Vierte Dimension

4/2016

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

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.

KIMA Echtzeitsysteme GmbH

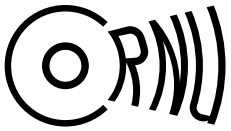
Güstener Strasse 72 — 52428 Jülich
Tel.: 02463/9967-0 — Fax: 02463/9967–99
`www.kimaE.de` — info@kimaE.de

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.



Cornu GmbH
Ingenieurdienstleistungen
Elektrotechnik

Weitlstraße 140
80995 München
`sales@cornu.de`
`www.cornu.de`

Unser Themenschwerpunkt ist automotive SW unter AutoSAR. In Forth bieten wir u.a. Lösungen zur Verarbeitung großer Datenmengen, Modultests und modellgetriebene SW, z.B. auf Basis eCore/EMF.

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`

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, Michael Kalus
E–Mail: `4d@forth-ev.de`

Anzeigenverwaltung

Büro der Herausgeberin

Redaktionsschluss

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

Erscheinungsweise

1 Ausgabe / Quartal

Einzelpreis

4,00€ + Porto u. Verpackung

Manuskripte und Rechte

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

Liebe Leser,

vermutlich ist es Euch schon aufgefallen, dass das Titelbild sich nun zum dritten Mal hintereinander absichtlich nur in den Farben unterscheidet. Weil das die Verschlüsselungs–Serie unterstreichen soll, die Rafael Deliano geschrieben hat. Wie angekündigt, hier also seine dritte und damit letzte Folge in dieser Reihe, der *SHA–1*.
In der Zwischenzeit ist Willem Ouwerkerks *noForth–Cloner* fertig geworden und hat mich persönlich total begeistert. Hab das also gleich nachgebaut und kann bestätigen: Funktioniert bestens!
Auch die EuroForth 2016 im Inselglück auf Reichenau war inzwischen. Die Insel im Bodensee hat ja bekantlich durch die temperaturausgleichende Wirkung des Bodensees, die positiven Auswirkungen des Alpenföhns und ihre hohe Zahl an Sonnentagen ein besonders mildes Klima. Obst und Gemüse gedeihen da prächtig, bis zu drei Ernten im Jahr hat es dort. Und gab es auch eine reiche Forth–Ernte? Seht selbst — Anton Ertel hat die Beiträge schon veröffentlicht.
Matthias Koch stellt heraus, dass es Clifford Wolf aus Österreich und seinem Team gelungen ist, den Bitstream der Lattice–iCE40–FPGAs zu verstehen und freie Werkzeuge für die Entwicklung von Logik zu schreiben — eine wegweisende Pioniertat! So wurde *mecrisp–ice* möglich.
Das Kairos, die günstige Gelegenheit, wurde bei den alten Griechen als Gottheit mit Locke, die ins Gesicht fällt, und kahlem Hinterkopf dargestellt. Man kann sie daher „beim Schopfe packen“, wenn sie kommt, aber nicht mehr fassen, wenn sie vorüber ist. Aber hier geht es ums Chronos, die gemessene Zeit in Stunden, Minuten und Sekunden ab Mitternacht eines Tages im Kalendersystem. Erich Wälde zeigt sein *Clock–Works*–Projekt für die Amateurfunkfreunde.
Und Jean–Claude Wippler stellt seinen Forth–Live–Explorer *Folie* vor. Auf das es von berufenen Linux–Menschen getestet und gewürdigt werde, was dann hier gerne abgedruckt werden kann. Aus Win7–Sicht kann ich dazu sagen: Läuft.
Eines Tages werde ich es noch erleben, dass die großen Forthsysteme auch schon Mittel an Bord haben, um sich darin zu orientieren. Gute Ideen dazu sind dringend nötig. Ein `words` mit *alpha–numerischer Sortierung* stellt Filippo Sala vor. Bitte mehr davon!
Stacks für Forth sind so fundamental, dass weder Forth 2012 noch die Forth Foundation Library sie für Anwender bereitstellen. Dennoch braucht man für gewisse Anwendungen zusätzliche Stacks. Matthias Trute liefert hier den Baukasten dafür.

So, und im kommenden Frühjahr, gleich nach Ostern, ist unsere *Forth–Tagung* im Westen des Landes — in einem Atomkraftwerk, das nie eins wurde. Meldet Euch und Eure Beiträge schon mal an!

Bis dahin!
Euer Michael

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
`http://fossil.forth-ev.de/vd-2016-04`

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

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

Liebe Maker, Tüftler, Bastler, Designer, Wissenschaftler und Erfinder,

es ist soweit! Die Maker Faire Ruhr geht in die zweite Runde! Nach der erfolgreichen Premiere dieses Jahr findet das kreative Erfinderfestival am Wochenende 25.–26. März 2017 in der DASA–Arbeitswelt–Ausstellung in Dortmund statt. Veranstalter der Maker Faire Ruhr ist die DASA, das Team von city2science ist Ansprechpartner für das Programm. Ab sofort können sich alle interessierten Maker anmelden, der Call for Makers ist eröffnet!

Infos findet Ihr unter:

`http://www.makerfaire-ruhr.com/`

und auf facebook:

`https://www.facebook.com/MakerFaireRuhr/`

Wir würden uns sehr freuen, Euch auf der Maker Faire in Dortmund begrüßen zu dürfen und sind schon ganz gespannt auf Eure Projekte! Gerne können wir auch gemeinsam Ideen für eine Präsentation entwickeln, sprecht uns einfach an!

Anmelden könnt Ihr Euch bis zum 16. Dezember 2016!

NoForth, das Egel Project und der Cloner

In ihrem *Werkboek*, dem *Egel Project*, haben Willem Ouwerkerk und Albert Nijhof in diesem Sommer eine stattliche online–Sammlung von elementaren Beispielen für den Umgang mit der TI[1]–MSP430G2553–MCU herausgegeben. Gut erklärt, klar bebildert und mit Quellenangaben versehen, leitet das Arbeitsheft an, sich den Chip mit Hilfe von *noForth* zu erschließen. Besonders angetan hatte es mir das Projekt Nr. 111: *Cloning noForth*. Willem war so nett, daraus eine Beitrag für unser Heft zu machen. Danke!

Mich selbst hat das dazu inspiriert, einen eigenen Schaltungsentwurf für den Cloner zu machen. Auf einem 7x12–Lochraster–Platinchen[2] wurde der Chip aufgebaut in Minimalbeschaltung, mit dem Luxus zweier LEDs für die Anzeige des Clone–Vorgangs — das *noForth–Plaatje*. Die Art der abgewinkelten Steckverbindungen der Plaatjes, an der einen Schmalseite Stecker, gegenüber Buchsen, bewirkt, dass man einfach zwei davon zusammenstecken kann, um von einem zum anderen seine App zu clonen. Das erste Plaatje ist mit dem Terminal via USB–Seriellem Wandler verbunden. Darin läuft des kleine Cloner-Programm, das auf noForth aufgebaut ist, und vom Terminal aus bedient werden kann. Im zweiten, dem Target, steckt eine „leere“ MCU, die der Clone wird. Das geht ruck zuck! noForth kopiert sich selbst von MCU zu MCU — klasse! Wird nicht geclont, ist die Buchse an der Schmalseite auch 4+1–Bit–Experimentierstecker. Das untere Nibble von P2 und ein weiterer Pin sind dann für Experimente oder kleine Anwendungen frei. Wer Interesse daran hat, kann mich gerne anschreiben. mk

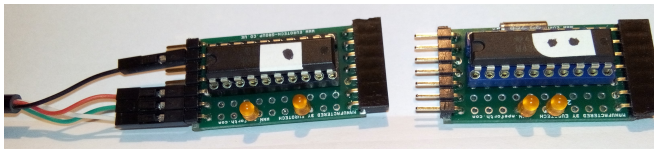


Abbildung 1: Die noForth–Plaatjes als Cloner

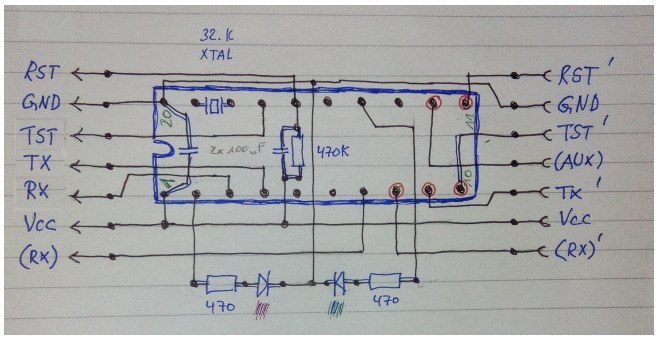


Abbildung 2: Der noForth–Plaatje–Schaltplan

JEELABS

COMPUTING STUFF TIED TO THE PHYSICAL WORLD

This is a daily weblog[3] about my experiments based on the mix of electronics and computing called *Physical Computing*. I'm building little *sensor nodes* to measure electricity use, temperature, light, etc. which they transmit wirelessly to one of the central computers in the house. One goal is to better understand how we're using electricity and how we're heating the home, so that we can try to improve on it. ...

If you are interested in electronics and/or computing, then maybe some posts on this weblog will encourage you to go out and explore the fascinating world of physical computing. There are 1400 posts. Yes, it's that much fun![4]

Explore! Tinker! Create! Learn! Invent! What better way is there to shape our future than to be part of it!

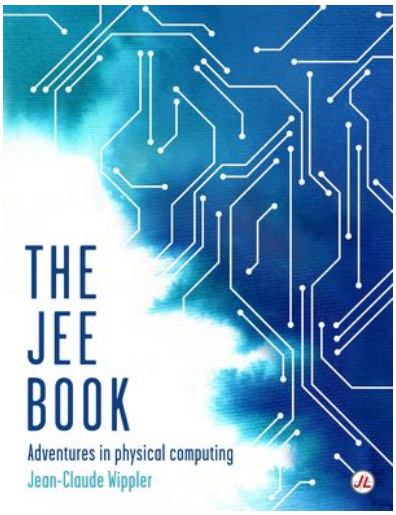
Oh, and enjoy your visit. jcw

[1] Texas Instruments

[2] Gestiftet von mpeforth, gemacht von eurotech co uk, und vermittelt von Jürgen Pintaske, erhielt ich neulich ein Bündel davon - Danke!

[3] http://jeelabs.org/intro/ — by Jean–Claude Wippler

[4] JCW used to pubish about ever day since 2008, so this is a rich source now !



PS: All seine Beiträge gibt es nun auch als ebook zu kaufen.

Tags & Topics: AC Mains Analog Arduino Audio CNC DIJN Displays Electronics Events Flashback Forth Graphics Heating Home automation HouseMon ISP JavaScript JeeBot JeeDay JeeHub JeeLink JeeMon JeeRev JeeSMD Linux Metering Musings MuxShield Network Oscilloscope POF Power RBBB Reflow RTOS Sensors Solar Teardown Toolkit What-If X10 mk

https://leanpub.com/jeebook

EuroForth 2016

The EuroForth 2016 proceedings and the individual papers and/or presentation slides, as well as BibTeX entries for these papers and slides, are now available.

The authors and titles of the papers are:

- Refereed Papers
  - Ulrich Hoffmann and Andrew Read: A synchronous FORTH framework for hard real-time control
  - Sergey Baranov: Simulating Recurrent Neural Networks in Forth
- Non-Refereed Papers
  - Nick J. Nelson: Tunnel Vision
  - Bill Stoddart: The Halting Problem in Forth
  - Andrew Read: An Axiomatic Approach to Forth
  - Wolf Wejgaard: Planet Holonforth M.
  - M. Anton Ertl: Sections
  - M. Anton Ertl: Recognizers: Arguments and Design Decisions
  - Stephen Pelc: The Sockpuppet Forth to C interface
- Late Non-Refereed Papers
  - Ulrich Hoffmann: Implementing the Forth Inner Interpreter in High Level Forth
  - Gerald Wodni: f --- A package manager for (the)Forth(.net)
- Presentation Slides
  - Bernd Paysan: net2o: Using net2o
  - M. Anton Ertl: Security

Videos have been recorded and will become available when they are ready.

Check these links for further informations about the Papers and BibTeX, to contact M. Anton Ertl, visit comp.lang.forth FAQs, get the *new forth standard* and see the EuroForth 2016 itself too:

`http://www.euroforth.org/ef16/papers/`

`http://www.euroforth.org/ef16/papers/euroforth16.bib`

`http://www.complang.tuwien.ac.at/anton/home.html`

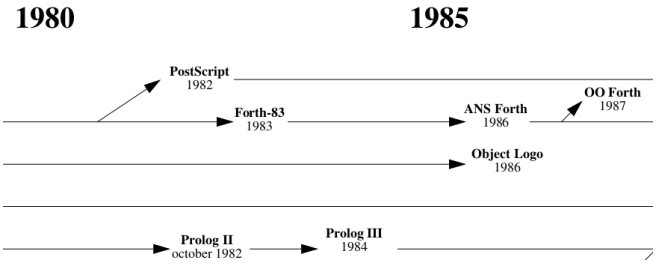
`http://www.complang.tuwien.ac.at/forth/faq/toc.html`

`http://www.forth200x.org/forth200x.html`

`http://www.euroforth.org/ef16/`

anton

Die Geschichte der Computersprachen



Éric Lévénez arbeitet bei der französischen Telecom, wo er mit MIRABEL die Funktion der Telefonleitungen prüft und Übertragungsprobleme behebt sobald sie auftreten. Sein Hobby sind historische Betrachtungen in der Computerei. Dabei ist auch ein Zeitstrahl zur Geschichte der Computersprachen entstanden. Diese stolze Wandtapete gibt es, zu betrachten mit einer hübschen Lupenfunktion, online auf seiner Homepage und auch als PDF. Er hat 50 Sprachen in die Tapete aufgenommen, beginnend mit Fortran. Forth hat auch seinen Platz darin gefunden. (aus dem ZDF; Brühl und Ertel gingen dem nach.) mk

`https://www.levenez.com/lang/`

Weiter auf Seite 10 ...

SHA–1 Secure Hash Algorithm

Rafael Deliano

Die bekanntesten kryptographischen Verfahren sind sicherlich Chiffren wie DES. Anwendungen wie digital signatures oder Bezahlsysteme benötigen aber andere Funktionen. Sie gewinnen auch in embedded Anwendungen an Bedeutung (Abb.9).

Signaturen

Diese wurden auch in unserer Zeitschrift Vierte Dimension schon mal behandelt [2]. Mit Hinweis auf die Anwendung im Segelflugwettbewerb, um die Authentizität von Messdaten sicherzustellen. Umfeld und Vorläufer von SHA werden in [3] und besonders [4] dargestellt.

Hashing wird in der Computertechnik z.B. zur Optimierung von Speicherzugriffen verwendet. Üblich sind dafür schnelle, aufwandsarme Verfahren. Man toleriert dabei viele Hash–Kollisionen. CRC wäre verwendbar und besser, ist aber schon zu aufwändig.

Verglichen mit SHA ist CRC mit dem simplen LFSR schnell berechenbar und die Prüfsumme von selten mehr als 32 Bit sehr kompakt. Es stört nicht, dass immer noch relativ viele Datenblöcke die gleiche Prüfsumme haben. Demgegenüber berechnet SHA–1 mit enormem Aufwand ein mit 160 Bit recht voluminöses Ergebnis (Abb.1). Vorteil ist aber, dass auch bei großen Datenblöcken wenige gekippte oder gefälschte Bits sofort die Signatur verändern.

Abbildung 1: Verwendung

SHA–1

Die Hash–Prüfsumme wird hier „Message Digest“ genannt, eine Referenz an Ronald L. Rivest von MIT (bzw. RSA Security). Der hatte mit MD4 den Vorläufer entwickelt. Der wurde von Rivest als MD5 verbessert und hatte auch Verbreitung. SHA ist eine auch auf MD4 basierende verbesserte Entwicklung der Regierungsbehörde NSA, und wurde 1993 von NIST als Standard veröffentlicht. Die erste Variante war leicht defekt und wurde 1995 durch SHA–1 ersetzt [1].

Unter dem Sammelbegriff SHA–2 sind damals drei weitere Varianten mit erhöhten Wortlängen beigepackt worden. Als SHA–3 wurden 2015 weitere „noch bessere“ Algorithmen verabschiedet. Die basierten auf einer öffentlichen Ausschreibung ähnlich AES. Mit ähnlichen Wortlängen wie SHA–2.

Bisher hat vor allem SHA–1 praktische Verbreitung.

Algorithmus

Der Daten–Eingangsstrom wird in Blocks zu 64 Byte verarbeitet (Abb.2). Das übliche Datenformat (Abb.3) sieht vor, dass der letzte Block mit Nullen aufgefüllt wird. Man beachte aber, dass vorher noch eine Eins eingefügt ist, hier 80h. Zudem befindet sich am Ende des Blocks eine 64–Bit–Summe der verwendeten Datenbits. Hier 1Ch = 24d = 3 Byte.

Wie von derartigen Verfahren gewohnt, wird sequentiell in Rounds durchgequirlt (Abb.4). Die 20 Byte Hash–Summe H und MD müssen für den ersten Datenblock mit Konstanten initialisiert werden (Abb.5).

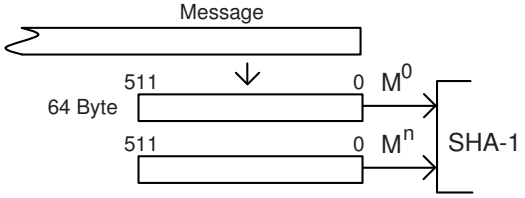


Abbildung 2: Zerlegung und Padding der Daten

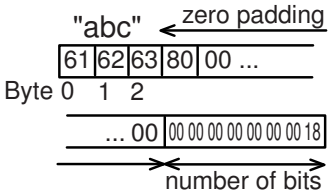


Abbildung 3: Format Message

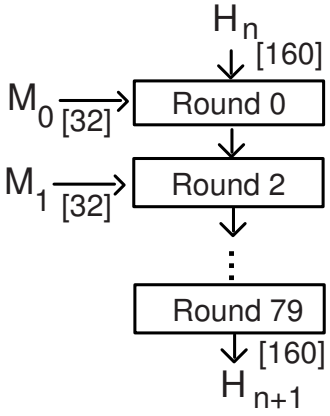


Abbildung 4: Rounds

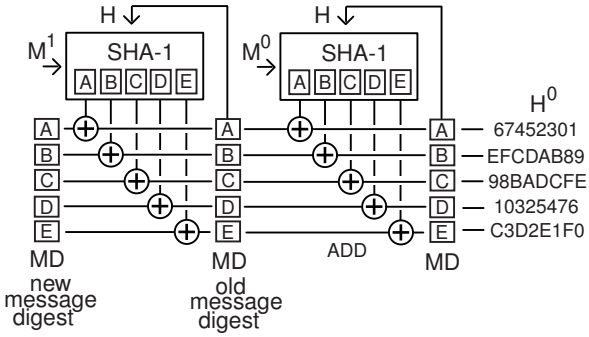


Abbildung 5: Initialisierung

Rounds

Die fünf 32–Bit–Datenworte werden getrennt verarbeitet (Abb.6). *A* ist das oberste, *E* das unterste Datenwort. Es ist naheliegend, 4 Schleifen zu kaskadieren, da Funktionen und die Kj Konstanten wechseln. Die bitweisen 32 Bit AND, OR und XOR sind schnell und simpel ausgeführt. Der ROL–Befehl hat 5 oder 30 Schritte. Bei den Additionen wird das überlaufende Carrybit ignoriert.

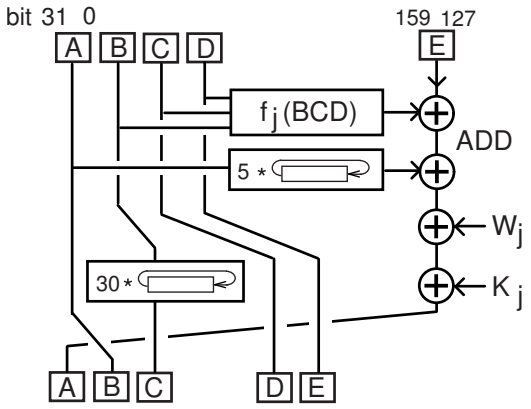


Abbildung 6: Round

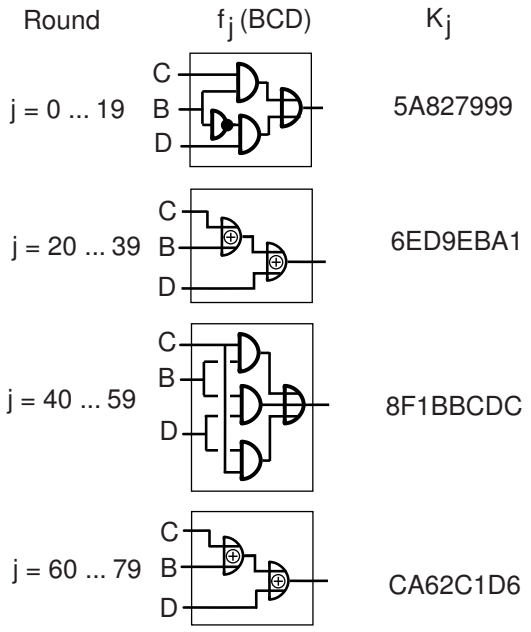


Abbildung 7: Teilfunktionen für Round

Message Schedule

Typisch wird die Tabelle (Abb. Bild 8) vor Beginn der Rounds komplett aufgebaut. Der Speicherverbrauch ist mit 380 Byte RAM üppig. Der untere Teil entspricht den 64–Byte–Datenblocks aus Bild 3 in 32–Bit–Worten angeordnet. Die folgenden Datensätze werden sequentiell aus zurückliegenden Datenworten berechnet.

In [1] wird alternativ eine Variante vorgeschlagen, bei der der obere Teil entfällt. Da j von 0 ... 79 kontinuierlich durchlaufen wird, aber nur die 16 unmittelbar zurückliegenden Worte benötigt werden, kann man den Speicherbereich auf dieses Fenster beschränken. M wird dabei aber überschrieben. Meist ist das nicht praktikabel.

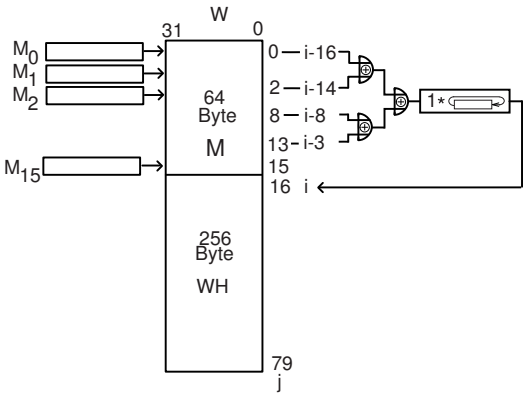


Abbildung 8: Message Schedule Tabelle W

Message Digest

Das Teilergebnis H muss zu MD addiert werden, um einen neuen Ausgabewert MD zu erzeugen. Dieser ist auch wieder die Initialisierung für H (Abb.5). Die dargestellte Nachricht belegt inklusive Padding zwei 64–Byte–Blocks .

Implementierung

Detaillierte Ausdrucke von Testläufen finden sich in [1].

Der GP32 hat keine zusammenhängende 380 Byte RAM verfügbar, W musste deshalb in zwei getrennte Tabellen M und WH aufgeteilt werden.

32–Bit–Befehlssatz in Assembler für die simplen logischen und arithmetischen Befehle war leicht nachzurüsten. Das Programm wird dann ziemlich kompakt, aber leider nicht schnell (Tabelle 1). Sobald man eine lauffähige Version hat, kann man die aber schematisch auf Assembler umsetzen. Das Listing wurde voluminös, aber die Ausführungszeit ist gut.

		HLL	Assembler
Init WH	[msec]	60	13
Rounds	[msec]	180	23
RAM	[Byte]	360	360
FLASH	[kByte]	1,4K	1,1K

Tabelle 1: Speicher und Timing; 64 Byte Block; MC68HC908GP32 @2,45MHz

Abbildung 9: iButton von Maxim mit SHA–1–Logik

Literatur

[1] Secure Hash Standard FIPS 180

[2] Hoffmann „Digitale Signaturen und eingebettete Systeme“ VD 2000/4

[3] Mitchell, Piper, Wild „Digital Signatures“ in: „Simmons Contemporary Cryptology“ IEEE Press 1992

[4] Menezes, Oorschot, Vanstone „Handbook of Applied Cryptography“ CRC 1997

Listing

```
\ primitives & assembler ----------------------

: D@  DUP 2+ @ SWAP @ ;   \ ( Addr -- D1 )
: D!  SWAP OVER ! 2+ ! ;  \ ( D1 Addr -- )
:CODE D+ ... CODE;        \ ( UD1 UD2 -- UD3 ) +
:CODE DNOT ... CODE;      \ ( UD1 -- UD2 ) NOT
:CODE DAND  ... CODE;     \ ( UD1 UD2 -- UD3 ) AND
:CODE DOR   ... CODE;     \ ( UD1 UD2 -- UD3 ) OR
:CODE DXOR  ... CODE;     \ ( UD1 UD2 -- UD3 ) XOR
:CODE 1*DROL  ... CODE;   \ ( UD1  -- UD2 ) ROL
:CODE 24*DROL ... CODE;   \ ( UD1  -- UD2 ) ROL
: 5*DROL   5 1 DO 1*DROL LOOP ; \ ( UD1  -- UD2 )
: 30*DROL  24*DROL        \ ( UD1  -- UD2 )
           6 1 DO 1*DROL LOOP ;

\ RAM ----------------------------------------

$HEAP  CONSTANT  WH  \ 256 byte RAM 16 ... 79
D% 20  ZVARIABLE H   \  20 byte RAM Hash
H         CONSTANT A"
H     4 + CONSTANT B"
H     8 + CONSTANT C"
H D% 12 + CONSTANT D"
H D% 16 + CONSTANT E"
D% 20  ZVARIABLE MD  \  20 byte RAM Message Digest
D% 64  ZVARIABLE M   \  64 byte RAM Message
                     \             W 0 ... 15
TABLE H-INIT  \ big endian
67 C, 45 C, 23 C, 01 C, \ A
EF C, CD C, AB C, 89 C, \ B
98 C, BA C, DC C, FE C, \ C
10 C, 32 C, 54 C, 76 C, \ D
C3 C, D2 C, E1 C, F0 C, \ E

\ Fill WH & read W --------------------------

: D@"   \ ( J  -- UD1 )
DUP 8000 AND
IF        \ M: -16 ... -1
  D% 16 + \      0 ... 15
  1<SHIFT 1<SHIFT M  + D@
ELSE      \ WH:  0 ... 63
  1<SHIFT 1<SHIFT WH + D@
THEN ;

: FILL-WH  \ (  --  ) in: M
D% 63 0 DO
I D% 16 - D@"
I D% 14 - D@" DXOR
I D%  8 - D@" DXOR
I D%  3 - D@" DXOR 1*DROL
I 1<SHIFT 1<SHIFT WH + D!
LOOP ;

: W@   \ ( J -- UD1 ) J= 0 ... 79d
DUP FFF0 AND
IF   10 - 1<SHIFT 1<SHIFT WH + D@ \ WH
ELSE      1<SHIFT 1<SHIFT M  + D@ \ WL
THEN ;

\ Rounds ---------------------------------

: (SHA1)   \ ( UD1 --  )
A" D@  5*DROL D+
D" D@            E" D!
C" D@            D" D!
B" D@ 30*DROL    C" D!
A" D@            B" D!
                 A" D! ;

: SHA1   \ (  --  ) in: H , M ; out: H
D% 19    0 DO
B" D@ C" D@ DAND B" D@ DNOT D" D@ DAND DOR
E" D@ D+ 7999 5A82 D+ I W@ D+
(SHA1) LOOP
D% 39 D% 20 DO
B" D@ C" D@ DXOR D" D@ DXOR
E" D@ D+ EBA1 6ED9 D+ I W@ D+
(SHA1) LOOP
D% 59 D% 40 DO
B" D@ C" D@ DAND
B" D@ D" D@ DAND DOR
C" D@ D" D@ DAND DOR
E" D@ D+ BCDC 8F1B D+ I W@ D+
(SHA1) LOOP
D% 79 D% 60 DO
B" D@ C" D@ DXOR D" D@ DXOR
E" D@ D+ C1D6 CA62 D+ I W@ D+
(SHA1) LOOP ;

: MD+   \ (  --  )
H          D@ MD          D@ D+ MD          D!
H     4 + D@ MD     4 + D@ D+ MD     4 + D!
H     8 + D@ MD     8 + D@ D+ MD     8 + D!
H D% 12 + D@ MD D% 12 + D@ D+ MD D% 12 + D!
H D% 16 + D@ MD D% 16 + D@ D+ MD D% 16 + D! ;

\ test 1 -----------------------------------

TABLE TEST-DATA1
61 C, 62 C, 63 C, 80 C, 00 C, 00 C, 00 C, 00 C,
00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 18 C,
```

```
110   : LOAD1
111   H-INIT     H  D% 20 CMOVE
112   H-INIT     MD D% 20 CMOVE
113   TEST-DATA1 M  D% 64 CMOVE ;
114
115   : TEST1
116   LOAD1 FILL-WH SHA1 MD+
117   CR ." Soll="
118   CR ." A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D"
119   ;
120
121   \ test 2 -----------------------------------
122
123   TABLE TEST-DATA2
124   61 C, 62 C, 63 C, 64 C, 62 C, 63 C, 64 C, 65 C,
125   63 C, 64 C, 65 C, 66 C, 64 C, 65 C, 66 C, 67 C,
126   65 C, 66 C, 67 C, 68 C, 66 C, 67 C, 68 C, 69 C,
127   67 C, 68 C, 69 C, 6A C, 68 C, 69 C, 6A C, 6B C,
128   69 C, 6A C, 6B C, 6C C, 6A C, 6B C, 6C C, 6D C,
129   6B C, 6C C, 6D C, 6E C, 6C C, 6D C, 6E C, 6F C,
130   6D C, 6E C, 6F C, 70 C, 6E C, 6F C, 70 C, 71 C,
131   80 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
132
133   TABLE TEST-DATA3
134   00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
135   00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
136   00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
137   00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
138   00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
139   00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
140   00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 00 C,
141   00 C, 00 C, 00 C, 00 C, 00 C, 00 C, 01 C, C0 C,
142
143   : LOAD2a
144   H-INIT     H  D% 20 CMOVE
145   H-INIT     MD D% 20 CMOVE
146   TEST-DATA2 M  D% 64 CMOVE ;
147
148   : LOAD2b
149   MD         H  D% 20 CMOVE
150   TEST-DATA3 M  D% 64 CMOVE ;
151
152   : TEST2
153   LOAD2a FILL-WH SHA1 MD+ MD.
154   LOAD2b FILL-WH SHA1 MD+ MD.
155   CR ." Soll="
156   CR ." F4286818 C37B27AE 0408F581 84677148 4A566572"
157   CR ." 84983E44 1C3BD26E BAAE4AA1 F95129E5 E54670F1" ;
```

Build Your Own Enigma Cipher Machine – Alan Turing not included

Abbildung 10: Der Enigma–Nachbau Mark 4

Nearly a century after its invention, the electromechanical Enigma cipher machine still strikes a deep chord among the digerati. Used by the German military to encode communications in the run–up to and during World War II, the Enigma has achieved a mythic quality in computing history — the Medusa slain by the hero Turing with the new weapon of digital logic.

Consequently, original Enigma machines are now collector's items that sell for tens of thousands of dollars. Even replicas are pricey. So the only alternative for those wishing to get to grips with this machine — and to better understand the mathematical, engineering, and operational feats that defeated it — has been to use one of a number of software emulators. But now there's a middle ground: a hardware kit that duplicates the physical operation of the Enigma's keyboard, display, and plugboard while replacing the rotating metal discs at the machine's heart with an Arduino Mega microcontroller.

The kit — called the Enigma Mark 4 — was created by S&T Geotronics as an open–source project with development funded by a Kickstarter campaign. ... (Stephen Cass, Posted 19 Dec 2014 | 20:00 GMT)

http://spectrum.ieee.org/geek-life/hands-on/build-your-own-enigma-cipher-machine

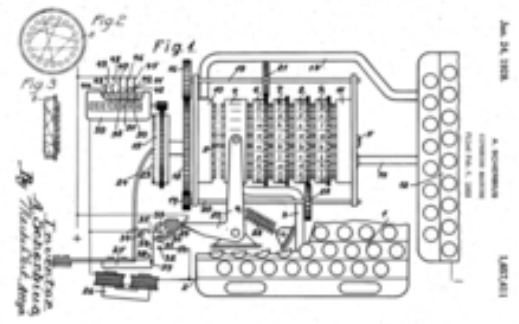


Abbildung 11: Zeichnung aus dem Patent US1657411: Ciphering Machine. Angemeldet am 6. Februar 1923, Erfinder: Arthur Scherbius. (Wikipedia.de)

Weiter auf Seite 29 ...

Mecrisp–Ice

Matthias Koch

Im Laufe des letzten Jahres ist in der programmierbaren Logik viel ins Rollen gekommen: Clifford Wolf aus Österreich und seinem Team ist es gelungen, den Bitstream der Lattice iCE40 FPGAs zu verstehen und freie Werkzeuge für die Entwicklung von Logik zu schreiben. Das ist ein monumentales Werk und eine wegweisende Pioniertat, wo ich nur sagen kann: Hut ab! Wer schon einmal mit kommerziellen FPGA–Werkzeugen gerungen hat, wird sehr angenehm überrascht sein, wie einfach und elegant es dadurch geworden ist, mit Verilog den Bitstream zu erzeugen.

Nachdem James Bowman, der für seinen eleganten J1–Prozessor bekannt ist, von den neuen Möglichkeiten wusste, dauerte es nicht mehr lange, bis er seinen J1a auf einem Icestick am Laufen hatte. Ich war als Tester von Anfang an mit dabei und konnte einige Ideen, wie die Unicode–Unterstützung, beitragen. Doch während James in seinem Swapforth vor allem Minimalismus und vollständige ANS–Konformität anstrebte, lag mein Augenmerk mehr auf einer schnellen und komfortablen Implementation, die gut zu den anderen Geschwistern in der Mecrisp–Familie passt und natürlich auch Optimierungen, wie Tail–Call, Inlining und Konstantenfaltung, beherrscht. An dieser Stelle möchte ich mich ganz herzlich bei allen Beteiligten für die großartigen Grundlagen bedanken, die Mecrisp–Ice erst möglich gemacht haben. Die besondere Abstammung ist auch der Grund dafür, dass Mecrisp–Ice ganz im Sinne von James Bowman unter der besonders freigiebigen BSD–Lizenz steht.

Eine kleine Einführung in FPGAs

Doch nun noch einmal einen Schritt zurück. Es soll eine kleine Einführung werden, die alle mitnimmt in die Welt der FPGAs, was „Field Programmable Gate Array“ bedeutet. Bastler der alten Schule werden sich noch daran erinnern, wie sie grundlegende Gatter in TTL–Chips mit vielen Drähten zu ausgefeilten Logikschaltungen verbunden haben. Welche Funktion dabei entsteht, hängt vor allem von der Art der Verdrahtung ab — so kann aus einzelnen Gattern ein Zähler, ein Addierer oder aus vielen Gattern auch ein Prozessor werden.

In einem FPGA sind nun ganz viele Gatter zusammen mit einem „Verdrahtungsgitter“ auf einem Chip versammelt. Über kleine Schalter kann gewählt werden, wie die Gatter ans Gitter geklemmt und wie die Gitterleitungen untereinander verbunden werden sollen. Soweit alles klar? Prinzipiell bestimmt das Schaltermuster für die „Verbindungsstellen“ damit, welche Schaltung im FPGA gebaut werden soll, genau wie die Drähte zwischen TTL–Chips auf einer Steckplatine. Nur die Größenordnung ist eine ganz andere: Wo früher gerade mal ein paar grundlegende Funktionen und ein Büschel Drähte Platz hatten, können nun tausende von Gattern und Verbindungsleitungen auf einem Chip untergebracht werden.

Und genau dieses „Schaltermuster“ ist es, welches in der Vergangenheit so viele Schwierigkeiten bereitet hat. Der genaue innere Aufbau des Verdrahtungsgitters war uns unbekannt und damit bestand keine Möglichkeit, mit freien Werkzeugen einen solchen sogenannten Bitstream zu erzeugen, welchen der FPGA beim Start benötigt, um seine Funktion festzulegen. Clifford ist es in mühevoller Kleinarbeit[1] gelungen, das Muster der Verdrahtungsleitungen im Gitter zu bestimmen und die Lage sämtlicher Verbindungsstellen zu enträtseln.

Prinzipiell wäre es mit diesem Wissen bereits möglich, eine Schaltung durch manuelles Setzen einzelner Verbindungsstellen zu entwickeln, doch bei der schieren Zahl von Gattern und Gitterleitungen wäre das ein äußerst mühseliges Unterfangen.

Deshalb gibt es Logikbeschreibungssprachen wie Verilog, die es in abstrakter Weise erlauben, die gewünschte Logikfunktion zu beschreiben — wobei hier nicht nur Grundlegendes wie XOR–Gatter, Inverter und Flipflops zur Verfügung steht, sondern auch große Sachen wie breite Register und Multiplizierer bequem spezifiziert werden können. Die Aufgabe eines „FPGA–Compilers“ (Flow genannt) ist es nun, in dem Synthese–Schritt, den *Yosys* erledigt, aus der abstrakten Beschreibung der Logik einen Schaltplan mit den im FPGA zur Verfügung stehenden einfachen Logikfunktionen zu entwickeln und dann eine Möglichkeit zu finden, diesen Schaltplan durch geschickte Ausnutzung der Gitterleitungen im FPGA nachzubauen, was Aufgabe des Place–and–Route–Schritts ist, welcher von *Arachne–PNR* erledigt wird. Dabei ist zu beachten, dass die Signale, die aus dem FPGA herein- oder herauskommen sollen, auch den gewünschten Pins zugeordnet werden (Constrains). Mit all diesen Vorbereitungen schließlich kann Icestorm das eigentliche „Schaltermuster“, den Bitstream, generieren.

Der Flexibilität wegen sind die „Gatter“ (LCs) im FPGA übrigens kleine Logiktabellen mit vier Eingängen und einem Ausgang, der durch ein Flipflop geschleift wird, so dass auch ausgefallene Wünsche umgesetzt werden können. Da Speicher sehr viele der programmierbaren Gatter verschlingen würde und häufig verwendet wird, sind fertige SRAM–Speicherblöcke (Block–RAMs) im FPGA vorhanden, die annähernd die gewohnten Adress- und

[1] die auf `http://www.clifford.at/icestorm/` in Augenschein genommen werden kann ...

Datenleitungen besitzen. Außerdem gibt es noch Spezialitäten, wie die analoge PLL, womit Frequenzen vervielfacht werden können, und die Eingabe– und Ausgabe–Leitungen, die im Gegensatz zum inneren Verbindungsgitter entsprechend starke Treibertransistoren für die Außenwelt und Schutz vor elektrostatischen Aufladungen benötigen.

Kurz zusammengefasst enthält der Bitstream von Mecrisp–Ice also den Schaltplan für einen kleinen Microcontroller und als Füllung für die RAM–Blöcke ein Forth, welches darauf läuft. Das Besondere ist, dass jeder, der mag, nicht nur das Forth, sondern auch den Aufbau des Microcontrollers selbst verändern kann!

Womit loslegen?

Im Moment läuft Mecrisp–Ice außer im Emulator auf drei verschiedenen Platinen, die ich hier kurz vorstellen möchte, um die Auswahl ein bisschen zu erleichtern.

Das Licht der Welt erblickte Mecrisp–Ice auf einem Lattice Icestick. Dieser kommt mit einer IrDA–Schnittstelle, 5 LEDs, 24 freien IO–Leitungen und ist handlich, preisgünstig und schön anzusehen. Ebenfalls mit einem iCE40–HX1K–FPGA ausgestattet ist das Nandland Go von Russell Merrick, welches mit zwei 7–Segment–Ziffern, 4 LEDs, 4 Tastern und einem VGA–Anschluss ausgestattet ist und 8 freie IO–Leitungen bietet. Für all jene, die nur mal einen Stackprozessor in einem FPGA ausprobieren möchten, sind beide gut geeignet. Einfach beide anschauen und zwischen IrDA und VGA wählen.

Wer allerdings gerne noch eigene Erweiterungen zum Befehlssatz hinzufügen möchte, Ideen für weitere Peripherie in Verilog ausprobieren mag oder große Ideen in Forth umsetzen will, für den ist der HX1K zu klein. Seine Gatter sind bereits bis zum Rand ausgefüllt und der Forth–Kern belegt 6,5 kb von den verfügbaren 8 kb. In diesem Fall sollte die Wahl auf das HX8K Breakout von Lattice fallen. Der HX8K–FPGA beinhaltet 16 kb RAM und viel mehr Gatter, so dass neben einem stärkeren Prozessor mit Barrel–Shiftern und Multiplikation in einem Takt für nützliche Erweiterungen und phantasievolle Ideen viel Raum bleibt. Das HX8K Breakout besitzt sehr viele freie IO–Leitungen, hat aber dafür außer acht Leuchtdioden gar keine Peripherie an Bord, so dass es für sich genommen nicht so spannend ist und es erst in Kombination mit eigenen Schaltungen sein volles Potential entfaltet.

Eine USB–Seriell–Terminalbrücke und die nötige Programmierhardware ist auf allen drei Platinen enthalten.

Immer noch unentschlossen? Dann lautet meine Empfehlung: Zwei Icesticks bestellen und Spaß mit einem Chat über IrDA oder Ledcomm haben!

Befehlssatz und Besonderheiten des J1a

Der J1a ist ein kleiner 16–Bit–Prozessor, dessen Instruktionen alle genau 16 Bits lang sind und zur Ausführung stets genau einen Takt benötigen, da die Stacks vom Speicher separat implementiert sind.

Fünf Typen von Instruktionen gibt es, die sogleich noch einzeln näher beschrieben werden sollen:

```
1--- ---- ---- ----   Literal
000- ---- ---- ----   Unbedingter Sprung
001- ---- ---- ----   Bedingter Sprung
010- ---- ---- ----   Call
011b bbbb Rsss rrdd   ALU
```

Der Literal–Befehl beinhaltet eine 15–Bit–Konstante und legt diese auf den Stack — so können alle positiven Konstanten in einem Takt bereitgelegt werden. Für negative Konstanten ist es nötig, das oberste Stackelement zu invertieren, was von einen Befehl des ALU–Typs erledigt wird. Doch dazu später mehr ! Die unbedingten und bedingten Sprünge sind aus klassischen Forths als `Branch` und `0-Branch` bekannt und verhalten sich auch genauso. `Call` ist ein Unterprogrammaufruf, der eine Rücksprungadresse auf den Return–Stack legt und die Zieladresse anspringt. Wichtig ist die Beobachtung, dass die Zieladresse in allen drei Fällen „nur“ 13 Bits Platz hat, was dazu führt, dass der J1a höchstens 16 kb Speicher für seine Programme sinnvoll nutzen kann.

Der ALU–Befehl schließlich hat mehrere Felder, die hier einmal kurz angesprochen werden sollen: Die b–Bits geben an, ob und wie die Recheneinheit aus TOS und NOS etwas Neues berechnen soll. Diese Worte stehen zur Verfügung: `+ - and or xor invert 2* 2/ = < u<` , im HX8K auch `lshift rshift arshift um*-low um*-high 1+ 1-` . Ebenso gibt es besondere Befehle, die die Stacktiefe zurückgeben und gemeinsam mit den anderen Feldern Stackjonglage und IO–Zugriffe ermöglichen. Das R–Bit kann für einen Rücksprung verwendet werden, so dass die letzte Rechenoperation in einer Definition auch zugleich ihr Ende sein kann. Das führt dazu, dass das Aufrufen einer Definition (Dank der Tail–Call–Optimierung) insgesamt nur einen Takt benötigt, wenn die Definition nicht gerade mit einem Literal oder einem Returnstack–Jongleur endet. Die s–Bits steuern Speicher– und IO–Zugriffe und erlauben es, Interrupts ein- und auszuschalten. Die beiden r–Bits und d–Bits schließlich bestimmen, ob die Stacks so bleiben, wachsen oder schrumpfen sollen.

Nicht alle Kombinationen ergeben immer sinnvolle Funktionen, aber auch wenn Mecrisp–Ice so raffinierte Optimierungen nicht selbst durchführen kann, ist es im Kern möglich, beispielsweise die Sequenz `2dup =` in einem einzigen Befehl auszuführen.

Da der J1 ein reiner 16–Bit–Prozessor ist, spielt das letzte Bit in einer Rücksprungadresse auf dem Returnstack keine Rolle — in der Tat springt der J1a an die gleiche Stelle, egal, ob dieses 1 oder 0 ist. Dieses Verhalten wird ausgenutzt, damit der Programmierer prüfen kann, ob Interrupts aktiv sind: Der Call–Befehl setzt das letzte Bit auf dem Return–Stack, wenn die Interrupts aktiviert gewesen sind.

	HX1K	HX8K	
RAM für Kern, Dictionary und Daten	$0000	$0000	Anfangsadresse nach einem Reset
	...	...	
	$1FFE	$3FFE	Wird bei einem Interrupt angesprungen
Fetch–Adressbereich	$2000	$4000	Anspringen dieser Adresse bewirkt "$0000 @ exit"
	...	...	
	$3FFE	$7FFE	"$1FFE @ exit" / "$3FFE @ exit"
(IO–Ports sind in einem separaten Adressraum)			

Tabelle 1: Memory Map

Byte–Zugriffe, die für Strings unbedingt benötigt werden, also `c@` und `c!` , werden vom Prozessor gar nicht unterstützt und müssen entsprechend langsam in Software aus den entsprechenden 16–Bit–Speicherzugriffen zusammengebaut werden.

Schließlich noch eine Besonderheit beim Lesen von Speicher: Leider haben die iCE40–FPGAs kein Dual–Port–RAM, sondern nur einen Lese–Port und einen Schreib–Port in jeden RAM–Block. Der Lese–Port ist für das Holen der Befehle aus dem Speicher zuständig und damit bereits belegt. Der Einfachheit des Prozessors zuliebe müssen aber alle Befehle in exakt einem Takt ausgeführt werden. Wie soll so ein Wert aus dem Speicher gelesen werden? Der Trick besteht darin, einen Call zu einer hohen Adresse oberhalb des tatsächlich vorhandenen Speichers durchzuführen. Der Speicher sieht davon nichts — die hohen Adressen werden nicht dekodiert — aber dieser Spezialfall wird von besonderer Logik im J1a erkannt, was dazu führt, dass der vom Speicher gelieferte „Befehl“ dann nicht ausgeführt wird, sondern unverändert auf den Daten–Stack gelegt und ein Rücksprung eingeleitet wird. So wird aus einem Fetch im J1a also ein High–Call. Im HX1K, wo mit 8 kb Speicher nur 12 Adressbits (Word–Adressierung!) belegt sind, kann ein Speicher–Fetch also über das 13. Adressbit in einem einzigen Call–Befehl kodiert werden. Im HX8K, wo alle 13 Adressbits mit den 16 kb Speicher bereits belegt sind, müssen für ein Speicher–Fetch leider zwei Befehle geschrieben werden: `Literal` und `Execute`. `Execute` hat übrigens keinen eigenen Befehl, kann aber dadurch erreicht werden, dass in einer Definition die gewünschte Adresse vor dem Rücksprung auf den Returnstack gelegt wird.

Zur Veranschaulichung zeigt Tabelle–1 eine kleine Speicherkarte aus der Sicht von Forth.

Interrupts selbst, die in James Variante des J1a gar nicht vorkommen, waren übrigens recht leicht zu implementieren: Im Falle eines Interrupts wird der Programmzähler nicht erhöht und anstelle des vom Speicher kommenden Opcodes wird ein fest verdrahteter Call–Befehl zur letzten Adresse im Speicher ($1FFE im HX1K, $3FFE im HX8K) ausgeführt, wo vom Programmierer ein entsprechender unbedingter Sprung oder ein Rücksprungbefehl ($608C) bereitgelegt werden kann. Das war es schon! Da der J1 außer dem Programmzähler, der dabei auf den Returnstack wandert, keinen inneren Zustand hat, der gesichert werden müsste, und Stacks von sich aus interruptfest sind, sind keine weiteren Besonderheiten zu beachten, sofern die verfügbare Stacktiefe ausreicht. Eine Interrupt–Sperrlogik, die beim Auslösen eines Interruptes sogleich folgende weitere Interrupts verhindert, gibt es übrigens nicht. Da in Mecrisp–Ice ein Timer die Interrupts erzeugt, reicht es, diesen so zu setzen, dass die Pausen zwischen den Interrupts lang genug sind. Würde es auch Pin–Interrupts geben, dann wäre allerdings eine Sperrlogik notwendig, damit es beispielsweise bei einem „wackelnden“ Signal nicht zu einem Returnstack–Überlauf käme.

Ein kleiner Kommentar noch zu den Stacks: Im J1a sind die Stacks als Schieberegister implementiert, die bei einem Stackunterlauf den festen Wert $55AA zurückliefern und einen Überlauf ignorieren. Ein „depth“ gibt es so nicht direkt, stattdessen läuft ein Zähler mit, der ausgelesen werden kann, aber bei Unter- und Überläufen entsprechend durcheinander kommt. Im HX1K sind beide Stacks 16 Elemente tief und es gibt aus Platzgründen nur einen Zähler für den Datenstack (depth), während im HX8K beide Stacks 32 Elemente tief sind und es auch einen Zähler für den Returnstack (rdepth) gibt.

Peripherie in Mecrisp–Ice

Die Ein- und Ausgabe funktioniert via `io@` und `io!` über einen separaten Adressraum, wobei zum Glück weniger Register unterzubringen waren, als Adressleitungen vorhanden waren. Damit reagiert jedes IO–Register auf „sein“ persönliches Bit — werden in einer IO–Adresse mehrere Bits gesetzt, so werden bei `io@` die Inhalte der verschiedenen Register „verodert“ und bei einem `io!` mehrere Register auf einmal auf den gleichen Wert gesetzt.

Welche Register es gibt, kann leicht verändert werden, ist je nach Platine verschieden und sollte im Einzelfall nachgeschlagen werden, doch ein paar wichtige Gemeinsamkeiten sollen hier beschrieben werden:

Die freien IO–Leitungen haben drei Register nach Art des MSP430: IN (nur lesbar), OUT und DIR (beide les- und schreibbar). Aus dem IN–Register kann jederzeit der tatsächliche aktuelle elektrische Zustand der Pins gelesen werden. Eine Eins an entsprechender Stelle im DIR–Register schaltet den betreffenden Pin auf

Ausgang, während eine Null (Startwert) den Pin als Eingang konfiguriert. Das OUT–Register schließlich enthält die Werte, die an den Pins ausgegeben werden sollten, die gerade Ausgänge sind. Die Register sind dabei völlig unabhängig voneinander, zum Beispiel bleiben die Werte im OUT–Register erhalten, auch, wenn zwischendurch die Pins Eingänge werden. Ebenso tauchen in dem IN–Register sowohl die von außen hereinkommenden Signale als auch die gerade selbst ausgegebenen Signale auf. Theoretisch sollten sich so auch Kurzschlüsse oder große kapazitive Lasten erkennen lassen: Ist ein Pin ein Ausgang und weicht sein Wert im IN–Register von dem gesetzten Wert im OUT–Register ab, so ist eine elektrische Schwierigkeit zu befürchten.

Der Timer schließlich zählt aufwärts, löst beim Überlaufen zurück zu null einen Interrupt aus (falls die Interrupts gerade aktiv gewesen sind, wofür die Worte `eint dint eint?` zuständig sind) und besteht aus einem einzigen Register: Dem aktuellen Timerstand. Wird dieses Register geschrieben, wird der Timer entsprechend gesetzt. Ein Setzen auf null löst übrigens keinen Interrupt aus — der Überlauf ist entscheidend, nicht der Nullwertl darin.

Ganz klein, aber nützlich mag ein frei laufender Ringoszillator sein, der als Zufallszahlengenerator fungiert. Eine serielle Schnittstelle ist natürlich auch mit dabei, doch es ist eine ganz einfache 115200–Baud–8N1–Implementation mit in Verilog fest vorgegebener Baudrate, deren Möglichkeiten von `emit emit? key` und `key?` vollständig abgedeckt werden.

Wer sich nun als findiger Leser denkt: Das ginge aber noch wesentlich raffinierter — dem sei in Erinnerung gerufen, dass der HX1K wirklich ein kleiner FPGA ist und schon fast aus allen Nähten platzt. Im HX8K dagegen bietet sich noch eine weite Wiese für kreative Ideen, die nach Wunsch bepflanzt werden kann. Die schönsten Blüten werden gesammelt!

Gegen die Vergesslichkeit

Mecrisp–Ice läuft vollständig im RAM der FPGAs, so dass alles, was wir schreiben, nicht für die Ewigkeit ist. Dennoch kann gegen die Vergesslichkeit etwas getan werden: Um dem FPGA beim Einschalten des Stromes sein „Schaltermuster“ zu servieren, enthalten alle der drei oben genannten Platinen einen kleinen SPI–Flash. Nachdem der FPGA den Bitstream erhalten hat, hat dieser Speicherbaustein Feierabend, wenn wir ihn nicht zusätzlich noch für andere Zwecke einspannen: Mit `LOAD SAVE` und `ERASE` , die alle eine Segmentnummer auf dem Stack erwarten, lässt sich nämlich der gesamte aktuelle Speicherinhalt in einen der Flash–Sektoren speichern und zurückholen. Der Bitstream selbst ist in Segment 0 (im HX8K in den Segmenten 0...2) und sollte nicht überschrieben werden, aber die darauf folgenden Sektoren sind frei und können verwendet werden. Der direkt nach dem Bitstream folgende Sektor hat dabei eine Besonderheit: Er wird, wenn verfügbar, vom Kern direkt nach dem Start von Forth automatisch geladen.

Wer von den anderen Geschwistern aus der Mecrisp–Familie kommt, wird gewöhnt sein, dass die Variablen ihren Initialisierungswert nach einem Reset zurückerhalten. Hier aber wird einfach nur ein Abbild des RAMs geschrieben, also enthalten die Variablen beim Laden den Wert, der zum Zeitpunkt des Speicherns in ihnen enthalten gewesen ist. Es gibt eine Variable „init“, in die entweder eine Null oder die Adresse der nach dem Laden des Sektors automatisch auszuführenden Definition gelegt werden sollte.

Beide Mechanismen zusammen erlauben es so, fertige Programme ganz von selbst anlaufen zu lasen.

Da nach einem Reset der letzte Speicherinhalt erhalten bleibt und der Prozessor einfach nur wieder von vorne anfängt, kann es übrigens nützlich sein, den Urzustand des Kernes mit `1 SAVE` (HX8K: `3 SAVE`) so zu sichern, dass nach einem Absturz und Reset stets ein frisches Kernabbild automatisch neu geladen wird.

Viel Spaß!

Für die neugierigen Entdecker ist bereits für die ersten Streifzüge ein Disassembler (`see`) und eine ausführtliche Wortliste mit sämtlichen Details der Dictionarystruktur (`insight`) enthalten, die aber nach einem `new` verschwinden, um Platz für tolle Ideen zu schaffen. Ein Durchstöbern der Beispiele ist als nächster Schritt angebracht — viel Spaß beim Entdecken!

Link

Zu Finden ist Mecrisp–Ice unter
`mecrisp.sourceforge.net`

Clock Works 1 — Die kleine Uhr

Erich Wälde

Die Zeit spielt im heutigen Leben bekanntlich eine große Rolle. So gibt es allerlei wunderliche Erfindungen, Uhren genannt, um den Verlauf des Tages in zeitlich kleinere Häppchen zu zerteilen, und es wird erwartet, dass ein jeder sich daran orientieren möge. So kommt es, dass es schon sehr lange Uhren in vielerlei Technologie gibt. Da ich mich gerne mit Mikrokontrollern und Forth beschäftige, ist es nicht verwunderlich, dass ich mich erneut *damit befasse [5], [6].*

Auslöser war der Wunsch eines Amateurfunkfreundes, der gerne eine Uhr hätte, die sich selbst stellt (dem DCF77 Signal lauschend), und die aber die Zeit in UT [2] anzeigt und nicht in MEZ/MESZ. Insbesondere soll sie zur Zeit der Sommerzeit–Umstellung keinen Schluckauf kriegen und gerne große Ziffern besitzen.

Und wenn man schon mal dabei ist, dann könnte es vielleicht eine Uhr werden, an die man verschiedenerlei Anzeigen anschließen kann, die die Zeit in UT, MEZ/MESZ und vielleicht auch in Unix Epochensekunden [1] oder in swatch beats [3] anzeigt. Ach und ein Wecker soll's auch werden. Und der soll vielleicht mehr können, als nur zu piepsen. Schaltsekunden korrekt berücksichtigen wär doch auch sexy, 'ne Sternzeit–Uhr auch, und wenn man das DCF77–Signal aufnehmen kann, dann könnte man die Drift des Uhrenquarzes doch gleich noch mitbestimmen — nach noch ein paar Dutzend Ideen und einigen Stunden mit halbfertigen Schaltungen und Programm–Schnipseln, hab ich's dann etwas entnervt in die Ecke gelegt. Und nur erschwerend kommt hinzu, dass kein Mensch noch *eine Uhr braucht — wir sind alle hervorragend ausgerüstet.*

Aber wie das mal so ist mit den Häckern und Bästlern — es lässt einem am Ende ja doch keine Ruhe. Also neuer Plan: erst mal eine minimale Angelegenheit, die gerade so als Uhr durchgeht. Darum soll es in diesem Artikel gehen.

Die *kleine* Uhr

Die erste Version dieses Projekts soll eine Uhr mit folgenden Eigenschaften sein:

1. Die Uhr läuft *frei* und bezieht den Uhrentakt aus einem zusätzlichen 32768 Hz–Uhren–Quarz
2. Die Anzeige ist sehr simpel, lediglich ein 2x8–Zeichen–LCDisplay wird angeschlossen
3. Die Sommerzeit–Umstellung wird ignoriert
4. Das Stellen der Uhr erfolgt über die serielle Schnittstelle — es ist eben doch eine Geek–Uhr!
5. Die Uhr soll die Zeit nicht vergessen, wenn der Strom mal aus ist. Es gibt eine zusätzliche, batteriegepufferte Hardware–Uhr (RTC) als Gedächtnis. Streng genommen haben wir jetzt schon zwei Uhren!
6. Außerdem hat das Prototypenboard 4 LEDs, die sich für mancherlei (Status–)Anzeigen eignen.

Die Punkte 3 und 4 reduzieren den Aufwand. Punkt 5 ist schon eher Luxus, hilft aber der Faulheit, von der man als Programmierer auch ein gesundes Maß besitzen sollte. Als Uhr ist das gerade eben so brauchbar. Allerdings ist das keine *minimale* Uhr, für die man noch allerhand sparen könnte.

Räderwerk

Was bei einer mechanischen Uhr das Räderwerk ist, das sind im Programm diverse Zähler, die von einer Taktquelle mit bekannter Frequenz die gewünschten Zeitintervalle wie Minuten oder Tage ableiten.

Quarz 32768/s Als Takt–Quelle habe ich mich für einen Uhrenquarz mit einer Frequenz von 32768 Hz entschieden. Der `atmega644p`–Kontroller hat zwei Pins (PC[6,7]), die einen solchen Quarz treiben können, und dessen Signal dann als Taktquelle an den Timer/Counter2 weitergegeben wird. Vorläufig gehe ich davon aus, dass die Frequenz des Uhrenquarzes ausreichend genau ist.

Tick 128/s Der Timer/Counter2 ist ein 8–Bit–Zähler. Er zählt bei jedem Takt der Quelle zu dem Wert in seinem Zählregister 1 dazu. Bei 255 angekommen, führt das nächste Taktsignal nach 256 Takten zum Überlauf und das Zählregister zeigt den Wert 0. Immer wenn dieser Überlauf stattfindet, wird ein Interrupt ausgelöst und die registrierte Interrupt–Service–Routine aufgerufen. Der Überlauf findet 32768/256 = 128 mal pro Sekunde statt. Diese Zeitspanne von 1/128 s nenne ich `tick`.

Sekunde 1/s Nach 128 `ticks` wird ein Bit gesetzt (`sec_over`), welches anzeigt, dass eine Sekunde vergangen ist. Das Hauptprogramm ruft dann die Funktion `timeup` auf, welches die Buchhaltung durchführt und die weiteren Zähler aktualisiert.

Minute 1/60 s

Stunde 1/3600 s Die Zähler von Minute und Stunde sind einfach zu aktualisieren. 60 Sekunden ergeben eine Minute, 60 Minuten ergeben eine Stunde. 24 Stunden ergeben bekanntlich einen Tag. Gelegentlich vorkommende Schaltsekunden werden mal locker ignoriert.

Tag 1/86400 s = 1/d Die Zähler des Kalenders (Tag, Monat, Jahr) sind etwas aufwändiger zu verwalten, weil die Monate unterschiedlich lang sind. Auch die korrekte Behandlung des Schaltjahres wird verlangt.

Monat 1/28..31 d Die Länge der Monate wird in einer Liste (`tu.lastday_of_month`) nachgeschaut. Die Länge vom Februar wird in Schaltjahren entsprechend korrigiert (Funktion `length_of_month`).

Jahr 1/365..366 d Die Bestimmung des Schaltjahres ist wieder recht einfach (Funktion `leap_year?`).

Andere Zeitrechnungen können hier entsprechend eingehängt werden. Die zur Sekunde aufgerufene Funktion könnte die Laufzeit der Uhr (`uptime`) zählen. Unix–Epochensekunden wären ebenfalls hier zu aktualisieren. Swatch Beats (ein Tag ist in 1000 Beats eingeteilt) und die Sternzeit (23.9345 Stunden) benötigen andere Teiler.

Die Hauptschleife des Programms wertet die zu den Zählern gehörenden Bits (oder Flags) `sec_over`, `min_over`, `hour_over` etc. aus und ruft für jedes dieser Zeitintervalle eine entsprechende Funktion auf. Beispielsweise wird in der Funktion `job.sec` die Sekundenanzeige auf dem Display aktualsiert, in der Funktion `job.min` der gesamte Displayinhalt.

Das Programm

Das komplette Programm ist recht umfangreich, allerdings sind die einzelnen Teile nicht speziell schwierig. Diese einzelnen Aspekte oder Module werden im folgenden Text erklärt. Mit den bei mir üblicherweise benutzten Bibliotheksmodulen von `AmForth` kommen da etwas über 1000 Zeilen bzw. etwa 3000 Worte zusammen (ohne Kommentare).

amforth 6.3

Seit kurzem ist `AmForth` in der Version 6.3 aktuell. Der Kontroller wird über einen Baud–Raten–Quarz mit der Frequenz 11.059200 MHz betrieben, die Baudrate der seriellen Schnittstelle beträgt 115200.

```
.equ F_CPU = 11059200
.set BAUD=115200
```

Ich bin außerdem ein Freund davon, Groß– und Kleinschreibung zu unterscheiden, und setze daher den Eintrag

```
.set WANT_IGNORECASE = 0
```

Diese drei Zeilen beinhalten alle Änderungen an der Datei `appl/template/template.asm`, die als Vorlage diente. Aus Gewohnheit verwende ich ab und zu das Wort `0<>`, welches zusätzlich in die einkompilierte Wortliste kommt (`dict_appl.inc`):

```
.include "words/notequalzero.asm"
```

Damit ist `AmForth` vollständig konfiguriert.

stationID

Meine Kontroller bekommen in der Regel eine Stationsnummer, und diese will ich auch im Prompt sehen. Bislang musste ich dafür die entsprechende Funktion im Assembler–Teil von `AmForth` verändern. Das ist jetzt nicht mehr nötig, weil diese Funktionen `deferred` sind.

```
include lib-avr8/forth2012/core/avr-values.frt

$0061 Evalue stationID

\ --- stationID prompt
\     .ready is a deferred word since AmForth 6.3
: .stationID_ready
  cr
  [char] ~ emit
  base @  $10 base !  stationID 2 u0.r  base !
  [char] > emit space
;
: +stationID
  ['] stationID_ready is ready
;
```

Könnte sein, dass die Uhr einmal am RS485–Bus wohnt, und da braucht sie eine Adresse, die `stationID`. Bei dieser Version ist das aber nicht notwendig und lediglich ein *Yes, we can!*–Feature.

Ansteuerung Status LEDs

Bislang haben meine Kontroller–Stationen 4 Status–LEDS. Die benutze ich, um beim Start hübsch zu blinken, damit ich sehe, dass sich etwas rührt. Eine minimale Uhr, zumal eine Uhr mit einem LCDisplay braucht das eigentlich nicht.

```
include lib-avr8/bitnames.frt
include atmega644p.fs

PORTB 0 portpin: lederr
PORTB 1 portpin: ledsec
PORTB 2 portpin: ledsensor
PORTB 3 portpin: leddata

include ewlib/leds.fs

: init
  +leds #200 ms leds-intro
  \ ...
;
```

Die LEDs werden dann mit den Worten `on` und `off` ein- und ausgeschaltet, z.B.:

```
lederr on
lederr off
```

Ansteuerung LCD, 4bit mode

Als Anzeige habe ich ein billiges LCDisplay mit 2x8 Zeichen eingesetzt [7]. Das Display wird mit einem `hd44780`–Kontroller angesteuert, allerdings ist der 4–bit–Modus zu verwenden, weil die Signale `D0` bis `D3` gar nicht verfügbar sind. Diese Angelegenheit hat mich länger aufgehalten, als erwartet. Aber das Wilde Weite Webernetz hält auch für diesen Fall Erleuchtung bereit. Die Initialisierungsequenz, um den Kontroller in den 4–bit–Modus zu bringen, ist etwas unoffensichtlich:

```
: hd44780-4bit-init
    _hd44780-rs pin_output
    _hd44780-rw pin_output
    _hd44780-en pin_output

    hd44780-command-mode
                              #15 ms
    $03 hd44780-write-nibble   #5 ms
    $03 hd44780-write-nibble     1ms
    $03 hd44780-write-nibble     1ms
```

```
    $02 hd44780-write-nibble     1ms
    \ we reach 4-bit mode here
    \ ...
;
```

Aber danach kann man das LCDisplay fertig initialisieren und Text anzeigen.

Ansteuerung Display

Hatte ich schon erwähnt, dass dieses Projekt vorübergehend eingestellt wurde, weil es fröhlich in alle denkbaren Richtungen explodiert ist? Eine der Richtungen war diese: man könnte doch verschiedene Anzeigen für die Uhrzeit realisieren. Anzeige von Uhrzeit und Datum würde über `deferred` Funktionen realisiert, so dass man für eine andere Anzeige das Programm nicht weiter ändern muss, als ein paar Funktionszeiger umzubiegen. Ich habe mich für drei Funktionen entschieden, die die Anzeige ansteuern sollen, sowie `init`. Das Präfix `cd` steht für `clock.display`

```
Edefer cd.init
Edefer cd.date
Edefer cd.time
Edefer cd.time.S
```

Die Funktion `cd.time.S` aktualisiert nur die Sekunden. Das ist bei LCDisplays evtl. sinnvoll. Ob das bei anderen Anzeigen auch sinnvoll ist, wird sich zeigen. Vielleicht sollte `cd.time` selbst herausfinden, wieviele Ziffern zu ändern sind und das dann heimlich optimieren. Das kann aber auch ein ungünstiges Vorgehen sein.

TWI Bus, RTC

Das *Gedächtnis* der Uhr ist eine zweite Uhr (RTC), die weiter mit Strom versorgt wird, wenn das Kontrollerboard stromlos ist. Die RTC ist über den i2c–Bus erreichbar. Details hierzu finden sich in [8] und [6].

Es gibt im Programm Funktionen, um die i2c–Uhr zu lesen und zu schreiben.

```
: rtc.get ( -- S/100 S M H d m Y ) ... ;
: rtc.set ( Y m d H M S S/100 -- ) ... ;
```

Zu den Werten, die die RTC Uhr immer aktuell liefert, habe ich zusätzlich den ganzen Wert des Jahres (2 Byte) im EEPROM der RTC abgelegt. Die Uhr selbst liefert nur die letzten beiden Ziffern.

timeup

Wie oben angedeutet, und wie detailiert in [5] und [6] beschrieben: Die Hauptschleife von Task 2 (`run-job`) läuft, solange nichts anderes zu tun ist. Ist ein `tick` vergangen, dann wird `job.tick` aufgerufen, was 128 mal pro Sekunde der Fall ist. Ist außerdem eine Sekunde vergangen, dann wird zusätzlich `timeup` aufgerufen. Diese Funktion aktualisiert die Zähler der Zeit und setzt Bit–Flags für vollendete Zeitintervalle. In weiteren Durchläufen der Schleife werden die Funktionen `job.sec`, `job.min` bis `job.year` aufgerufen, sofern die entsprechenden Bits gesetzt sind.

Damit wird sowohl die Buchhaltung der Zeit bewerkstelligt, als auch eine Möglichkeit geschaffen, periodisch gewisse Arbeiten zu erledigen.

Phasen–Akkumulator

In älteren Implementierungen der `timeup`–Uhr habe ich keine Möglichkeit vorgesehen, die Drift der Uhr irgendwie ausgleichen zu können. Hat man einen sehr genauen Uhrenquarz, dann ist das ja auch nicht nötig. Billigst–Quarze können aber schon spürbar von der Sollfrequenz abweichen. Dazu eine kurze Abschätzung: Ein Jahr hat $86400 \cdot 365 = 31536000$ Sekunden. Weicht die Quarzfrequenz um 10^{-4} von der Sollfrequenz ab, dann sind das immerhin 3154 Sekunden oder fast eine Stunde Abweichung im Jahr. Es wäre doch schick, wenn man eine solche, als konstant angenommene, Abweichung mit schlauen Maßnahmen ausgleichen könnte.

Michael Kalus hat in der VD [9] gezeigt, wie man das machen kann. Das Verfahren ist allgemein bekannt als *numerically controlled oscillator*, der Artikel in der Englisch–sprachigen Wikipedia erklärt das Verfahren [4].

Anstatt nach Ablauf eines `ticks` stur deren Anzahl um 1 zu erhöhen, verschieben wir die Angelegenheit mal um ein paar Bits nach links: Angenommen, wir haben zwei Variablen `f_phase` und `f_control`, welche 16 Bit breit sind. Wir starten mit den Werten

```
f_phase = 0
f_control = [Konstante]
```

Wir wollen nach 128 (2^7) Iterationen von

```
f_phase = f_phase + f_control
```

erreichen, dass `f_phase` überläuft, d.h. 2^{16} erreicht. Also gilt

```
f_control = 2^16 / 2^7 = 2^9 = 512
```

Soweit ist erst mal alles wie erwünscht. Die folgende Tabelle zeigt den Inhalt von `f_phase` und die Anzahl der erzeugten overflows (d.h. vollständige Sekundenintervalle) als Funktion der fortlaufenden `ticks` (Spalten 1 bis 3).

Wenn wir jetzt probehalber den Wert von `f_control` um 5 erhöhen, dann erfolgt der erste overflow bereits nach 127 ticks und nicht erst nach 128 (Spalten 4 und 5).

Allerdings bleibt ein Rest von 123 in `f_phase` übrig. Bei jedem Überlauf (Sekunde) kommen weitere 123 hinzu. Übersteigt die Summe dieser Reste den Wert $517-127 = 390$ dann erfolgt der nächste overflow schon nach 126 ticks: Sichtbar in der Tabelle bei tick 634 statt 635!

f_control	512		517	
system-tick	f_phase	over flow	f_phase	over flow
0	0	0	0	0
1	512	0	517	0
2	1024	0	1034	0
3	1536	0	1551	0
126	64512	0	65142	0
127	65024	0	123	1
128	0	1	640	1
253	64000	1	65265	1
254	64512	1	246	2
255	65024	1	763	2
256	0	2	1280	2
379	62976	2	64871	2
380	63488	2	65388	2
381	64000	2	369	3
382	64512	2	886	3
383	65024	2	1403	3
384	0	3	1920	3
506	62464	3	64994	3
507	62976	3	65511	3
508	63488	3	492	4
509	64000	3	1009	4
510	64512	3	1526	4
511	65024	3	2043	4
512	0	4	2560	4
633	61952	4	65117	4
634	62464	4	98	5
635	62976	4	615	5
636	63488	4	1132	5
637	64000	4	1649	5
638	64512	4	2166	5
639	65024	4	2683	5
640	0	5	3200	5

Es gibt also Sekunden, welche 127 ticks dauern, und Sekunden, die 126 ticks dauern. Eigentlich ist das furchtbar, es gibt verschieden lange Sekunden in diesem Programm. Aber über längere Zeit ergibt sich eine *mittlere* Sekunde. Der Wert von `f_control` wird so gewählt, dass bei abweichender Frequenz des Uhrenquarzes die *mittlere* Sekunde der `timeup`–Uhr mit der wahren Sekunde übereinstimmt. Die `timeup`–Uhr *geht dann mit der Zeit.*

Die Verwendung von 16–Bit–Variablen erlaubt keine sehr feine Justage. Im Programm verwende ich 32–Bit–Variablen. Interessanterweise ist $2^{32}/128 = 33554432$ ganz in der Nähe der Anzahl der Sekunden in einem Jahr (31536000). Das bedeutet, dass man theoretisch eine Abweichung von ca. 1 Sekunde pro Jahr kompensieren kann. Allerdings spielen dann vermutlich andere Effekte wie Temperaturschwankungen und Schwankungen in der Spannungsversorgung eine viel größere Rolle. Die zugehörigen Forth–Worte sollten sich jetzt fast von alleine erklären:

```
\ 2015-10-10 EW   ewlib/clock_tick_v2.5.fs

\ default: 2^32/2^7 = 2^25 = $2000000
\                          = #33554432
$2000000.  2Evalue    EE.fcontrol \ eeprom
           2variable  ct.fcontrol
           2variable  ct.phase

variable ct.flags
: tickover? ( -- t/f )
  ct.flags @ $0001 and ;
: tick.done ( -- )
  ct.flags @ $0001 invert and ct.flags ! ;
: secover?  ( -- t/f )
  ct.flags @ $0002 and ;
: sec.done  ( -- )
  ct.flags @ $0002 invert and ct.flags ! ;

\ overflow2 interupt service routine
\ increment tick
: tick_isr
  \ call to this isr means: tick.over
  ct.flags @ $0001 or ct.flags !

  \ check for overflow!
  \ compare high bytes, whether
  \ phase_{i+1} u< phase_{i}
  ct.phase    2@      dup >r
  ct.fcontrol 2@ d+   dup >r
  ct.phase 2!
  r> r> u< if
    \ overflow means: second.over
    ct.flags @ $0002 or ct.flags !
  then
;
\ enable ticks
\ crystal:   32768 /sec
\ clock src: 32768 /sec
\ overflow:  32768/256 = 128 /sec
\             =^= 7.8125 milli-sec ticks
: +ticks
    EE.fcontrol ct.fcontrol 2!
    0.          ct.phase    2!
    0           ct.flags     !
    \  clock_ts2/1
    [ %001
    ] literal TCCR2B c!
    \ source: 32 kiHz
    ASSR_AS2 ASSR  c!
    \ register interrupt
    ['] tick_isr TIMER2_OVFAddr int!
    \ enable timer2 interupt
    TIMSK2 c@ $01 or TIMSK2 c!
;
```

timeup jobs

Wie mehrfach erwähnt, werden periodische Arbeiten in Funktionen erledigt, welche nach Ablauf des zugehörigen Intervalls aufgerufen werden.

job.tick blinkt mit einer Status–LED, quasi eine Debug–Anzeige. Hier kann in der Zukunft die Abtastung des DCF77–Signals erledigt werden

job.sec blinkt ebenfalls mit einer Status–LED, erhöht den Wert in `uptime` um 1 und aktualisiert die Anzeige der Sekunden auf dem LCDisplay

job.min aktualsiert die komplette Information, die das LCDisplay anzeigt

job.hour leer

job.day leer

job.month die Länge des Monats in `tu.limits` wird aktualisiert

job.year das Jahr im EEPROM der RTC wird aktualisiert

multitasker

Der Multitasker wird in der von `AmForth` mitgelieferten Form benutzt. Es gibt nur 2 Tasks: Task 1 bearbeitet die Befehls–Schleife an der seriellen Schnittstelle, und Task 2 betreibt die `timeup`–Uhr und ihre Jobs.

turnkey

Jetzt sollten wir noch dafür sorgen, dass die Uhr beim Einschalten gestartet wird. `applturnkey` ist die mitgelieferte Start–Funktion. Diese wird üblicherweise über die Funktion `turnkey` aufgerufen. `init` initialisiert die gewünschte Peripherie und die Variablen des Programms, registriert die ISR des clock ticks, liest die RTC Zeit, überträgt sie in die Zähler der `timeup`–Uhr und initialsiert die Anzeige. `starttasker` setzt Task 2 korrekt auf, verwandelt die Befehlsschleife in Task 1 und startet den Multitasker.

```
: run-turnkey
  applturnkey
  init
  starttasker
;
' run-turnkey to turnkey
```

Kalibrierung

Die bis hierher realisierte Uhr läuft *frei*. Ihre angezeigte Zeit hängt nur von der Frequenz des Uhrenquarzes ab. Die Zeit der RTC kann auf der seriellen Schnittstelle gestellt und anschließend auf die timeup–Uhr übertragen werden:

```
~61> decimal
 ok
~61> 2016 8 2 19 35 30 0 set.rtc hwclock>timeup
 ok
```

Weicht die Anzeige der Uhr nicht spürbar vom tatsächlichen Verlauf der Zeit ab, dann sind wir jetzt fertig. Ist das nicht der Fall, dann müssen wir die Abweichung der Uhr messen und die Konstante `f_control` berechnen, um die Abweichung zu reduzieren. Angenommen, die Uhr geht zu schnell, und zwar um $\Delta t = 79$ Sekunden in $\Delta T = 24$ Stunden. Dann läuft die Uhr um den Faktor

$$(\Delta T + \Delta t)/\Delta T = 86479/86400 \approx 1.00091435$$

zu schnell. Dann muss man den aktuellen Wert von `f_control` ($2000000 = 33554432) durch den gerade genannten Faktor teilen:

$$33554432 \cdot \frac{86400}{86479} = 33523779 = \$1FF8843$$

Die Korrektur beträgt 30653. Der neue Wert ist kleiner, d.h., es braucht manchmal 129 statt 128 ticks, bis der Überlauf zur Sekunde eintritt. Die Uhr wird langsamer. Der neue Wert kann direkt in `EE.fcontrol` bzw. `ct.fcontrol` eingetragen werden.

```
~61> $1FF8843. to EE.fcontrol
 ok
~61> $1FF8843. ct.fcontrol 2!
 ok
```

Bei meinem Prototypen ist die Uhr deutlich zu langsam — was auch der Grund ist, warum ich mich auf dieses Spiel überhaupt eingelassen habe. Beim Versuch, die Abweichung zu messen, stellte sich heraus, dass der Uhrenquarz *unregelmäßig* läuft, die Abweichung schwankte deutlich. Nach 5 Tagen habe ich den Quarz durch einen neuen ersetzt und die Kontakte neu verlötet. Zusätzlich habe ich das gleiche Experiment auf einer anderen Platine installiert. Man weiß bekanntlich nie, wo es klemmt.

	ΔT	Δt	$f_{control}$
2016-08-25 19:00			$2000000
2016-08-27 19:00	90000	-39	$20038D3
2016-08-28 11:00	147600	-64	$20038DC

Schluss

Der erreichte Zustand ist als Uhr benutzbar. Wenn man die Drift gemessen und ausgeglichen hat, und wenn man die Uhrzeit auf Universal Time stellt, dann hat man das anfangs gesteckte Ziel erreicht, mit Ausnahme der LED Anzeige. Auch bei diesem Projekt zeigt sich: Artikel schreiben ist ein guter Audit für das Programm. So manche Verbesserung wäre sonst nicht zustande gekommen.

Die RTC hat übrigens ihren eigenen Quarz. Der ist deutlich näher an der korrekten Frequenz. Die RTC geht ein wenig zu schnell, die Abweichung ist aber gering. Man könnte sie sogar ignorieren.

Langjährige Leser der VD kennen diesen Themenbereich von mir, denn in meinen allerersten Artikeln für die VD (2006-03,-04) habe ich darüber schon geschrieben — damals noch mit gforth–ec auf dem Renesas r8c Kontroller. Wie die Zeit vergeht! Damit habe ich jetzt quasi 10–jähriges und seither knapp 20 Artikel beigesteuert. Schnaps!

Die besondere Uhr Das Internet ist voller schöner, eleganter, bizarrer Beschreibungen für selbstgebaute Uhren. Gislain Benoit hat eine aus diskreten Bauteilen aufgebaute Uhr geschaffen — ein Meisterstück. Unbedingt ansehen und staunen! [10]

Verweise

1. Unix–Zeit, Epochensekunden `https://de.wikipedia.org/wiki/Unixzeit`
2. Universal Time (Coordinated) `https://de.wikipedia.org/wiki/Universal_Time`
3. Swatch Beats `https://de.wikipedia.org/wiki/Swatch-Internetzeit`
4. Numerically Controlled Oscillator `https://en.wikipedia.org/wiki/Numerically_controlled_oscillator`
5. VD-2006-04, S.9ff — E. Wälde, Adventures 2: Eine Forth Uhr
6. VD-2007-01, S.10ff — E. Wälde, Adventures 4: Eine Funkuhr
7. Datenblatt LCDisplay `https://www.pollin.de/shop/downloads/D120622D.PDF`
8. VD-2006-03, S.14ff — E. Wälde, Adventures 1: i2c Thermometer
9. VD 2008-03, S.35ff — M. Kalus, Mit der Zeit gehen
10. Gislain Benoit — The Clock `http://techno-logic-art.com/clock.htm`

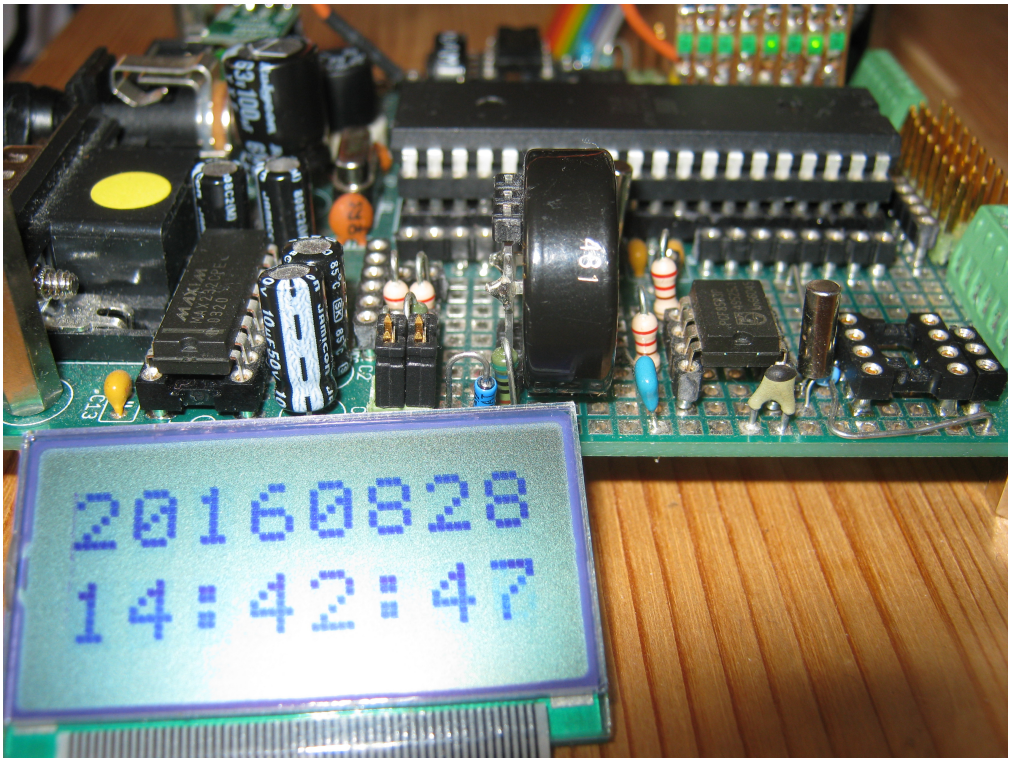


Abbildung 1: Clock Works 1: Prototyp

Listings

Das Programm ist viel zu lang, um es hier komplett abzudrucken. So beschränke ich mich hier auf das Hauptprogramm. Das komplette Programm kann auf der Webseite der Vierten Dimension heruntergeladen werden. Vorsicht, manche `include`–Zeilen wurden für den zweispaltigen Satz umgebrochen.

```
1   \ 2016-08-02 ew
2
3   include lib/builds.frt
4   include lib/forth2012/core/erase.frt
5   include lib/dot-base.frt
6   include lib-avr8/imove.frt
7   include lib-avr8/bitnames.frt
8   include lib-avr8/forth2012/core-ext/
9            marker.frt
10  include lib-avr8/forth2012/core/
11           environment-q.frt
12  include lib-avr8/dot-res.frt
13  include lib-avr8/forth2012/core/
14           avr-values.frt
15  include lib-avr8/forth2012/core-ext/
16          avr-defers.frt
17  include lib/forth2012/core/is.frt
18  include lib/forth2012/tools/dumper.frt
19  include atmega644p.fs
20  include lib/forth2012/double/2variable.frt
21  include lib/forth2012/double/2constant.frt
22  include lib/forth2012/double/2-fetch.frt
23  include lib/forth2012/double/2-store.frt
24  include lib/forth2012/double/m-star-slash.frt
```

```
include lib/quotations.frt
include lib-avr8/eallot.frt
include lib-avr8/2evalue.frt


marker --start--

$0061 Evalue stationID
$0011 Evalue swVersion

\ --- ports, pins, masks -------------------

PORTA 1 portpin: _hd44780-rs
PORTA 2 portpin: _hd44780-rw
PORTA 3 portpin: _hd44780-en
$f0    constant hd44780-data-mask
PORTA  constant hd44780-port
DDRA   constant hd44780-ddr

PORTB 0 portpin: lederr
PORTB 1 portpin: ledsec
PORTB 2 portpin: ledsensor
PORTB 3 portpin: leddata

\ --- famous includes and other words ------
: ms        ( n -- )
  0 ?do pause 1ms loop ;
: u0.r      ( u n -- )
  >r 0 <# r> 0 ?do # loop #> type ;

\ --- save original emit
' emit defer@ constant emit.orig
: /lcd  emit.orig to emit ;

\ --- handle bit flags
include ewlib/flags.fs

\ --- ready prompt shows stationID
: .stationID_ready
  cr
  [char] ~ emit
  base @  $10 base !
  stationID 2 u0.r
  base !
  [char] > emit space
;

\ --- driver: status leds
include ewlib/leds.fs

\ --- driver: 2x8 pollin LCD (4bit mode)
include ewlib/hd44780-4bit.fs
include ewlib/lcd_2x8_pollin.fs

: lcd-msg
  hex lcd.page >lcd
  0 0 lcd.pos   ." stn " stationID 4 u0.r
  1 0 lcd.pos   ." ver " swVersion 4 u0.r
  /lcd
;

\ --- driver: i2c rtc clock
include ewlib/twi.fs
include ewlib/i2c.fs
\ requires counters year .. second
include ewlib/i2c_rtc.fs


\ --- timeup: time/date system with jobs
include ewlib/timeup_2.2.fs

: .date
  year  @    4 u0.r
  month @ 1+ 2 u0.r
  day   @ 1+ 2 u0.r
;
: .time
  hour @ 2 u0.r [char] : emit
  min  @ 2 u0.r [char] : emit
  sec  @ 2 u0.r
;

: hwclock>timeup ( -- )
  rtc.get    \ --
             year  !
  bcd>dec 1- month !
  bcd>dec 1- day   !
  bcd>dec    hour  !
  bcd>dec    min   !
  bcd>dec    sec   !
  drop \ 1/100 secs
  year @   month @ 1+  tu.upd.limits
;
: timeup>hwclock ( -- )

  year @   month @ 1+  day @ 1+
  hour @   min   @     sec @
  tick @ #100 #128 m*/
  ( Y m d H M S S/100 ) rtc.set
;

\ --- clock display (deferred)
include ewlib/clock_display_defer.fs
include ewlib/clock_display_pollin_2x8.fs

\ --- timer2 clock tick
2variable uptime
: .uptime  ( -- )
  uptime 2@ decimal ud. [char] s emit ;
: ++uptime ( -- )
  1.  uptime 2@  d+  uptime 2! ;
include ewlib/clock_tick_v2.5.fs

\ --- multitasker
include lib/multitask.frt
: +tasks  multi ;
: -tasks  single ;

\ --- timeup jobs --------------------------
: job.tick
  leddata toggle
;
: job.sec
  ledsec toggle
  ++uptime
  cd.time.S
;
: job.min
  cd.date
  cd.time
;
: job.hour  ;
: job.day   ;
```

```
: job.month
  \ update length of month in tu.limits
  year @  month @ 1+  tu.upd.limits
  ;
: job.year
  \ update YYYY in eeprom of rtc
  year @  rtc.set.year
;

create Jobs
  ' job.tick ,
  ' job.sec , ' job.min ,   ' job.hour ,
  ' job.day , ' job.month , ' job.year ,

variable jobCount
: jobCount++
  jobCount @
  6 < if
    1 jobCount +!
  then
;

\ --- task 2 -------------------------------
: run-job
  [’] tx-poll to emit    \ add emit to inner
                         \ workings of run-job
  begin
    tickover? if
      1 tick +!
      job.tick
      tick.done
    then

    secover? if
      0 tick !
      timeup
      1 bv tu.flags fset
      1 jobCount !
      sec.done
    then

    \ these are run "one job per loop",
    \ not all in one go.
    jobCount @
    bv tu.flags fset?
    if
      jobCount @ dup
      Jobs + @i execute
      bv tu.flags fclr
    then
    jobCount++

    pause
  again
;
$40 $40 0 task: task_job \ create task space
: start-job
  task_job tib>tcb
  activate           \ words after this line
                     \   are run in new task
  run-job
;
: starttasker
  task_job task-init     \ create TCB in RAM
  start-job             \ activate tasks job

  onlytask            \ make cmd loop task-1
  task_job tib>tcb alsotask   \ start task-2
  multi              \ activate multitasking
;


\ --- main ---------------------------------
: init
  hd44780-4bit-init lcd.page
  +leds leds-intro
  lcd-msg #2000 ms
  [’] .stationID_ready is .ready
  +twi

  0. uptime 2!
  +ticks
  timeup.init
  i2c_addr_rtc twi.ping? if
    hwclock>timeup
  else
    #2016 1 1 0 0 0 tu.set
  then

  -clock.display
  +clock.display.pollin_2x8
  lcd.page
  cd.date
  cd.time
;

: run
  init
  starttasker
;

: run-turnkey
  applturnkey
  init
  starttasker
;
\ ’ run-turnkey to turnkey

: .d ( -- )
  decimal
  .uptime space space
  tu.show space
  tick @ . space
  cr
;
```

How to clone noForth on a MSP430G2553

Willem Ouwerkerk

The project was born out of a need to copy noForth[1] onto the MSP430. And that without using any special software on your personal computer! Each platform has its own programming tools. Linux has MSPDEDUG, Windows MSPFET, MSP430Flasher or Lite FET–Pro430, etc — and the 4E4th–IDE for the MSP430G2553. Different MSP430 series do often also need other software or at least a different USB driver . . . and that does not always go well!

Boot Strap Loader (BSL) in the MSP430 Familie MCUs

The older MSP430F149 boards are only flashable via the built–in BSL. The newer Launchpad boards need the correct USB drivers installed to be programmed. In general, the flash of the most MSP430 can be programmed in several ways:

1. BSL (RS232/I2C)
2. JTAG
3. Spy–Bi–Wire (serial version of JTAG)

Further search revealed that any MSP430 comes with BSL software inside. In essence that is nothing more than a simple monitor program with some flash programming functionality.

Unfortunately there are traps. The built in BSL comes in many variations! Evenmore those in the MSP430F149 have a broken flash programming function, you have to program a patch in RAM to get it working.

Other BSL variants have additional commands and / or a different method for the checksum. As a first target, the much–used MSP430G2553 is chosen, which is also part of the „EGEL“ project for the MSP430. The cloner fits in there nicely. This first implementation is easy. It clones itself. The next version will provide other MSP430 targets too. For example by addition of an external serial EEPROM many binaries may be stored. On every OS, the cloner only requires the same simple two tools: a MSP430G2553 with noForth and a terminal program.

Documentation

To build the cloning software these documents are important:

• SLAU319J.PDF — BSL description

• msp430g2xx3_dco_flashcal.s43 — Re–calibrate the DCO

Implementation

It had a number of difficulties to be overcome:

1. Every MSP430 is using certain pins as BSL, for MSP430G2553 these are: TEST, RESET, RX = P1.5[2], TX = P1.1, VCC and GND.
2. Communication via RX and TX is fixed at 9600 baud with 8 data bits, even parity and one stop bit. For this, for lack of a free UART, a bit–bang RS232 software is used. To receive data, we ignore the parity for a simpler implementation!
3. The data is sent via data frames with a variable length, and a checksum at the end. For each command first a sync must be sent ($80) and the BSL responds with an ACK ($90) or NACK ($A0), and after a command an ACK or NACK follows too.

Figure 1: A Frame

4. To enter the BSL two things have to be done. First send a special reset sequence (RESET–TO–BSL), then send a password. The BSL command for this is $10 followed by the contents of the interrupt table of the target! That's a problem in (no) Forth![3]

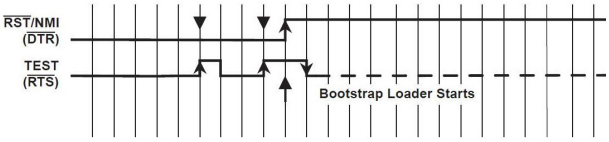


Figure 2: BSL Entry Sequence

5. If the password is wrong, the BSL in MSP430G2553 wipes out the entire flash memory. What, the whole flash? Yes, also the Info–A block[4] with the DCO- and ADC[5] constants. As you may know already, these are by TI calibrated values at the end of the production process! Without these values the DCO initializes no longer a defined frequency. But a work around is fortunately available to recalibrate the DCO with the help of a 32KHz crystal.

[1] noForth : 16 bit forth for MSP430 by Albert Nijhof & Willem Ouwerkerk

[2] BSL uses port pin P1.5 instead of UART P1.2 !

[3] A forth app may change this table ! Then the password in the clonig MCU is different from that in the target.

[4] This is obviously a mistake in the BSL, more on that later.

[5] DCO : digital controlled oscilator; ADC : analog to digital converter

6. The timing of the BSL protocol is also difficult to figure out. An ACK or NACK follows as soon as a BSL command is received or executed. But for the delete command it comes after the command has been completed. This can last a few milliseconds up to half a second (WAITACK).
7. To restore lost DCO constants a separate programming example is created. See the documentation above. For this BSL I made a slightly improved version. By connecting a 32KHz crystal on the pins P2.6 and P2.7 it generates valid DCO constants and writes them back to Info–A. This happens when noForth in that just cloned target starts running on its first regular reset, with its boot application set : `' restore-dco to app`

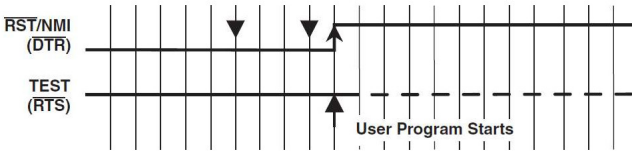


Figure 3: Regular Reset

Well, it is done now. So have fun using noForth as a cloner of your apps.

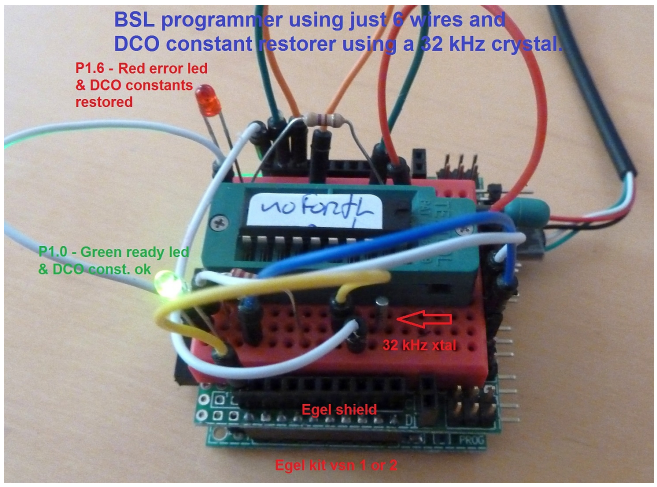


Figure 4: The Cloner build with EGEL Kit on an EGEL shield

Link

http://noforth.bitbucket.org/site/egel%20for%20launchpad.html#e111

Table 1. Data Frame of BSL Commands (1)(2)(3)(4)(5)(6)

Received BSL Command	HDR	CMD	L1	L2	AL	AH	LL	LH	D1	D2...Dn	CKL	CKH	ACK
RX data block	80	12	n	n	AL	AH	n–4	0	D1	D2 … Dn–4	CKL	CKH	ACK
RX password	80	10	24	24	xx	xx	xx	xx	D1	D2 … D20	CKL	CKH	ACK
Erase segment	80	16	04	04	AL	AH	02	A5	–	– – –	CKL	CKH	ACK
Erase main or info	80	16	04	04	AL	AH	04	A5	–	– – –	CKL	CKH	ACK
Mass erase	80	18	04	04	xx	xx	06	A5	–	– – –	CKL	CKH	ACK
Erase check	80	1C	04	04	AL	AH	LL	LH	–	– – –	CKL	CKH	ACK
Change baud rate	80	20	04	04	D1	D2	D3	xx	–	– – –	CKL	CKH	ACK
Set mem offset	80	21	04	04	xx	xx	AL	AH	–	– – –	CKL	CKH	ACK
Load PC	80	1A	04	04	AL	AH	xx	xx	–	– – –	CKL	CKH	ACK
TX data block	80	14	04	04	AL	AH	n	0	–	– – –	CKL	CKH	–
BSL responds	80	xx	n	n	D1	D2 ...	...	...	...	... Dn	CKL	CKH	–
TX BSL version	80	1E	04	04	xx	xx	xx	xx	–	– – –	CKL	CKH	–
BSL responds	80	xx	10	10	D1	D2 ...	...	...	...	... D10	CKL	CKH	–

(1) All numbers are bytes in hexadecimal notation.

What does the code do?

- 9600 baud bit–bang RS232 software with even parity — `BSL-EMIT` and `BSL-KEY`
- Resetting the target to the built in BSL — `RESET-TO-BSL`
- Normal reset so that the MSP starts running — `RESET-CLONE`
- Sync command ($ 80) to adjust the RS232 — `SYNC`
- Generate a checksum for the command block — `INIT CHECKSUM` and `SEND-CHECKSUM`
  The checksum comprises a 16–bit XOR over all the words of a command.

- Commands without data — `COMMAND` and `ERASE-FLASH`
  | 80 | cmd | L | L '| AL | AH | LL | LH | CKL | CKH |
  The response is usually an ACK ($90)
- Commands with data — `DATACOMMAND SEND-PASSWORD` and `WRITE-BLOCK`
  | 80 | cmd | L | L '| AL | AH | LL | LH | data block | CKL | CKH |
  The response is usually an ACK ($90)
- An array for the password, print and fill — `.PASS !PASS EMPTY-PASS`
  A password is made up of the interrupt table, which has a length of 16 cells.
- Used flash erase and write — `ERASE-FLASH` and `WRITE-FLASH`
  First, the main flash is erased with: `hex C000 A504 16` commandand after that the info–B flash with the noForth startup values: `1080 A502 16` command After that writing of the flash is divided into three pieces: the Info–B flash, the main flash in chunks of $80 bytes, and finally, the interrupt table as a block of $40 bytes.
- Finaly, a re–calibration of the DCO, if necessary, and the rewriting of the DCO constants — `CALIBRATE-DCO` and `RESTORE-DCO`

Listing

```
(* E111 - For noForth C&V2553 lp.0, CLONE noForth to another MSP430G2553
  & calibrate DCO to an 32KHz XT, when the DCO constants where destroyed!

  Do not forget to mount a 32kHz crystal. It is used as a reference to generate and restore a set of new DCO-constants.
  For more information on the built in capacitators read SLAU144J.PDF page 274 and beyond.

  ------------------------------------------------------------------
  LANGUAGE    : noForth BSL & calibrate DCO vsn 0001 April 2016
  PROJECT     : BSL with software RS232
  DESCRIPTION : With half duplex software RS232
  CATEGORY    : Application, size: ~1340 bytes.
  AUTHOR      : Willem Ouwerkerk, 12 April 2016
  LAST CHANGE : Willem Ouwerkerk, 07 june 2016
  ------------------------------------------------------------------

About baudrates.

Not all baudrates are applicapable using the word WAIT-BIT. The maximum delay now is 65535.
It is to the programmer to write alternative versions, 9600baud is chosen as default baudrate.
It works for every supported frequency.

  #027 CONSTANT BITRATE#           \ 9600 Baud at 1 MHz
  #059 CONSTANT BITRATE#           \ 9600 Baud at 2 MHz
  #129 CONSTANT BITRATE#           \ 9600 Baud at 4 MHz
  #268 CONSTANT BITRATE#           \ 9600 Baud at 8 MHz
  #545 CONSTANT BITRATE#           \ 9600 Baud at 16 MHz

More info about the built-in BSL in SLAU319L.PDF . Read the pages 5 to 12 , it contains the core of the BSL.

Extra comment:
 1) BSL versions of 2.01 & higher do not! protect info flash during mass erase.
 2) BSL versions of 1.60 & higher do an automatic erase check after an erase.
    The ACK flag signals an succes, a NACK a not succes! The erase check instructions are not implemented here!!!
 3) A mass erase must be executed at least 12 times to ensure
    A total erasure time of 200 ms or larger. For the MSP430G2553 this is minimal 20 ms!!
 4) The cumulative program time of a 64-byte flash block must not exceed 10 ms for the MSP430G2553.
 5) Location FFDE may be used to secure the flash programming. AA55 = Disable BSL, 0000 = Disable erasure.
 6) For the MSP430G2553 Receive = P1.5, Transmit = P1.1
 7) Timing for BSL entry improved must be at least a 100 us wait!
 8) Take care when using an incorrect password a MASS-ERASE is done!! This leaves you without correct DCO constants.
 9) It takes almost 16 seconds to perform a CLONE action
10) Connections from programmer to target
    Programmer         Target
    --------------------------------
    P2.0 = Output   -> P1.5 RX
    P2.1 = Input    -> P1.1 TX
    P2.2 = Test     -> TST
    P2.3 = Reset    -> RST
    VCC             -> VCC
    GND             -> GND
    --------------------------------
    P1.0 = red led, P1.6 = green led
 *)
```

```
54
55  hex chere  ( Remember CHERE for space calculation )
56
57  routine WAIT-BIT  ( -- a )      \ Wait bittime
58      dm 268 # day mov
59      begin,  #1 day sub  =? until,
60      rp )+ pc mov  ( ret )
61  end-code
62
63  code BSL-EMIT   ( char -- )     \ RS232s Char to RS232
64      #0 sun mov   ( Make parity even, code Albert Nijhof)
65      tos day mov
66      begin,
67          day sun bix             \ Gather lowest bit
68          day rra                 \ Shift char to right
69      =? until,                   \ Until no bits are left high
70      #1 sun bia                  \ Use bit 0 only
71      sun swpb                    \ Use parity as bit-8
72      sun tos bis                 \ Add to char
73      200 # tos bis               \ Bit-9 is stopbit
74      tos tos add                 \ Make room for startbit
75      0B # moon mov               \ Start + 8 + Parity + Stop bits
76      begin,
77          tos rrc                 \ Get next bit to carry
78          cs? if,
79              #1 29 & bis         \ Send one
80          else,
81              #1 29 & bic         \ Send zero
82          then,
83          wait-bit # call         \ Wait bittime
84          #1 moon sub             \ Send all bits
85      =? until,
86      sp )+ tos mov  next
87  end-code
88
89  \ A timeout of 4 ms is added to the startbit loop.
90  \ When a timeout occurs a NACK (A0) is left on the stack. The parity bit is not checked in this implemention!
91  code BSL-KEY    ( -- char )     \ rs232s read char from rs232
92      tos sp -) mov
93      1000 # moon mov             \ ~ 4 millisec. timeout
94      begin,
95          #1 moon sub
96          =? if, A0 # tos mov next then,  \ Leave NACK at timeout
97          #2 28 & bit             \ wait for startbit
98      cc? until,
99      30 # moon mov               \ wait extra to reach sample point
100     begin,  #1 moon sub  =? until,
101     #8 moon mov                 \ read 8 databits & parity bit
102     begin,                      \ Leave stopbit
103         wait-bit # call         \ wait bittime
104 \ )     20 # 21 & bix           \ Trace sample moment at P1.5
105         #2 28 & bit             \ read rx line to carry
106         tos rrc                 \ shift rx into char
107         #1 moon sub             \ return at stopbit
108     =? until,
109     wait-bit # call             \ wait bittime for parity
110     wait-bit # call             \ wait bittime for stopbit
111 \ ) 20 # 21 & bix               \ Trace stopbit moment at P1.5
112     tos swpb
113     #-1 tos .b bia  next        \ Use low byte only
114 end-code
115
116 \ BSL programmer for Launchpad and/or Egel kit
117
118 : ?SIGNAL       ( fl -- )       \ Red is on when fl = true, otherwise green
119     if      01 21 *bis  40 21 *bic
120     else    40 21 *bis  01 21 *bic
121     then ;
122
123 : LEDS-OFF      ( -- )   41 21 *bic ;
124
125 code SPLIT      ( x -- bl bh )  \ Split word in bl=lowbyte bh=highbyte
126     tos day mov
127     #-1 day .b bia
128     day sp -) mov
129     tos swpb
```

```
130     #-1 tos .b bia
131     next
132 end-code
133
134 : SETUP-BSL     ( -- )      \ P2.0 is Output, P2.1 is Input
135     02 2A *bic  0D 2A *bis  \ P2.2 is Test, P2.3 is Reset
136     01 29 c!  10 ms ;       \ P2.0 is high, rest low
137
138 : RESET-TO-BSL  ( -- )
139     08 29 *bic  10 ms               \ Reset low & wait
140     04 29 *bis  noop  04 29 *bic    \ Test pulse 1
141     04 29 *bis  1 ms  08 29 *bis    \ Pulse 2 & BSL reset
142     1 ms  04 29 *bic  1 ms ;        \ Wait and release test
143
144 : RESET-CLONE   ( -- )      08 29 *bic  1 ms  08 29 *bis ;  \ Give reset
145 : ACK?          ( -- fl )   bsl-key 90 = ;  \ Check for ACK=true
146 : NACK?         ( -- fl )   ack? 0=  dup ?signal ;  \ noAck=true
147
148 : SYNC          ( -- )      \ Sync RS232 for BSL
149     2 ms  80 bsl-emit  nack? ?abort  2 ms ;
150
151 : WAITACK       ( -- )      \ Wait max. 500 ms for an ACK
152     80 0 ?do
153         ack? if
154             false ?signal  unloop  exit
155         then
156     loop  true ?signal  true ?abort ;
157
158 value CHK   \ Generate checksum for commands
159 : INIT          ( -- )          0 to chk  leds-off ;
160 : CHECKSUM      ( b1 b2 -- )    >< or  chk xor  to chk ;
161 : SEND-CHECKSUM ( -- )          chk invert  split >r  bsl-emit  r> bsl-emit ;
162 : BSL-DATA      ( b1 b2 -- )    over bsl-emit  dup bsl-emit  checksum ;
163
164 \ A command contains:  Header of 8 bytes, data block & checksum
165 \  |80|cmd|L|L'| AL| AH| Ll | LH| data block | CKL | CKH |
166 \  Answer most of the time an ACK = 90
167 : COMMAND)      ( adr len2 len1 command -- adr len2 )
168     sync  init  80 swap bsl-data    \ a 12 11  Sync first, then command
169     4 + dup bsl-data                \ a 12     Send record length
170     over split bsl-data             \ a 12     Send address
171     dup split bsl-data ;            \ a 12     Send data length (max 250)
172
173 \ BSL command without data block
174 : COMMAND       ( adr len2 len1 command -- )
175     command) 2drop  send-checksum ;
176
177 \ BSL command with data block
178 : DATACOMMAND   ( adr len2 len1 command -- flag )
179     command)                        \ Send command frame
180     bounds ?do                      \          Send data block
181         i @ split bsl-data
182     2 +loop
183     send-checksum                   \ Finally checksum
184     nack? ;                         \ Show if (no)Ack received?
185
186 create PASS 20 allot               \ Hold BSL password
187 : EMPTY-PASS    ( -- )          pass 20 FF fill ; \ PW for empty MPU
188 : .PASS         ( -- )          pass 20 dump ;  \ Show used PW
189 : !PASS         ( x +n -- )     pass +  ! ;     \ Fill cell +n in PW array
190
191 : MODIFY-PASS   ( a -- )            \ Set PW for non empty noForth MPU's
192     1E 0 do   FFDE i !pass  2 +loop  01E !pass ;
193
194 \ Used bootloader commands
195 : SEND-PASSWORD ( -- )      pass  0020   dup  10 datacommand ?abort ;
196 : WRITE-BLOCK   ( a +n -- )             dup  12 datacommand ?abort ;
197
198 : ERASE-FLASH   ( -- )
199     ." E "  C000  A504  0000  16 command  waitack   \ Erase main
200             1080  A502  0000  16 command  waitack ; \ Erase INFO-B
201
202 : WRITE-FLASH   ( -- )
203     ." W "  1080 40 write-block     \ Copy INFO-B flash segment
204     chere C000 ?do  i 80 write-block  80 +loop \ Copy main Flash
205     FFC0 40 write-block ;           \ Copy vector table
```

```
206
207  \ Write used part of flash, vector table & segment-B of info flash
208  : CLONE         ( -- )
209      setup-bsl  reset-to-bsl  send-password
210      erase-flash  write-flash  100 ms reset-clone ;
211
212  (* DCO calibration for noForth C&V for MSP430G2553
213
214  The program uses the capture unit to build a PLL = Phase Locked Loop to adjust the DCO frequency
215  It does so by comparing the DCO to a 32KHz external clock, divided by 8 = 4096 Hz. 1 Mhz counts exactly to 244
216  in one 4096 Hz period. The routine GET-DCO trims the DCO and then checks the count again, until 244 is reached.
217  No more no less! Finally the values are stored in the INFO-A flash segment.
218
219  Current version is more accurate by capturing the DCO two times. Doing it this way, the DCO runs at a
220  stable frequency. By feeding the routine with real DCO values is stabilizes very quick.
221
222      0244 =  1 MHz capture value, DCO seed 86D4, address 10FE
223      1953 =  8 MHz capture value, DCO seed 8D8A, address 10FC
224      2930 = 12 MHz capture value, DCO seed 8E95, address 10FA
225      3906 = 16 MHz capture value, DCO seed 8F8F, address 10F8
226
227  By putting this routine in the APP vector the damaged MSP is automaticly corrected. The program does nothing
228  when the DCO values are not erased! DCO values are DCOCTL & BCSCTL1 in a word as stored in info-A: |BSCTL1|DCOCTL|
229   *)
230
231  code GET-DCO    ( delta-freq dco-seed -- dco-parameters )
232      tos 056 & .b mov  \ Seed lowbyte to DCOCTL
233      tos swpb
234      tos 057 & .b mov  \ Seed highbyte to BCSCTL1
235      sp )+ tos mov
236      30 # 057 & .b bis \ BCSCTL1  Setup Timer A & capture
237      5100 # 162 & mov  \ TA0CCTL0  Capture on rising edge, using ACLK
238      0224 # 160 & mov  \ TA0CTL  Source=SMCLK, cont. mode, clear TA
239  \ Capture DCO two times, for more accurate measurement
240      begin,            \ Start DCO trim loop
241        begin,  #1 162 & bit  cs? until,  \ Capture1?
242        #1 162 & bic
243        172 & moon mov  \ TA0CCR0  Get timer
244        begin,  #1 162 & bit  cs? until,  \ Capture2?
245        #1 162 & bic
246        172 & day mov   \ TA0CCR0  Get timer
247  \ Test against delta value and go adjust DCO
248        moon day sub    \ Calc. capture difference
249        tos day cmp     \ PLL check delta against capture difference
250        =? if,    \ Delta is equal?
251            #0 162 & mov  \ TA0CCTL0
252            #0 160 & mov  \ TA0CTL
253            30 # 057 & .b bic \ BCSCTL1
254            057 & tos .b mov
255            tos swpb      \ High byte = BCSCTL1
256            056 & day .b mov  \ Low byte = DCOCTL
257            day tos bis
258            next
259        then,
260        >? if,    \ Delta is greater, DCO is to slow
261            #1 056 & .b add   \ Increase DCOCTL
262          2over cs? until,    \ Overflow?
263            #1 057 & .b add   \ Increase BCSCTL1
264          2over again,
265        then,             \ Delta is smaller, DCO is to fast
266        #1 056 & .b sub   \ Decrease DCOCTL
267      2dup cc? until,     \ Underflow?
268        #1 057 & .b sub   \ Decrease BCSCTL1
269      again,
270  end-code
271
272  code {I         ( -- )  \ Activate INFO-A Flash write sequence
273      sr day mov          \ Save status register
274      #8 sr bic           \ Interrupts off
275      A540 # 012C & mov   \ Disable lockA
276      A540 # 0128 & mov
277      next
278  end-code
279
280  code I}         ( -- )  \ End INFO-A Flash write sequence
281      A500 # 0128 & mov
```

```
282       A550 # 012C & mov   \ Enable lock & lock A
283       #8 day and>         \ Restore interrupt enable
284       day sr bis
285       next
286   end-code
287
288   \ Calibrate all four DCO constants and write to flash but only when there are no DCO constants
289   : CALIBRATE-DCO ( -- )
290   ( ) 0C 053 *bis             \ Xcap=12.5pF
291   ( ) begin  1 053 bit* 0= until \ Wait for LFXT1 correct
292       dm 3906 8F8F get-dco    \ Get 16 Mhz constants
293       dm 2930 8E95 get-dco    \ 12 MHz
294       dm 0244 86D4 get-dco    \ 1 Mhz
295       dm 1953 8D8A get-dco    \ 8 MHz
296       swap  10F8 10FE do  i {i ! i}  -2 +loop ; \ Write to info-A
297
298   \ Rewrite DCO constants but only when they are erased
299   : RESTORE-DCO   ( -- )
300       10F8 @ -1 =         \ DCO constants erased?
301       dup ?signal  if     \ Yes, show
302           calibrate-dco   \ Calibrate & write them
303       then ;
304
305     ' restore-dco  to app \ RESTORE-DCO in startup vector
306   \ DB0E modify-pass      \ Set BSL noForth password c2553 Lp 160228
307   \ DB14 modify-pass      \ Set BSL noForth password C2553 LP 160202
308     empty-pass            \ Set BSL for empty MSP430G2553 password
309   freeze
310
311   chere swap - dm u.  ( Show size )        ( End )
```

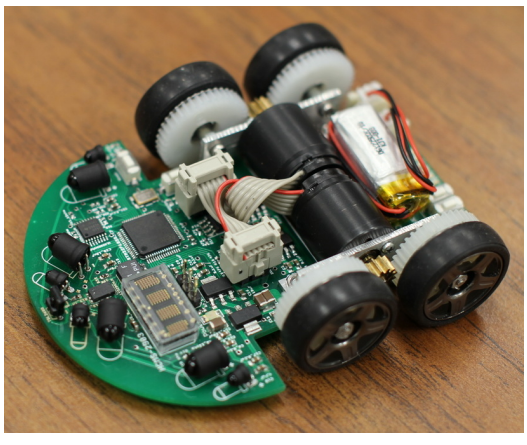
Mikro–Mäuse

Micromouse ist der Name eines Wettbewerbes, in dem sich kleine, autonome Roboterfahrzeuge in einem Labyrinth zurechtfinden müssen. Der vermutlich erste dieser Wettbewerbe wurde 1977 vom Institute of Electrical and Electronics Engineers (IEEE) in seiner Zeitschrift IEEE Spectrum ausgerufen. Inzwischen werden jährlich mehr als 100 Micromouse–Wettbewerbe, meist von Universitäten oder dem IEEE gesponsert, u. a. in den USA, im Vereinigten Königreich und in Japan veranstaltet.

Das Labyrinth hat eine quadratische Grundfläche von etwa 3 m × 3 m, darauf sind Wege in einem Raster von 16 × 16 Quadraten mit je 18 cm Seitenlänge aufgebracht; die Wände sind 5 cm hohe und 1,2 cm dicke Brettchen.

Die Micromouse–Roboterfahrzeuge sind vollständig autonom, d. h., sie bewegen sich ohne Hilfe von außen durch das Labyrinth. Es ist dem Entwickler freigestellt, wie das Fahrzeug die Wände des Labyrinthes erkennt; verbreitet sind vorn am Fahrzeug angebrachte und schräg nach außen ausgerichtete Infrarot–Abstandssensoren.

Eine aus 2 × 2 Einheiten bestehende quadratische Fläche in der Mitte des Labyrinthes stellt das Ziel dar. Die Roboter starten zunächst in einer der vier äußeren Ecken, erkunden das Labyrinth und berechnen den kürzesten Weg von der Startecke zum Ziel in der Mitte, wobei unterschiedliche Lösungsalgorithmen für Irrgärten zur Anwendung kommen. Anschließend durchfahren sie diesen Weg in möglichst kurzer Zeit. Micromouse–Fahrzeuge können Geschwindigkeiten von mehr als 3 Metern pro Sekunde erreichen, die schnellsten unter ihnen durchfahren ein Labyrinth in wenigen Sekunden.



Seit einigen Jahren gibt es die „Half–size Micromouse“–Variante, bei der die Maße der Wege und der Rasterquadrate bei gleichbleibender Labyrinthfläche auf die Hälfte reduziert wurden. Ein solches Labyrinth ist also in einem Raster von 32 × 32 Quadraten mit je 9 cm Seitenlänge realisiert. (Quelle: `Wikipedia.de`)

A new serial tool: vive la Folie!

Jean–Claude Wippler

Working with embdded µC boards involves quite a few steps: apart from the hardware itself, you need to connect to it and figure out how to upload code, of course. But you also think about the development cycle and revision control. The list of tools needed to get going can grow quickly ...

Not so with Forth. It's all self–contained and self–hosted. The embedded µC does it all, it just needs to be sent the source code. The same applies to uploading the initial firmware image: all you need, is a way to send the proper bytes to the µC while it is in a special boot loader mode.

In the Arduino world, this is handled by an IDE, which combines an editing environment, a cross–compiler for the µC you're using, the *avrdude* or *stlink* utilities to handle firmware uploads, and a built–in serial terminal for actually interacting with the code, once it's running. The success of the Arduino is probably largely due to the fact that this all–in–one approach has been properly packaged as a single "app" which runs on Windows, Mac OSX, and Linux.

Could something similar be done in a Forth–based environment?

Yes, it sure can, but quite differently: meet the **Fo**rth **Li**ve **E**xplorer — Folie, in short!

Folie combines a number of functions into an *installation–free* (!), *single–file* (!) executable, and is available for Windows, Mac OSX, and Linux (both Intel and ARM):

- it's a serial terminal to let you talk to the attached µC board
- there's command history to re–use easily previous entries via up–arrow
- there's a file include mechanism to send sources as if you had typed them in
- all lines sent are throttled to avoid over–running the Forth word parser

Here is a brief interactive session as example:

```
$ folie -p /dev/cu.usbserial-A8009L2N
Connected to: /dev/cu.usbserial-A8009L2N
ok.
7 8 * . 56 ok.
\ >>> include a
1 2 + . 3 ok.
3 a 0 1 2 ok.
\ <<<<<<<<<<< a (3 lines)
\ done.
8 9 * . 72 ok.
^D
$
```

To clarify this further, here is what was typed in, literally:

```
<CR>
7 8 * .<CR>
include a<CR>
8 9 * .<CR>
<CTRL-D>
```

The contents of the "a" file is:

```
1 2 + .
: a 0 do i . loop ;
3 a
```

As you can see, it's more or less just a line–by–line serial terminal, whereby each line is sent when you hit return (it can be edited locally until then). That and (nestable) include file expansion.

The line starting with `include` was not sent — the file's contents was sent instead. Lines which generate no output will not echo back, only lines which cause some other effect will be shown. Which is why the `: a ... ;` text does not show on the screen, but everything else does.

Folie is clearly not an IDE — it's not even trying to be one. It only handles the communication with an attached µC running Mecrisp Forth. Folie is in fact intended to be used alongside your own preferred editor. Once done editing, switch to Folie and enter `include somefile.fs` to apply the changes (or hit up–arrow plus return). With proper definitions at the beginning, this'll replace or extend what was already loaded — you can even include commands to start things up.

That's the whole Forth development cycle: edit, send changes, explore interactively, repeat ...

One more thing: to get started, Mecrisp Forth needs to be uploaded and *flashed* onto the µC. Since Folie has support for the STM32F103 ROM boot loader, it can also be used for this step:

- set up the µC board to have the BOOT0 pin tied high, with the jumper in position "1"
- insert the USB interface and board to power it up
- launch Folie as follows:
  `folie -p <comport> -u <firmware-file>`

If all is well, you should see something like figure 1.

```
$ folie -p /dev/cu.usbserial-A8009L2N -u mf224.hex
Connected to:  /dev/cu.usbserial-A8009L2N
File:  mf224.hex
Count:  15684 bytes (converted from Intel HEX)
Checksum:  4e555979 hex
Synchronise:  .+ OK
Boot loader:  22 hex
Chip type:  0410 hex - STM32F1, performance,
medium-density
Unprotect:  OK
Resume:  .+ OK
Mass erase:  OK
Uploading:  ++++++++++++++++++++++++++++++++ ...
+++++ OK
$
```

Figure 1: screen log of folie response

Now restore the BOOT0 jumper to its normal "0" setting, and press reset.

Note: as of this writing (end March 2016), uploading works on Mac OSX and Linux, but not on Windows (grrr!) — you will need to use another mechanism, see this article[1] for some options.

The latest Folie binary releases can be found here[2]. If you have Go installed and properly set up, you can also build from source (on GitHub) using this magic incantation:

```
go install github.com/jeelabs/embello/tools/folie
```

Folie is a fairly small open source application, written in Go — it is still evolving quite rapidly at the moment, but everything should work as expected w.r.t. what has been described so far. A lot of its magic comes from the excellent *chzyer/readline*[3] and *tarm/serial*[4] packages it is based on.

The one remaining issue is: what firmware do we upload to the STM32F103?

If you're really impatient, you can extract the latest `mecrisp-stellaris-stm32f103.hex` image from the Mecrisp release[5] and get your feet wet in the world of Forth on ARM chips — or . . . check out the next article for a more complete image with RFM69 wireless radio[6] support.

Links

http://jeelabs.org/article/1613b/

There is a list of latest weblog posts (more about FPGA right now than Forth), last book articles, and of course all posts and articles, along with hardware categories, at:

http://jeelabs.org/archive/

Figure 2: Screenshot of Folie, Windows 7 and eForth in LaunchPad

[1] http://jeelabs.org/book/1546c/
[2] https://github.com/jeelabs/embello/releases
[3] https://github.com/chzyer/readline
[4] https://github.com/tarm/serial
[5] https://sourceforge.net/projects/mecrisp/files/
[6] http://jeelabs.org/2015/index.html — RFM69s, OOK, and antennas

awords — ein Tool

Filippo Sala

WORDS mit alpha–numerischer Sortierung.

Dieses Tool ist nicht unbedingt notwendig aber manchmal vielleicht nützlich. Für die Anzeige der sortierten Worte von WORDS werden 2 Versionen vorgestellt, und zwar für Gforth und für Win32for. Die Version für Gforth kann man leicht auch für andere Forth–Systeme anpassen. Die Version für Win32for ist speziell.

Für die Sortierung der Worte wäre die neue Ordnung der Zeichenketten zu zeitraubend. Hierfür eignet sich sehr gut meine Sortiermethode *Indexsort*. Die Methode wurde in der VD–2003/1[1] vorgestellt. Der Name zeigt, dass die Indizes und nicht direkt die Daten sortiert werden. Das Datenfeld bleibt unverändert. Die maximale Anzahl der Worte wird auf 3000 begrenzt. Win32for hat schon jetzt ca. 2500 Worte. Die Länge `len` ist auf 16 gestellt, d.h., die kürzeren Forth–Worte werden mit Leerzeichen verlängert und die längeren abgeschnitten. Damit hat man eine optimale Bildschirmanzeige.

Listing

```
\ ========================
\ AWORDS für Gforth
\ Filippo Sala - Muenchen
\ ========================

DECIMAL

0    VALUE awflag
0    VALUE wdelay       \ Anzeige bremsen
2000 VALUE wmax         \ maximale Anzahl der Worte
16   VALUE wstrlen      \ Laenge der Worte im Speicher
0    VALUE nwords       \ Zaehler
0    VALUE wordsarray   \ Zeichenketten-Feld
0    VALUE indicesarray \ Indices-Feld

: words[]  ( i -- adr[i] ) wstrlen * wordsarray + ;
: windex[] ( i -- adr[i] ) cells indicesarray + ;

: wordsmove ( adr u -- )
   wstrlen min nwords words[] swap move
   nwords wmax > abort" too many words, increase wmax"
;

: indicesarray_init ( -- )
   nwords 0 DO i i windex[] ! LOOP
;

\ ------------------------------------------------------
: fwords
  \ Speicher reservieren
  wmax wstrlen *
  dup >r ALLOCATE throw dup to wordsarray r> bl fill
  wmax cells
  dup >r ALLOCATE throw dup to indicesarray r> bl fill
  \
  \ ------------------------------------------------
  \ Worte zum Words-Array ( 4 + für Gforth )
  \ ------------------------------------------------
  0 to nwords
  context @ 4 +
  BEGIN
    @ dup
  WHILE
    dup name>string wordsmove
    nwords 1+ to nwords
    nwords wmax > abort" too many words, increase wmax"
  REPEAT
  drop
  \
  Indicesarray_init
  \
  \ ------------------------------------------------
  \ Worte sortieren
  \ ------------------------------------------------
  awflag
  IF
    indicesarray  \ Indizes-Feld
    wordsarray    \ Words-Feld
    nwords        \ Anzahl der Elemente
    wstrlen       \ Element-Laenge
    0             \ Datentyp Zeichenkette
    isort         \ Aufruf
  THEN
  \
  \ ------------------------------------------------
  \ Worte ausgeben
  \ Die Indizes für words[] liefert windex[]
  \ ------------------------------------------------
  cr
  nwords 0
  ?DO
    wdelay ms
    i windex[] @ words[] wstrlen type 2 spaces
    i 1+ 5 mod 0= IF cr THEN
  LOOP
  \
  base @ decimal nwords cr . ." Words" base !
  0 to awflag
  wordsarray free throw
  indicesarray free throw
;

: awords ( -- ) -1 to awflag fwords ;

cr .( ----------------------------------------------)
cr .( Anwendung
cr
cr .( wdelay  VALUE Anzeige bremsen, wenn noetig)
cr .( fwords  WORDS formatiert)
cr .( awords  WORDS mit alphanumerischer Sortierung)
cr .( ----------------------------------------------)
```

[1] Das Forth Magazin „Vierte Dimension“, Heft 1/2003, kannst du aus unserem Archiv runterladen: `www.forth-ev.de` ⟶ Menü/Download/PDF–Archiv

Stacks für Forth

Matthias Trute

*Stacks **für** Forth klingt wie Eulen nach Athen tragen. Wer hat schon mal einen Stack in Forth benutzt? Also seinen eigenen, nicht einen von denen, die Forth mitbringt. Dabei fällt auf, Stacks sind so fundamental, dass weder Forth 2012 noch die Forth Foundation Library sie für Anwender bereitstellen.*

Motivation

Stacks sind ein Kernelement von Forth. Forth hat zwei davon, plus den für die Gleitkommazahlen. Ab und zu geistert ein Stringstack durch die Community. Der ist wahlweise ein Stack von Zahlenpaaren oder hat die Strings selbst auch inklusive. Esoterische Randbereiche, wie Objekte, erscheinen gelegentlich, verschwinden jedoch genauso schnell, wie sie kommen. Viele dieser Stacks geben sich große Mühe, den Datenstack sauber zu benutzen, oder sie verwirren bei dem Versuch, dies zu schaffen. Prominentes Beispiel ist der Gleitkommazahlenstack, der es letztlich nicht geschafft hat, den Datenstack sauber zu halten und abgewandert ist.

Was wäre, wenn es die Möglichkeit gäbe, eigene Stacks sauber zu definieren und zu benutzen? Im Compiler gibt es so manches, was man damit machen könnte. Der Control–Flow–Stack hat es immerhin bis in den Standard geschafft, leider mit der Bemerkung, dass er auf dem normalen Datenstack liegen kann. Camelforth und seine Freunde, zu denen 4€th und amforth gehören, haben einen eigenen Stack für `LEAVE`–Adressen. Dort werden die Adressen gespeichert, die `LEAVE` in den `DO..LOOP` Schleifen benutzt, um die Laufzeit zu verbessern und RAM zu sparen.

Einfacher Start

Stacks sind Datenstrukturen die meist, aber nicht nur, im RAM und anderen wiederbeschreibbaren Speichern liegen. Der einfachste Stack ist für zellengroße Elemente gemacht. Die kann man am Stück mit `GET-STACK` und `SET-STACK` oder auch einzeln mit `>STACK` und `STACK>` zwischen Datenstack und Zielstack austauschen. Auf dem Datenstack stehen dann die vielen normalen Operationen, wie `DUP` und `DROP`, zur Verfügung.

Daneben muss man einen Stack auch definieren können, ein Name ist da jedoch nicht unbedingt vonnöten. Um es einfach zu halten, wird auch eine maximale Größe aka Tiefe mitgegeben.

```
: STACK ( size -- stack-id )
  1+ ( size ) CELLS HERE SWAP ALLOT
  0 OVER ! \ empty stack ;
: SET-STACK ( rec-n .. rec-1 n recstack-id -- )
  OVER 0< IF -4 THROW THEN \ stack underflow
  2DUP ! CELL+ SWAP CELLS BOUNDS
  ?DO I ! CELL +LOOP ;
: GET-STACK ( recstack-id -- rec-n .. rec-1 n )
  DUP @ >R R@ CELLS + R@ BEGIN
  ?DUP
  WHILE
  1- OVER @ ROT CELL - ROT
  REPEAT
  DROP R> ;
```

Einen Namen kann man auch später vergeben. Zum Beispiel als `CONSTANT` oder `VALUE` . Die Zuweisung zu einer Konstanten heißt nicht, dass der Stack selbst konstant ist. Es ist nur die Zuordnung des Namens zum Stack, die konstant wird. Ein Value im Gegenzug kann unter einem Namen verschiedene Stacks haben.

Es ist gute Tradition in Forth, dass man allen neuen Worten auch Tests beifügt.

```
4 STACK CONSTANT test
T{ -1 test ' SET-STACK CATCH -> -1 test -4 }T
T{ 0 test SET-STACK -> }T
T{ test GET-STACK -> 0 }T
T{ 42 1 test SET-STACK -> }T
T{ test GET-STACK -> 42 1 }T
T{ 42 4711 2 test SET-STACK -> }T
T{ test GET-STACK -> 42 4711 2 }T
T{ 42 4711 0815 3 test SET-STACK -> }T
T{ test GET-STACK -> 42 4711 0815 3 }T
```

Erste Kür

Mit den obigen Worten kann man zumindest schon mal spielen. Was braucht man noch? Stacks werden von oben befüllt und von dort auch gelesen. Also sind Worte nützlich, die das Top–Of–Stack–Element verändern können. Die Namen stehen in der Tradition von `>R` und `R>`.

```
: >STACK ( x stack-id -- )
  2DUP 2>R NIP GET-STACK 2R> ROT 1+ SWAP SET-STACK ;
: STACK> ( stack-id -- x )
  DUP >R GET-STACK 1- R> ROT >R SET-STACK R> ;
```

Was fällt auf? Die Implementierung ist portabel, aber grauenvoll ineffizient. Jedesmal den gesamten Stack zu transferieren, nur um ein Element zu verändern, ist einfach nur schlecht. Das lässt Raum für optimierte Fassungen, die natürlich die Tests bestehen müssen:

```
T{ 42 1 test SET-STACK -> }T
T{ 4711 test >STACK -> }T
T{ test GET-STACK -> 42 4711 2 }T
T{ test STACK> -> 4711 }T
T{ test GET-STACK -> 42 1 }T
```

Manchmal ist der Zugriff auf das unterste Element interessant.

```
: >BACK ( x stack-id -- )
  DUP >R GET-STACK 1+ R> SET-STACK ;
: BACK> ( stack-id -- x )
  DUP >R GET-STACK 1- R> SET-STACK ;
```

Auch für diese Implementierung der Worte gilt: Portabel, aber ineffizient. Die Testfälle helfen bei der Optimierung:

```
T{ 0 test SET-STACK -> }T
T{ 42 test >BACK -> }T
T{ test GET-STACK -> 42 1 }T
T{ 4711 test >BACK -> }T
T{ test GET-STACK -> 4711 42 2 }T
T{ test BACK> -> 4711 }T
T{ test GET-STACK -> 42 1 }T
```

Die Testfälle sehen trivial aus, die Implementierung auch. Trotzdem ist es möglich, nur einige Tests zu bestehen und bei anderen durchzufallen. Selbst hello–World Programme können Bugs haben, wie jeder Programmierer weiss.

Aktive Stacks

So ein Stack kann mehr als nur Daten speichern und sie auf Zuruf bereitstellen. Man kann zum Beispiel über alle Elemente eines Stacks Aktionen ausführen. Die Informatiker haben das MAP genannt, dummerweise haben sie dabei die Argumente vertauscht. Aber Forth wäre nicht Forth, wenn es nicht alles etwas anders machen würde. Also heißt die Funktion trotzdem MAP, auch wenn es den NZUPK's[1]die Zehnägel aufwölbt:

```
: MAP-STACK ( i*x XT stack-id -- j*y f )
  DUP CELL+ SWAP @ CELLS BOUNDS ?DO
    ( -- i*x XT )
    I @ SWAP DUP >R EXECUTE
    ?DUP IF R> DROP UNLOOP EXIT THEN
    R> CELL +LOOP
  DROP 0 ;
```

Die Implementierung ist schon etwas effizienter, da sie die Interna der Stackdatenstrukturen kennt und ausnutzt. Dafür ist sie nicht anwendbar auf Stacks, die zum Beispiel im EEPROM liegen, wo der `@`–Operator nicht hinkommt.

Was diese Funktion macht, zeigen die Testfälle.

```
: s1 1 0 ; : s2 2 0 ;
: s3 3 0 ; : s4 4 -1 ;
T{ ' s1 ' s2 ' s3 3 test SET-STACK -> }T
T{ ' EXECUTE test MAP-STACK -> 3 2 1 0 }T
T{ ' s1 ' s2 ' s4 3 test SET-STACK -> }T
T{ ' EXECUTE test MAP-STACK -> 4 -1 }T
```

In diesem Fall werden alle Elemente des Stacks als Execution Token (`XT`) behandelt, die via `EXECUTE` ausgeführt werden. Dabei liefern sie jeweils zwei Zahlen zurück, von denen die erste `MAP-STACK` verbraucht. Die 0 steht für *noch nicht aufhören*, die -1 heißt *Abbruch!, nicht weitermachen*. Die zweite Zahl in den s–Worten ist einfach nur für den Test, damit man prüfen kann, ob sie auch ausgeführt wurden, oder eben auch nicht.

Das Flag `f` im Ergebnis von `MAP-STACK` signalisiert, ob es einen derartigen vorzeitigen Ausstieg gab, oder ob der gesamte Stack durchlaufen wurde.

Ausblick

Man wird es ahnen, Stacks haben was mit Recognizern zu tun. Genau genommen sind sie ein Spin-Off. Viele Worte aus dem RFD sind Aliase der hier vorgestellten Befehle.

```
: REC-STACK ( size -- stack-id )
  STACK ;
: SET-RECOGNIZERS
  ( rec-n .. rec-1 n recstack-id -- )
  SET-STACK ;
: GET-RECOGNIZERS
  ( recstack-id -- rec-n .. rec-1 n )
  GET-STACK ;
```

`SET-ORDER` und `GET-ORDER` sind genauso, sie haben nur eine andere stack-id.

Wenn man einmal anfängt, Stacks als neues Datenelement in Forth zu denken, wird es schnell spannend. Warum sollen nur Elemente von der Größe einer Zelle in Stacks verwaltet werden? `2STACK` oder `FSTACK` sind da naheliegend. Oder die bei Microcontrollern so beliebten getrennten Speicherbereiche, die so manche Eigenheiten haben, die man sinnvoll nutzen will.

Die vielen Worte des Datenstacks wie `DUP` werden sicher auch auf den eigenen Stacks Anwender finden. Die kann man portabel umsetzen und wirft dann den Optimierer an. Matthias Koch hat eindrucksvoll gezeigt, was mit Register Allokation und Konstantenfaltern alles gehen kann. Wer das letzte Quäntchen herausholen will, wird natürlich von Hand polieren wollen und könnte vielleicht noch etwas mehr schaffen.

Der beschriebene Code ist im theforth.net–Portal von Gerald Wodni verfügbar. Das Tag heißt *stack*. Getestet ist das alles mit *gforth* auf dem PC.

Link

`theforth.net` — Package manager and repository for Forth.

`http://amforth.sourceforge.net/` — AmForth is an easily extendible command interpreter for the Atmel AVR8 Atmega micro controller family and some variants of the TI MSP430.

`http://amforth.sourceforge.net/Recognizers.html` — Recognizer – Dynamically Extend The Forth Interpreter (Version 2), Matthias Trute, 30.05.2014 – PDF, 6 Seiten.

[1] Nietenzähler und Pufferküsser, ein Begriff aus der Modellbahnwelt, der Leute beschreibt, die bei Modellbahnloks die Nieten des Modells mit den Nieten des Vorbilds abgleichen; was sie mit den Puffern machen, ist offensichtlich

Forth–Gruppen regional

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

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

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

Ruhrgebiet **Carsten Strotmann**
`ruhrpott-forth@strotmann.de`
Treffen alle 1–2 Monate Freitags im Unperfekthaus Essen
`http://unperfekthaus.de`.
Termine unter: `http://forth-ev.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 41 03) – 80 48 41** `uho@forth-ev.de`
Forth–Vertrieb volksFORTH ultraFORTH RTX / FG / Super8 KK–FORTH	Ingenieurbüro **Klaus Kohl–Schöpe** Tel.: **(0 82 66)–36 09 862**$_p$

Termine

Donnerstags ab 20:00 Uhr
Forth–Chat net2o `n2o chat forth@bernd` mit dem Key `n2o keysearch kQusJ`,
voller Key: `kQusJzA;7*?t=uy@X}1GWr!+0qqp_Cn176t4(dQ*`

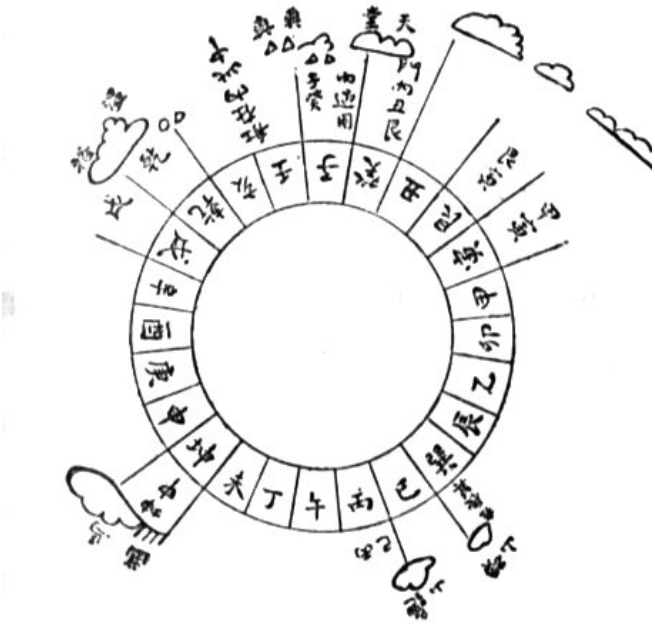
27.–30. Dezember 2016
33c3 Caos Communication Congress in Hamburg
`https://events.ccc.de/tag/33c3`

25.–26. März 2017
Maker Faire Ruhr, in der DASA in Dortmund
`http://www.makerfaire-ruhr.com`

21.–23. April 2017
Forth–Tagung in Kalkar
`https://tagung.forth-ev.de`

25.–27. August 2017
Maker Faire Hannover
`http://www.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 2017 vom 21. bis 23. April in Kalkar

Die Tagung der Deutschen Forthgesellschaft e.V. 2017 findet vom 21. April bis zum 23. April 2017 im Wunderland Kalkar statt. Das Wunderland Kalkar wurde in den Bauruinen des niemals fertiggestellten Kernkraftwerkes Kalkar am Niederrhein errichtet und ist heute ein Freizeitpark mit einigen einmaligen Attraktionen wie dem Kühlturm des geplanten Kraftwerkes. Begleiter und Tagungsteilnehmer können die Angebote des Wunderlandes frei nutzen. Langeweile dürfte daher nicht aufkommen. Im unterirdischen Teil befindet sich eine Partymeile mit vielfältiger Gastronomie.

Kernkraftwerke baut man gemeinhin nicht in Innenstädten von Ballungsgebieten. Das Wunderland liegt daher jwd (janz weit draußen). Die Anreise mit dem PKW wird empfohlen, von Flughafen Düsseldorf oder Weeze aus kann ein Shuttelservice genutzt werden. Wir empfehlen die Bildung von Fahrgemeinschaften. Die Familien Bitter bieten an, bis zu 8 Personen ab Mehrhoog (Anbindung an das Netz der DB) mitzunehmen.

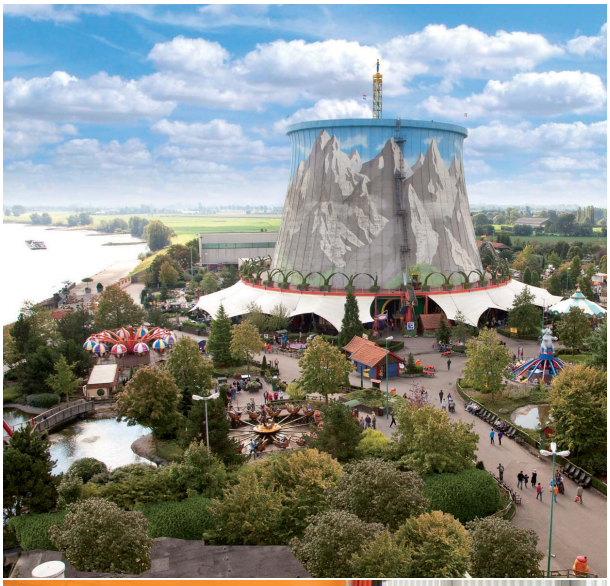
Ein Onlineanmeldeformular gibt es demnächst auf `https://tagung.forth-ev.de`. Dort wird auch Raum für Mitfahrangebote geboten.

Vorläufige Preise

Tag 0:	110€ (AE Übernachtung FS)
Frühbucher:	−20€ Rabatt
Teilnehmer:	330€ (310€)
Begleitung:	260€ (240€)
Kinder 3–12 J:	120€ Kein Frühbucherrabatt
Kinder 0–3 J:	0€

Programm

Do.	ab 13:00	Frühankommer(–Workshops)
Fr.	vormittags	Frühankommer(–Workshops)
	nachmittags	Begin der Tagung, Vorträge und Workshops
Sa.	vormittags	Vorträge und Workshops
	nachmittags	Exkursion
So.	09:00	Mitgliederversammlung
	13:00	Ende der Tagung



Bildquelle: Broschüren des Wunderland Kalkars