und ganz gegen alle Gewohnheit konstruktiv und sachlich.

Im Ergebnis stellte sich heraus, dass die Recognizer auf Interesse stießen und dass man ihnen viel zutraut. Kritik gab es kaum. So kam der Wunsch, einige Aspekte so zu formulieren, dass sie die bestehenden Definitionen nicht ersetzen sondern ergänzen. Das betraf insbesondere den Umstand, dass man mit den Recognizern den bisherigen starren Ablauf „FIND / NUMBER / NOT-FOUND“ aufbricht und er nicht mehr zwingend Grundlage des Textinterpreters ist. Dass man den Forth Interpreter komplett seines Dictionarys berauben kann, hat nicht jedem gefallen, da kam das Usenettemperament doch ein klein wenig durch. Am Ende gab es ein eindeutiges Votum, dass es so bleiben soll, wie vorgeschlagen. Inklusive dem Recht, sich mit der selbstgebauten Waffe in den eigenen Fuß schießen zu dürfen.

Daneben gab es einen Vorschlag, die Dinge zwar ganz anders zu sehen, aber trotzdem die gleichen Ziele auf einem sehr ähnlichen Weg zu erreichen. Es wird dadurch jedoch abstrakter und damit erklärungsbedürftiger. Dafür werden andere Dinge etwas weniger komplex. Die zweite Runde des RFD wird das sicher näher ausloten, vermutlich gemeinsam mit anderen Änderungen.

Einen sehr viel breiteren Raum nahmen jedoch Diskussionen über mögliche Einsatzgebiete ein. Ein Punkt, der schon auf der Mailingliste aufkam, war, dass es eigentlich doppelter Code sei, wenn man sowohl Recognizer als auch eine Search Order hat. Überhaupt schien die Beziehung zu den Aktionen, die der Interpreter mit den Ergebnissen der Suche anstellt, auf die Diskutanten anregend zu wirken.

Suchen

Das Search–Order–Wordset hat zwei Seiten. Zum einen einige Basisworte, damit man überhaupt mit Wortlisten arbeiten kann, und darüberhinaus eine ganze Menge an Worten, wie der Textinterpreter mit mehreren Wortlisten umgehen soll. Da gibt es eine, in die neue Worte hinein kommen (CURRENT), und ein Stack von Wortlisten, die der Reihe nach durchsucht werden sollen (ORDER), wenn es gilt, ein Wort im Dictionary zu finden. Dazu gibt es so schöne Worte wie `ALSO` und `PREVIOUS` oder `ONLY`.

Wie passt das zu Recognizern? Der Zusammenhang besteht im Wort `SEARCH-WORDLIST`. Das ist eines der wenigen Worte im Search–Order–Wordset, die eine Basisfunktion bereitstellen. Es sucht genau eine Wortliste nach genau einem Wort. Das Ergebnis ist für die bisherige Forthwelt maßgeschneidert: Execution Token und das Immediateflag in einer sonst ungebräuchlichen Notation. Das kann ein Recognizer einfach kapseln:

```
: rec:forth ( addr len --
    xt +/-1 r:word | r:fail )
  forth-wordlist search-wordlist
  ?dup if r:word else r:fail then ;
```

Damit hat man einen Recognizer, der die Standardworte von Forth finden kann. Will man mehr Wortlisten, definiert man sich einen Recognizer für diese und hängt ihn an der gewünschten Stelle im Recognizer–Stack ein. Und hier wird der gesamte ORDER–Stack überflüssig. Mitsamt den ihn verwaltenden Worten.

`ALSO` ist so eines. Es verdoppelt den ersten Eintrag in der Suchreihenfolge. `PREVIOUS` löscht den. Für diese finden sich Definitionen wie etwa

```
: also
   get-order over swap 1+ set-order  ;
: previous
   get-order nip       1- set-order  ;
```

Die kann man naheliegenderweise durch die Standardworte `DUP` und `DROP` ersetzen.

Interessant wird es, wenn man an die Feinheiten kommt. Wurden schon bei der Formulierung des Recognizer–RFD viele Dinge erst dann klar, als er geschrieben wurde, dürfte das bei einer angedachten Abschaffung eines ganzen Wordsets noch viel komplexer werden. Etwa folgendes: LOCALs werden vor allen anderen Worten gesucht. Nun kann man sich einen LOCALs–Recognizer vorstellen, der ganz oben im Stack steht. Was macht dann `PREVIOUS`? Das müsste eigentlich auf den speziellen LOCALs–Recognizer prüfen und den separat behandeln. Warum sollte nur dieser spezielle eine Sonderbehandlung erhalten?

Was wird aus `CURRENT`? Genauso, wie Recognizer bei der Analyse des Eingabetextes eingreifen, kann man sich einen Stack vorstellen, der bei der Definition neuer Worte konsultiert wird, um die passende Wortliste auszuwählen. Vielleicht wird das auch ein Ort, wo man Definitionen der Art `: 0 -1. ;` abfangen kann, allen Wünschen nach Schüssen in den Fuß zum Trotz.

Name Tokens

Name Tokens sind kurz vor Toresschluss in den Forth–2012–Standard aufgenommen worden. Sie werden von `traverse-wordlist` bereitgestellt und mit Worten wie `name>interpret` für den Rest der Forth–Welt verständlich gemacht. Sie sind für alle Dictionaryeinträge verfügbar, :nonames haben also keine. Welches Problem sie genau lösen, ist mir etwas unklar. Was sie aber machen können, ist, Execution Tokens den Rang abzulaufen.

Ganz unabhängig von den möglichen Änderungen beim SEARCH–ORDER–Stack kann man auch hier Recognizer benutzen, um neue Welten zu erkunden. Das zugrundeliegende Forth muss nur recht aktuell sein, Forth 2012 mitsamt den `NAME>x`–Worten ist noch nicht allzu verbreitet.

```
\ der Aktionsteil vom Recognizer
:noname name>interpret execute ;
:noname name>compile execute ;
:noname name>compile swap
   postpone literal compile, ;
recognizer: r:name

\ Hilfswort: Ist ein String addr/len der
\ Gleiche wie im Name Token? flag wird
\ von traverse-wordlist benutzt.
: is-in-nt? ( addr len flag nt --
    addr len false | nt true )
  >r drop 2dup r@ name>string
  compare if
    r> drop 0 true
  else
    2drop r> 0
  then ;

\ Analog zu search-wordlist
: search-name ( addr len wid -- nt | 0 )
  >r 0 ['] is-in-nt? r> traverse-wordlist
  dup 0= if
   2drop drop 0
  then
;

\ Der Parsingteil vom Recognizer
: rec:name ( addr len -- nt r:name | r:fail )
   forth-wordlist search-name
   ?dup if r:name else r:fail then ;

\ Ersetze rec:word mit rec:name und
\ alles sollte so laufen wie zuvor.
```

Auch hier gibt es natürlich ein paar offene Punkte. So ganz kann man wohl nicht auf Execution Tokens verzichten. Was macht ein `RECURSE` ohne sie? Wie kann ein `:NONAME` ein *Name* Token haben?

Zusammenfassung

Soweit der Stand der Dinge nach den ersten öffentlichen Diskussionen. Die Vertreter der „professionellen“ Forths

(Forth Inc und MPE) blieben bislang still, auch wenn sie zweifellos die Diskussionen verfolgt haben.

Die skizzierten Ideen für Einsatzgebiete von Recognizern auf Forth–Standardniveau stehen noch ganz am Anfang. Sie strahlen eine gewisse Einfachheit aber auch Radikalität aus. Nun verschwinden keine Dinge aus Standards, das dauert sehr lange, bis das passiert, wenn überhaupt. Wer die Sequenz `ONLY FORTH ALSO DEFINITIONS` mag, wird sie mit Sicherheit noch lange benutzen dürfen. Auf der anderen Seite kann man eine neue Welt erkunden und, nach meinen Erfahrungen mit dem RFD, mit Gewinn auch mitgestalten.

Bei den Recognizern reichten 5 neue Worte, von denen 4 nur der Verwaltung dienen und das 5. auch nur mäßig komplex ist, aus. Früher hätte man gesagt, ein Screen voll.

SWAP und seine Freunde

SWAP hat Freunde in aller Welt, z. B. diesen hier:

Frage: **Wo befindet sich dieser Swap-Freund?**

Antworten bitte an die Redaktion unter vd@forth-ev.de.

Die Antwortgeber der richtigen Antworten werden im kommenden Forth-Magazin veröffentlicht. uho

Forth–Gruppen regional

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

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

Hamburg **Ulrich Hoffmann**
Tel.: (04103) – 80 48 41
`uho@forth-ev.de`
Treffen: Alle 14 Tage in der FH Wedel/Tematik, Mittwochs ab 10:00.

Ruhrgebiet **Carsten Strotmann**
`ruhrpott-forth@strotmann.de`
Treffen alle 1–2 Monate Freitags im Unperfekthaus Essen
`http://unperfekthaus.de`.
Termine unter : `http://forth-ev.de`

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

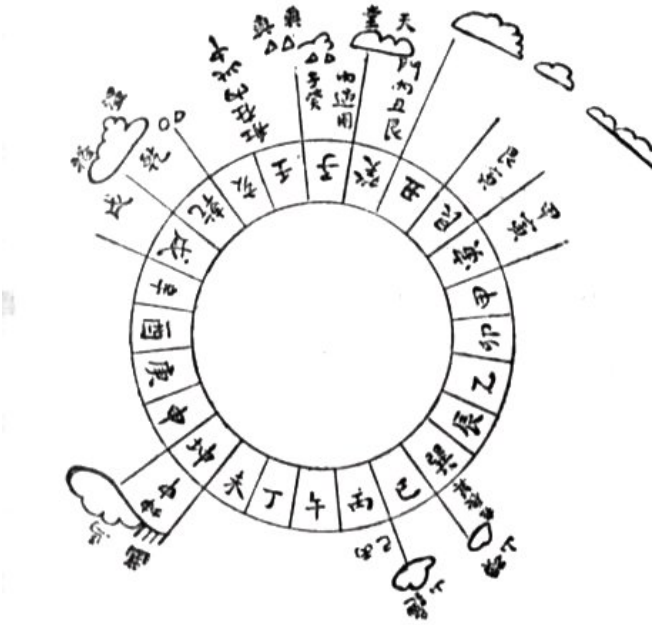
Forth-Hardware in VHDL microcore (uCore)	**Klaus Schleisiek** Tel.: **(0 75 45) – 94 97 59 3**$_p$ kschleisiek@freenet.de
KI, Object Oriented Forth, Sicherheitskritische Systeme	**Ulrich Hoffmann** Tel.: **(0 43 51) – 71 22 17**$_p$ Fax: **– 71 22 16**
Forth–Vertrieb volksFORTH ultraFORTH RTX / FG / Super8 KK–FORTH	Ingenieurbüro **Klaus Kohl–Schöpe** Tel.: **(0 82 66)–36 09 862**$_p$

Termine

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

9.–12. April 2015:
Forth–Tagung in Hannover
1.–3. Mai 2015:
VCFe 16 in München `http://vcfe.org`
6.–7. Juni 2015:
Makerfaire in Hannover `http://makerfairehannover.com/`

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



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

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

Einladung zur

Forth–Tagung 2015 vom 9. bis 12. April in Hannover

Unterbringung und Tagung im Bed'nBudget + CongressCentrum Wienecke XI. in der Hildesheimer Straße 380 · 30519 Hannover

Anreise

Hannover ist über die Autobahn A7 erreichbar, mit dem ICE und hat natürlich auch einen Flughafen. Vom Hauptbahnhof kommt man mit der Straßenbahn Linie 1 und 2 zur Haltestelle Wiebergstraße zum Hotel.

Anmeldung

Auf `http://tagung.forth-ev.de` finden sich noch viele weitere Informationen, das aktuellste Programm sowie die elektronische Anmeldung.

Programm

Donnerstag

ab 13:00	Frühankommer(–Workshops)

Freitag

vormittags	Frühankommer(–Workshops)
nachmittags	Begin der Tagung, Vorträge und Workshops

Samstag

vormittags	Vorträge und Workshops
nachmittags	Exkursion

Sonntag

09:00	Mitgliederversammlung
13:00	Ende der Tagung