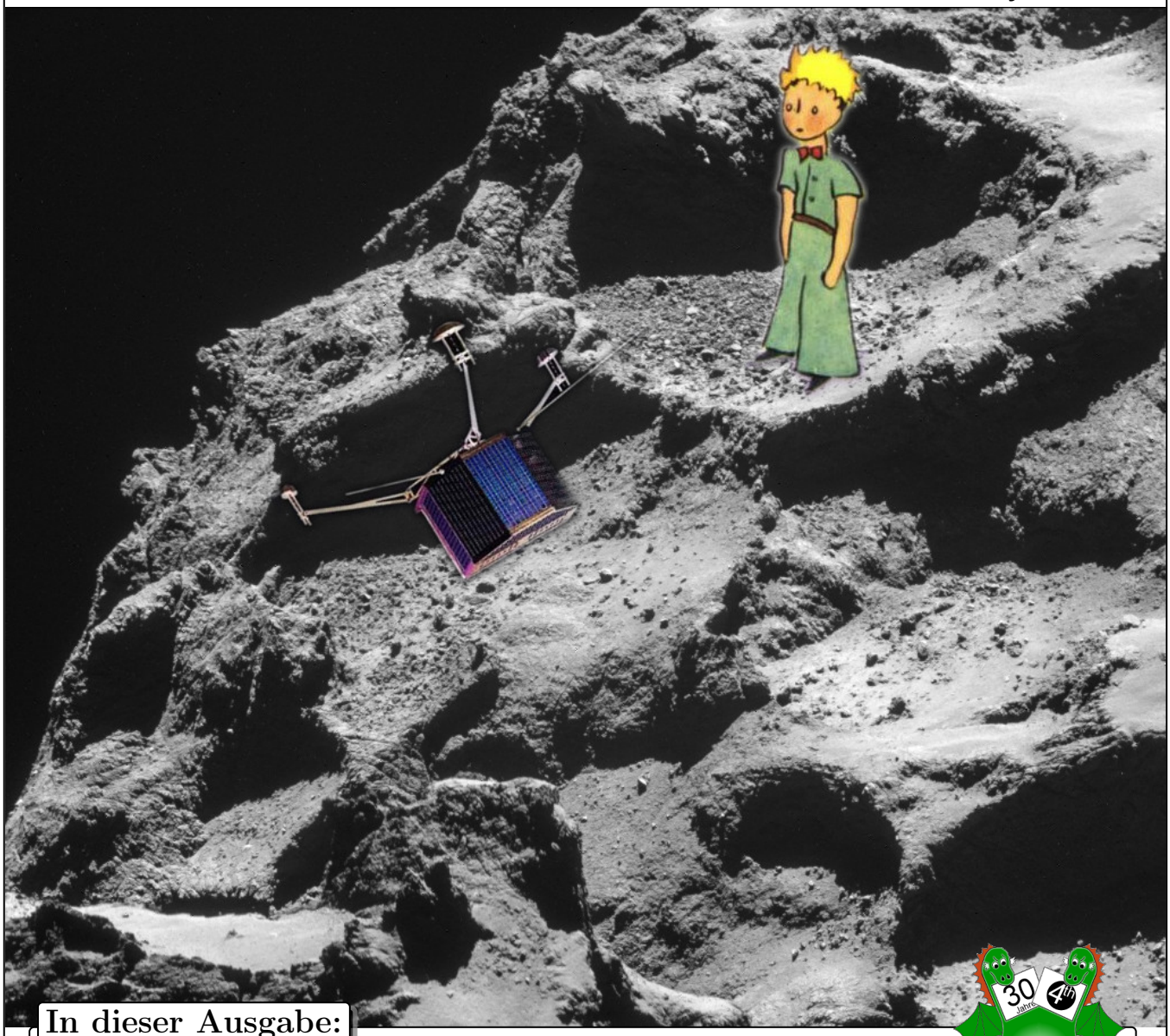
Vierte Dimension

3–4/2014

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

Impressum

Name der Zeitschrift

Vierte Dimension

Herausgeberin

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

Redaktion & Layout

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

Anzeigenverwaltung

Büro der Herausgeberin

Redaktionsschluss

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

Erscheinungsweise

1 Ausgabe / Quartal

Einzelpreis

4,00€ + Porto u. Verpackung

Liebe Leser,

Als der kleine Prinz über einen kartoffelförmigen, aber kohlrabenschwarzen Kometen lief, fand er eine blaue Blechkiste, die kopfüber in einer Spalte steckte. Vorher war sie über den Kometen gehüpft, und mehrmals im Staub gelandet — ganz langsam. „Wo kommst du denn her, blaue Blechkiste?“ — „Von der Erde“ sagte die Blechkiste. „Und ich heiße Philae. Ich bin hier auf Tschurjumow–Gerassimenko heruntergefallen und stecke nun fest“ Der kleine Prinz wunderte sich über den komplizierten Namen. „Gebt ihr auf der Erde allen Kometen so komplizierte Namen?“ — „Ja,“ sagte die Kiste, „Wir benennen sie nach den Entdeckern. Das sind oft Russen, denn die sehen ständig Sterne. Entweder nach dem Wodka–Trinken oder wenn sie in den Himmel gucken. Wenn uns das zu kompliziert ist, nennen wir den Kometen auch einfach 67P. Aber ich kann nicht viel mit dir quatschen, meine Batterie ist bald leer, und hier unten bekomme ich kein Licht, um sie wieder aufzuladen.“

Die Kiste senkte ihren Bohrer ins Gestein, um noch schnell eine Probe zu nehmen, und schaltete sich dann ab. Das war kein besonders gesprächiger Freund für den kleinen Prinz.

Bei solchen interplanetaren Expeditionen geht ja viel schief, und es gibt auch viel Unbekanntes zu entdecken. Mit dabei ist oft Forth, auch in diesem Fall: 10 strahlungsfeste RTX2010 sollen in dem Lander stecken, ausgewählt, weil sie besonders wenig Strom verbrauchen. Die Mission ist in einer DSL geschrieben, darunter arbeitet ein Forth–System, der Compiler ist von MPE. Das ist ein Standardmodul für solche Missionen.

Heute werden oft nur FPGAs eingesetzt, und die tragen dann halt eine Softcore–CPU als Nutzlast mit. Den RTX2000 gibt es von MPE auch als solchen Softcore, aber wenn man schon einen Softcore hat, kann man auch einen mit C nehmen, das kennt dann jeder. Allerdings braucht man dann auch eine größere Batterie.

Der Rest des Hefts ist dann ganz irdisch, ebenso wie die anderen News aus der Forth–Ecke: Die EuroForth hat in Palma de Mallorca getagt, mit zwei kleinen nachträglichen Online–Abstimmungen, den Forth–2012–Standard verabschiedet und zwei Tage, bevor Philae auf 67P gelandet ist, veröffentlicht.

Das letzte geplante Heft für dieses Jahr wird ein ARM–Sonderheft, und kommt auch erst nächstes Jahr, deshalb ist diese reguläre Ausgabe eine Doppelausgabe.

Bernd Paysan

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-2014-03`

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

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

TclForth — Symbiose von Tcl und Forth

Tcl und Forth sind getrennte Welten, und doch haben sie einiges gemeinsam. Beide beruhen auf Worten ('Befehle' in Tcl), die ihre eigene Sache machen und damit die Programmiersprache bestimmen. Der Unterschied ist die Notation. Tcl ist prefix (und infix in der Arithmetik). Ein postfix–Tcl wäre Forth.

Ich verwende Tcl seit meinem Umzug von DOS auf GUI's, weil es das bietet, was Forth nicht hat: Ein volles Software–Ökosystem auf dem Desktop, identisch auf allen Plattformen. Inklusive dem grafischen Toolkit Tk, das schon einige Forth–Programmierer zu Verbindungen mit Tcl gereizt hat.

Doch wer sein Softwareleben in postfix verbracht hat, wird mit Tcl nicht völlig glücklich, und ich freue mich daher, ein postfix–Tcl vorzustellen, das den Reichtum von Tcl in Forth zugänglich macht.

Ich habe die einfachste Lösung gewählt. Forth ist als ein Paket über Tcl geladen und realisiert Interpreter/Compiler, Datenstack und Dictionary in Tcl. Anders gesagt: TclForth ist ein Forth, das Tcl als Muttersprache (native language) benutzt.

Code- und Colonworte werden als Tcl–Prozeduren realisiert, die ihre Argumente nun nicht mehr aus dem Script parsen sondern aus dem Stack holen. Damit ist die weitere Verarbeitung Tcl überlassen. Der Bytecode–Interpreter von Tcl ist der innere Interpreter von TclForth.

Das funktioniert auch in größeren Anwendungen erstaunlich flüssig.

TclForth ist verfügbar in `https://code.google.com/p/tclforth/` als Apps für Windows und OS–X und als Source für Tcl in Linux. TclForth ist open–source wie Tcl.

Wolf Wejgaard

Floating-Point

> Reliable portable numerical software was becoming more expensive to develop than anyone but AT&T and the Pentagon could afford. As microprocessors began to proliferate, so did fears that some day soon nobody would be able to afford it. (An Interview with the Old Man of Floating-Point; Reminiscences elicited from William Kahan by Charles Severance. 20 Feb. 1998)

Aus: `http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`

Wo stehen wir da heute eigentlich mit Forth?

Oder wird DEC64 eine Rolle spielen im Standard?

> DEC64 is intended to be the only number type in the next generation of application programming languages.

`http://www.dec64.org/`

Ich würde mich freuen, hierzu was zu lesen in unserer Zeitung.

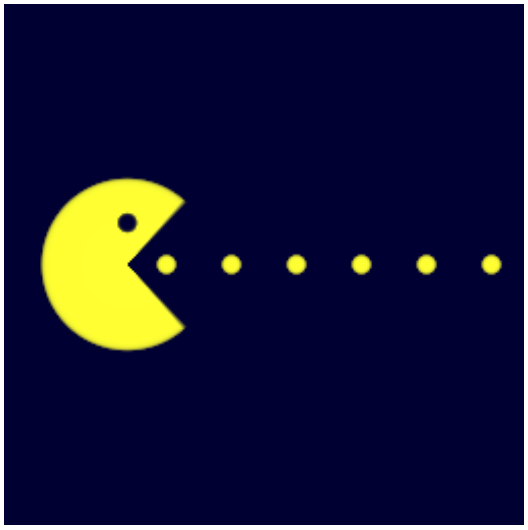
Michael Kalus

Es ist ein Teil des IEEE754-Standards auch in Forth standardisiert, der Rest wartet auf einen besser ausgearbeiteten RFD; da ist auch schon einige Arbeit hineingesteckt worden. Siehe `http://www.forth200x.org/ieee-fp.txt`.

DEC64 ist IMHO außerhalb des Rechnens mit Geld Quatsch. Und für das Rechnen mit Geld, bei dem Zahlen mit einer spezifizierten Anzahl dezimaler Nachkommastellen gesetzlich vorgeschrieben sind, sind heute einfach lang genuge Zahlen das Mittel der Wahl (etwa 128 Bits, also Doubles in einem 64- Bit-System).

Dezimalzahlen sind nicht „richtiger" als Binärzahlen, es wird nur in bestimmten Fällen anders gerundet. Gerundet werden muss immer, die Ergebnisse von Fließkommarechnungen sind immer Näherungen. Relevant ist das mit dem Runden nur dann, wenn es vom Gesetzgeber vorgeschrieben ist — und das ist bei der Buchhaltung eben der Fall.

Bernd Paysan



Haiku — 俳句

Ein rechter Haijin versucht es in jeder Sprache. Also auch in Forth. Auf BradN's wiederbelebtem blog wird man fündig.

> A Forth Haiku is an attempt to mix mathematics, art, and the Forth programming language. It resembles a texture shader, however, the emphasis is on direct expression in the resulting image.

Wie das geht? Im Browser läuft eine Forth-Maschine in Java-Skript und berechnet für jedes Pixel eines kleinen Quadrates das Forth Haiku. Dieses Forth im Browser kennt nur wenige Worte, mit denen man dann neue Forthworte machen und diese dann ausführen kann. Das gibt dann z.B. so etwas wie den PACKMAN. Und wenn der Browser das Skript gar nicht kann, so wie mein altes Firefox unter OSX, dann sieht man gar nix. Selbst

Chrome unter Win7 zeigte nur die Bilder, nicht die Animation. Carstens Firefox unter Linux aber konnte es - danke für die gute Unterhaltung damit. mk

Link: `http://forthsalon.appspot.com/`

```
Core words (Glossary):
x y t push pop dup over 2dup drop swap rot
-rot = <> < > <= >= and or not min max + -
* / mod pow atan2 negate sin cos tan log
exp sqrt floor ceil abs pi z+ z* random :

PACKMAN
: d dup ;
: m 1 min ;
: f d floor - ;
: c cos abs ;
: j t 4 + 2 * x 8 * floor 8 / + 4 *
  c 2 / t 4 + 2 / c 4 ** * - ;
: a 1 x x 8 * floor 0.5 + 8 / - d * y ;
: b - d * + sqrt 50 * 8 ** ;
: p x t 4 + pi / f 1.6 * - 0.2 + ;
: v t 4 + pi 2 * / f ;
a j 0.5 b -
v d 0.5 < * 4 * m *
1 p d * y 0.5 - d * + 36 * 30 ** m -
y 0.5 - p atan2 abs t 10 * c 0.8 * -
16 * m * 0 max
a 0.5 b - 0 max d p 16 * < * +
p d * y 0.58 b m *
v 0.5 >= *
+ d 0.2
```

Zen of LaunchPad - 430eForth

(Auszugsweise; Vollständige Fassung in der Redaktion erhältlich. 14.10.2014.)

> 430eforth is case sensitive. It does not have DO-LOOP. It uses FOR-NEXT. It is not ANSI standard, but it is ANSI compatible. It is a direct descendant of figForth, in spirit and in implementation.
>
> ... I went through the whole 430eForth system to optimize it. I first deleted all the commands I had never used, and the number of commands was reduced from 224 to 180. I then re-coded as many high level commands to machine instruction primitive commands, and the number of primitive commands was increased from 31 to 62. I also removed the headers of all the commands which are only used in building the Forth interpreter and compiler. These commands are never used in application programming, and are better hidden from the user. Only 145 commands are exposed to the user. I used the 17 Forth lessons to exercise the system, to make sure that the new eForth system can compile all the lessons correctly. The lessons use 60 commands, which I suppose are the most commonly used Forth commands. The new 430eForth system occupies 5592 bytes of flash memory in MSP430G2553. I believe this is the smallest Forth system capable of developing substantial applications on the MSP430G2 LaunchPad kit. ...
>
> The new 430eforth v3.1 is released in two forms: a full system with assembly source code 430eforth_8.asm to be assembled and tested with Code Composer Studio, available on www.offete.com for $25; and a free system with 430eforth_8.a43 object code with 430eforth–IDE. The free system can also be downloaded from www.offete.com. Both systems are in the public domain and you can use them freely to develop private or commercial applications.
>
> *Chen-Hanson Ting*

Link: `http://www.offete.com/`

AmForth unter GPLv3

AmForth ist ab der nächsten Release unter GPLv3. Dabei ist auf der Mailing–Liste eine heftige Diskussion entbrannt, weshalb ich die Gelegenheit nutze, als GNU–Forth–Maintainer ein Statement zu meinem Verständnis der Rechtslage zu geben. Das ist natürlich keine Rechtsberatung, aber mit meiner Autorität versehen.

Die GPLv2 und GPLv3 verfolgen das gleiche Ziel: Die Gewährleistung der vier Freiheiten von Software (nutzen, studieren, weitergeben, ändern). Der ganze juristische Text ist nur der „Code“, um dieses Ziel zu erreichen. Die GPLv2 hat einige Bugs, die unter manchen (relativ obskuren) Umständen dazu führen, dass das Ziel nicht erreicht wird. Die GPLv3 ist ein kompletter Rewrite mit dem Ziel, diese Bugs zu fixen. Insbesondere ist die GPLv3 deutlich verständlicher geschrieben, weshalb es bei diesen Umstellungen immer Leute gibt, die zum ersten Mal verstehen, was die GPL eigentlich will, und die deshalb total überrascht sind. Die GPLv3 ändert für den, der nach Treu und Glauben freie Software nutzt und entwickelt, überhaupt nichts.

Da die GPL das Urheberrecht nutzt, um freie Software zu promoten, wo es geht, gibt es natürlich auch Umstände, unter denen die GPL das nicht kann. So hat z.B. die GPL keinen Einfluss auf die Lizenzsituation des Compilats eines C–Compilers. Der Übersetzungsvorgang ist automatisch, also kein Werk im Sinne des Urheberrechts, und der C–Compiler wird nicht Teil des Compilats (nur unbedeutende Teile wie `crt0.o`). Das erlaubt es, den GCC für allerlei proprietäre Software zu verwenden.

Bei Forth ist das anders: Ein Forth–Compiler wird durch die Anwendung *immer* erweitert. Gibt man die Anwendung getrennt vom Compiler weiter, als Quelltext, ist sie nicht von der Lizenz betroffen, gibt man sie als Image weiter oder in Form eines geflashten Chips, handelt es sich um ein abgeleitetes Werk, da der Compiler ja komplett enthalten ist. Deshalb kann man ein freies Forth nicht für proprietären Code verwenden, ohne eine Lizenzausnahme mit den Autoren vereinbart zu haben.

Das gilt für andere Script–Sprachen ebenso, wobei diese meist nicht die Möglichkeit bieten, Images abzuspeichern. Insofern stellt sich die Frage dann gar nicht. Selbstverständlich fördert die Free Software Foundation auch dort freie Software.

Bernd Paysan

LPC1114FN28

32kB flash, 4kB SRAM, DIP28 package

> The LPC1114FN28 is an ARM Cortex-M0 based, low-cost 32-bit MCU, designed for 8/16-bit microcontroller applications, offering performance, low power, simple instruction set and memory addressing together with reduced code size compared to existing 8/16-bit architectures. The LPC1114FN28 operates at CPU frequencies of up to 50 MHz. The LPC1114FN28 includes up to 32 kB of flash memory, up to 4 kB of data memory, one Fastmode Plus I^2C-bus interface, one RS-485/EIA-485 UART, one SPI interface with SSP features, four general purpose counter/timers, a 10-bit ADC, and up to 22 general purpose I/O pins. ...

Und das Schönste: NPX besinnt sich auf die DIP28–Gehäusetradition. Bin gespannt, wie lange es dauert, bis Forth da drauf ist. :-)

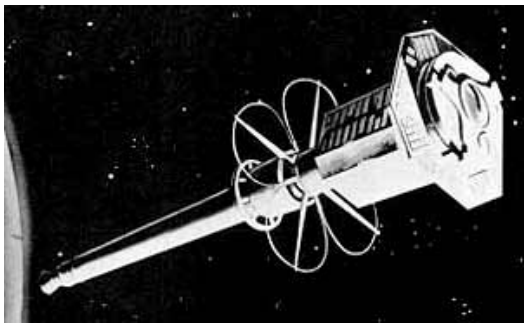
`http://www.nxp.com/products/microcontrollers/cortex_m0_m0/lpc1100/LPC1114FN28.html`

Michael Kalus

Vom Explorer 11 zu Philae

Herzlichen Glückwunsch, lieber Chuck!

Du hast 1958 mit einem Programm zur Bestimmung der Flugbahnen von Satelliten begonnen, und 1960 deinen Bachelor of Science in Physik mit einer Arbeit über Datenreduktion für den Gammastrahlen-Satelliten Explorer 11 gemacht. Es folgte 1965 ein Echtzeit-Gaschromatograph auf deinem ersten Minicomputer.



Deine spezielle Programmiersprache, mit der du 1970 das 11m-Teleskop und den Empfänger des National Radio Astronomy Observatory (NRAO) am Kitt Peak USA für ein nationales Programm zur Beobachtung und Aufzeichnung von Millimeterwellen im All steuern konntest, wurde 1980 vom Magazin BYTE mit einem Sonderheft bedacht, „The Forth Language“.

Der nächste Schritt war, einen Mikroprozessor zu entwickeln, der die Grundbefehle dieser Programmiersprache als Maschinencode in Silizium implementierte - zu jener Zeit eine unvorstellbare Herausforderung. Aber es gelang Dir, und der erste dieser Mikroprozessoren, der Novix NC4000, hatte nur 4000 CMOS Gatter, mit denen das realisiert werden konnte!

Auf der Basis des NC4000 entwickelte HARRIS Semiconductor die RTX2000 Familie, die neben der enormen Leistungsfähigkeit äußerst geringen Stromverbrauch hat und Raumfahrt-tauglich ist. Nun sind diese Mikroprozessoren dabei, mit der Raumsonde Philae nach einer zehnjährigen Reise den Kometen Churyumov-Gerasimenko zu erforschen. Auch ein Massenspektrometer, kontrolliert von einem RTX2010, ist mit dabei. Am 12. November 2014 ist Philae gelandet. Eine erstaunliche Erfolgsgeschichte der Rosetta-Mission - und von Deinem Forth!

db,mk

Quellen

`http://de.wikipedia.org/wiki/Charles_H._Moore`
`http://en.wikipedia.org/wiki/RTX2000` `http://www.computinghistory.org.uk/det/25228/Novix-NC4000-Beta-Board/`

Notes on STM32eForth720 Implementation

Chen–Hanson Ting

STMicroelectronics recently released a very nice microcontroller kit STM32F4-Discovery, for about $20. It is a very power microcontroller development system. The STM32F407 chip on Discovery Kit has 1 MB flash memory and 192 KB of RAM. It is a chip I do not feel being constrained by RAM memory in implementing a Forth on it.

STM32eForth720.s assembly source code is assembled on Keil's uVision 5.10. The object file is 8492 bytes long. The object code in stm32eforth720.hex can be downloaded free from `www.offete.com`. The complete package for STM32F4-Discovery Kit is available for $25.

I kept a notebook while porting eForth to the Discovery Kit. I hope it interesting to you if you are looking for the newest and greatest ARM chips.

HyperTerminal Setup

Discovery Kit does not have a USART port per se, even STM32F407 chip has 6 USART devices in it. Stm32eforth720 uses USART1 port to communication with a terminal. On STM32F407VG, USART1 can be configured to use either Pins PA9-10 or PB6-7 for communication. Since the USB on CN5 is using PA9-10 pins, I have to use PB6-7 for eForth. I am using a separate PC to run HyperTerminal through a USB to serial converter, which happens to be an Arduino Uno Kit. Arduino Uno Kit has a integrated USB to serial converter connecting its STmega328P chip to the host PC. To use its USB to serial converter, I remove the ATmega328P chip, and connect the PB6 (TX) on Discovery to D1 port on Arduino, the PB7 (RX) on Discovery to D0 port on Arduino. A ground wire connects the ground pins on both boards.

HyperTerminal on PC is configured at 115200 baud, 1 start bit, 8 data bits, 1 stop bit, no parity, no flow control. The USART1 on STM32F407 is configured similarly. STM32F407 is clocked by its high speed internal clock HSI at 16 MHz on reset. Since this HSI is factory trimmed to 1% accuracy, it is adequate to provide reliable communication on USART1.

Hardware Initialization

The only IO device stm32eforth720 really needs is an USART. It has to be initialized properly to communication with a terminal. In STM32F407, however, I have to initialize USART1, GPIOB which lends pins to USART1, and the Reset and Clock Control RCC to clock the USART1 and GPIOB. I want to lit up the on-board LED's to give visual confirmation on booting up eForth. Hence, I need to initialize port GPIOD to drive the LEDs as well.

If we will be adding new commands to the flash memory, it must be unlocked by writing two specific consecutive words into the Flash Key Registers FLASH_KEYR.

eForth code is stored in flash memory. Its entirety is copied from flash memory to RAM. The RAM memory is remapped to Page 0, and eForth is running in RAM.

The CPU registers R1, R2, R3 and R5 are initialized as the parameter stack pointer SP, the return stack pointer RP, the user variable array pointer UP, and top of parameter stack TOS, respectively. The hardware interrupt stack pointer R13 is not used, but it is initialized to point to CCM memory at 0x10000400 anyway.

Virtual Memory of STM32F407

STM32F407 has this memory map:

Virtual Memory	`0x00000000-000FFFFF`
Flash Memory	`0x08000000-080FFFFF`
Core Coupled Memory	`0x10000000-1000FFFF`
System Memory	`0x1FFF0000-1FFF77FF`
RAM Memory	`0x20000000-2001FFFF`
IO Devices	`0x40000000-FFFFFFFF`

1 MB of memory space from 0 to 0xFFFFF is the virtual memory, which can be mapped or aliased to Flash memory, or RAM memory. Identical code in these physical memories can assume logical addresses in the virtual memory or Page 0 memory, and can be executed as though it is in Page 0 physical memory.

Mapping RAM memory to Page 0 is especially convenient for stm32eforth, because new commands can be easily added in the RAM memory to extend the command dictionary. It is possible to have eForth in the flash memory and add new command to the flash memory directly. However, it requires a different set of memory store commands for the RAM memory and for the flash memory, and the system becomes more complicated than it should be.

eForth dictionary is initially stored in the flash memory. Upon booting, eForth copies the entire dictionary from flash memory to RAM memory, re-maps RAM memory to Page 0, and executes from Page 0. The dictionary can grow at will, as new commands are added to RAM memory mapped to Page 0. When an application is complete, the entire dictionary including added commands can be saved back to the flash memory. When re-booted, the new eForth system will be activated. This way, we can develop new application interactively in RAM memory, and then save the results in flash for final product to be released.

To get eForth executing in Page 0, I had to cheat the assembler so that it assembles code using Page 0 addresses,

instead of addresses in the flash memory, as the linker generally places code in the flash memory, starting at 0x8000000. I used a set of mapping constant to ask the assembler-linker to produce the Page 0 addresses I need.

```
;RAMOFFSET EQU 0x00000000 ;absolute
;MAPOFFSET EQU 0x00000000 ;absolute
RAMOFFSET EQU 0x20000000 ;remap
MAPOFFSET EQU 0x08000000 ;remap
```

All the STM32F4 transfer instructions, branching and conditional branching, use relative addressing, and are assembled correctly for physical memory and for virtual memory. The high level branching commands in eForth use absolute addresses. So are the link field addresses which link the eForth commands as a linear list. These absolute addresses have to be corrected by the constant in MAPOFFSET. If the code is executed in the physical flash memory, MAPOFFSET is 0. If the code is executed in the virtual memory, MAPOFFSET must be 0x8000000. User variables stored in RAM must be so corrected with RAMOFFSET.

Flash Memory Commands

Here are eForth commands to manipulate flash memory:

I! (n a –) Store 32 bit data n into flash memory location a.

ERASE_SECTOR (sector –) Erase one sector (0-11) of flash memory.

TURNKEY (–) Copy eForth dictionary from RAM to flash.

UNLOCK (–) Unlock flash memory to be writable.

System Startup

In the software package DEMO provided by STmicroelectronic with STM32F407-Discovery, there is a file startup_stm32f40_41xxx.s, which is loaded by the MDK tool chain to boot up the STM32F407 chip. 29 KB long, and looks awesome. It sets up the interrupt vector table, with lots of initial interrupt handlers which loop back to themselves. It also sets up interrupt hardware stack and the heap.

In stm32eforth720, we do not allow interrupts, do not use the interrupt stack, and do not use the heap. So, the startup code is reduced to:

```
AREA          RESET, CODE, READONLY
              THUMB
              EXPORT       __Vectors
                  ; linker needs it
              EXPORT       Reset_Handler
                  ; linker needs it
__Vectors DCD             0x10000400
                  ; Top of hardware stack in CCM
              DCD          Reset_Handler
                  ; Reset Handler
              ENTRY
Reset_Handler
              BL           InitDevices
                  ; RCC, GPIOs, USART1
              BL           UNLOCK
                  ; unlock flash memory
              BL           REMAP
                  ; remap RAM to page 0
              LDR          R0,=COLD-MAPOFFSET
                  ; start Forth
              BX R0
              ALIGN
```

Initialize Devices

Stm32eforth720 uses only USART1 for communication, and GPIOD to lit up the LEDs. However, USART1 uses pins PB6-7 for TX and RX; therefore, GPIOB has to be initialized. All three devices need to be clocked, and the Reset Clock Controller RCC must be initialized. Here are the code of initDevices.

Remap RAM memory

The primary objective in stm32eForth720 is to run in RAM, so that new command can be added to the dictionary freely. Once an application is completely debugged, the entire dictionary can then be saved into the flash memory to become a turnkey system, ready to run at power-up. The REMAP routine first copies the eForth dictionary image from flash memory to RAM memory. Then, it remaps RAM memory to Page 0, and starts eForth executing in RAM. To remap, we simply write a 3 into the System Configuration Register SYSCFG.

```
; Remap eForth to execute from RAM
; Copy eForth from flash to RAM
REMAP mov r0,#0x8000000 ; flash
mov r1,#0x20000000 ; RAM
add r2,r1,#0x10000 ; copy 64KB
REMAP1 cmp r1, r2
ldrcc r3, [r0], #4
strcc r3, [r1], #4
bcc REMAP1
; Remap RAM to page 0
movw R0,#0x3800 ; SYSCFG register
movt R0,#0x4001
mov R1,#3
str R1,[R0,#0] ; map RAM to page 0
bx lr
align
```

THUMB2—Death of RISC

The ARM architecture was hailed the prince of RISC, as its original name says: Acorn RISC Machine. The major disadvantage of RISC is its poor coding density. A 32-bit instruction does not do much work. Lots of bits in the instruction, like the 4-bit condition field, are wasted. ARM Holdings tried mightily implementing the THUMB instruction set to complement the ARM instruction set. In the end, the THUMB is waggling the ARM, and the

THUMB2 instruction set basically gets rid of the ARM instructions. THUMB2 is clearly a CISC architecture. Cortex M4 core inside STM32F407 is an extremely complicated instruction set computer. Intel had proved that the RISC architecture is of no special value, and ARM Holdings concurred.

The Dire Consequences of the Moore's Law

Moore's Law marches on, and more and more circuits are crowded into microcontrollers. In the last 15 years, I had programmed many ARM chips, and had watched with amazement the progress of the ARM chips. My approach had always been to port an eForth system onto the chips and tried to make good use of the chips.

A couple of years ago, I told my friends in the Silicon Valley FIG and Taiwan FIG that I had to really retire from Forth programming. I did, and worked peacefully on translating Bach's cantatas from German to Chinese, and fitting Tang poems into Schubert's songs, and many other things I had neglected all the years. Then, last month, friends in Taiwan FIG sent me this ForthDuino Board, which they used to control a laser cutting machine to make PC boards. It had footprints of IO sockets of Arduino board and MSP430 LaunchPad. I guess that they intended to suck in all applications from Arduino and LaunchPad. I was told that the ARM chip on ForthDuino is the same one used in the STM32F4-Discovery Kit. Looking up the STM32F407 chip, I was shocked to see so much memory, and so many IO devices. 1 MB of flash and 192 KB of RAM. It is a WOW chip, and in desperate need of a good eForth system.

So. I re-open my workbench, unpacked my tools, download all necessary IDE and programming tools. But, the world has changed since I stopped watching. Keil is still there, but its toolchain became uVision5. STM32F4 is no longer an ARM chip. It is a Cortex M4 chip. There is no ARM in STM32F4. All that's left is a THUMB.

There is no simple examples to guide you, to start your exploration. The Demo project provided with STAM32F4-Discovery Kit is a huge package with 7 folders and 31 files. There is no clear entry point. I spent 3 weeks wandering around in the hardware and software maze, looking for an entry point. The great breakthrough was when I realized that I only had to set up the reset vector correctly, everything would follow smoothly from that point on. Throw away all the header files, init files, device driver files. I only need one assembly file to do what I have to do.

Since STM32F4 is no longer an ARM7 chip.It is not necessary to keep the name ARM in my eForth implementations. I planned and completed 4 versions of eForth for this chip:

A while ago, I was amazed at the 566 page reference manual of ATmega328 from Atmel, which is a lowly 8-bit microcontroller. The reference manual of STM32F407 is 1713 pages thick. How can anybody wading through the documents to get a handle on this chip and all these many peripheral devices?

I opened the Demo project for the STM32F407-Discovery Board on Keil's uVision5. In the Project panel I counted 7 folders with 31 files in them. Just for a Demo! It is true that the Demo does a lot of interesting things, like reading the 3-axis accelerometer and the USB connection to PC. I have great sympathy for people who get this kit and are confronted by this huge document and software mess.

The dire consequences of the Moore's Law and complexity beyond comprehension.

The only way to deal with this complexity is the Forth way. Or, put it more bluntly:

KISS Keep It Simple, Stupid!

The first thing is to put eForth on board. The 16 MHz high speed internal clock HSI in the chip is good enough for a USART. Forget about the fancy PLL that pushes the clock to 168 MHz. We can deal with it when we really need the speed. Just get the USART working, and we can walk into the guts of the microcontroller and actually control it through eForth.

What about interrupts, threads, heaps, preemptive task switching? All the great things this ARM/THUMB chip can do? Forget them! You will learn them in the senior year of computer engineering, if you have time to go to school. All these things can be added to eForth when you really need them.

eForth exposes all the memory and the registers to you. You can read them, and you can tinker with them. This is the way to study the peripheral devices and learn how to control them and make use of them. Focus on one device you will use. Read that chapter in the reference manual. Inspect the status and control registers. Flip bits in the control register and see what happens. Write short commands to perform the tasks you want. These commands will be called from your eventual applications.

Oddity of Thumb Transfer Instructions

In the transfer instructions, THUMB2 requires that the target addresses must be odd. Bit 0 of the address is deposit into the T bit in the EPSR status register, indicating that it is a Thumb instruction. The actual address is on the 2 byte half word boundary. Much grief was encountered when I implemented eForth, as I tried to align addresses to the 4 bytes boundary, as any nice 32-bit instruction would do. If bit 0 is not set in these addresses, the CPU may work correctly, and it may also crash. I learnt the hard lesson when building turnkey systems. The correct procedure is:

```
; load Lesson16.txt
; load Lesson17.txt
0 ERASE_SECTOR
' GUESS 1+ 'BOOT !
TURNKEY
```

In most cases, eForth takes care of this oddity. But, when you are using addresses explicitly, make sure bit 0 in the address is set before jumping to it. (Well, storing an even address in 'BOOT should be considered a bug in eForth. So, I corrected it in the final release.)

eForth1 and eForth2

The original eForth Model was designed by Bill Muench in 1990. It was based on the Direct Thread Forth Model, in which the body of a high level Forth command contained a list of execution addresses, preceded by a CALL NEST machine code. Bill was very ambitious in that he laid down hooks for multitasking and the CATCH-THROW mechanisms for error handling. List of execution addresses were very easy for porting to other processors. Indeed, many people did port this model to about 30 different processors. At that time, assemblers for these processors were not easily available and very different. This simple model was very easy to be adapted to a particular assembler. In a few cases, MASM from Microsoft was used to assemble eForth for a different processor.

Getting into this century, I ported eForth for many microcontrollers at work. With good native assemblers, I was able to optimize eForth for performance necessary in actual products. I used the Subroutine Thread Model throughout, and realized many other advantages besides speed. Machine code can be mixed with subroutine calls. Interrupt service routines can be written in high level Forth. In all these applications, multitasking was not necessary and many user variables can be eliminated. The CATCH-THROW mechanism was not needed, and the error handling was greatly simplified. The cumulative result was eForth2, and earlier implementations were classified as eForth1.

eForth2 implementations were all written using native assemblers provided by microcontroller manufacturers.Forth commands which could be expressed in native machine instructions were so re-coded.

THUMB2 Instruction Set

When I started porting eForth to STM32F407-Discovery Board, I was not aware of the THUMB2 instruction set. I used sam7ef.s (for AT91SAM7x256 chip) and tried to assemble it under uVision5. Lots and lots of error messages. Total confusion. The first thing I noticed was that the startup_stm32f4_xxx.s file used a THUMB directive, and I used ARM directive in sam7ef.s. Changing the THUMB directive to ASM caused more errors. Changing the ARM directive to THUMB, the assembler was much happier, but still shot lots of errors and warnings at me.

Then I found that ARM Holdings changed the CPU core behind my back when I was not watching. Their chips are ARM chips no more. They are Cortex-M4 chips with only Thumb2 instructions. They assembler was also changed to UAL, as stated in one of its manuals:

> *Unified Assembler Language* (UAL) is a common syntax for ARM and Thumb instructions. It supersedes earlier versions of both the ARM and Thumb assembler languages.

These are errors which I had to correct:

> B<addr> is a 2-byte Thumb instruction. It causes following instructions to be misaligned.
>
> It must be changed to B.W<addr>, to retain 4 byte word alignment.
>
> Target address in Forth transfer commands must have bit 0 set.
>
> RSC (Reverse subtract with carry) disappeared. It was used only in DENEGATE.

I had made many mistakes on my own. I had to upgrade eForth1 to eForth2, though most changes were deleting things I did not need. During the process, I really appreciated the debugger in the Keil uVision5. It allowed me to set up to 6 break points freely. Watching CPU registers and registers in any device while single stepping the assembly code was very helpful. In the end, it is very pleasing to see stm32eForth signed on and processed my command correctly.

Branch and Link

Subroutine call uses the Branch and Link BL<addr> instruction. All high level Forth commands were assembled as BL instructions. BL instruction, as invented in the RISC architecture, assumed a return stack of 1 level. If the called subroutine had to call other subroutines, the return address in LR had to be saved on a real return stack of adequate depth. (In eForth, the return stack and the parameter stack run to about 20 levels deep. 64 levels are reserved for the return stack. About 16K levels are available for the parameter stack.)

I watched the disassembled BL instructions while single stepping through the code, but could not figure out how the instructions were encoded. Only when I was testing the decompiler command SEE, I had to figure it out without a shiver of doubt. It is composed of two 16-bit THUMB instructions in the form of:

```
|11111| addr bits 22-12||11110|addr bits 11-1 |
|   byte 1  |  byte 0  ||  byte 3  |  byte 2  |
```

Very strange, indeed! But, I was able to shift the bits around and eventually get the correct address back.

Build Turnkey Application

The goal of Firmware Engineering is to build turnkey system, which will execute application program on boot up. Everything has to be embedded. Nothing can be left to the weather. CPU has to be initialized correctly. Memory has to be initialized correctly. All peripheral devices have to be initialized correctly. You application must run correctly. That's a tall order. Has anybody taught you how to prove the correctness of your program?

This is my way to build reliable applications. I do not yet have an application to try out on STM32F407-Discovery, but I had done many applications on many other eForth systems. I experimented each device I had to use. I wrote short Forth commands to initialize it, and to do simple things. These commands were put in a short text file, which could be loaded on eForth through HyperTerminal. Many of these short text files were later combined into a longer file to ease loading. Each short command was tested exhaustively so that its correctness was of no doubt. These proven short commands were used to build short commands at a higher level, and they were tested exhaustively. When a good collection of commands were all verified, they were loaded into the eForth system, and the entire eForth system was saved in flash memory or in ROM memory. New components of the application were added gradually. Eventually, the entire application was committed to ROM and the product was shipped.

This turnkey capability is built into stm32eforth720.

You have to use Keil's uVision5 to assemble stm32eforth720.s and use its debugger to download stem32eforth into the flash memory of STM32F407 chip. You can add new commands to the stm32eforth720.s file, but this is definitely now a good way to build significant applications.

After stm32eforth boots up and signs on HyperTerminal, you can write short commands and test them interactively. Useful commands are written into text files which can be downloaded to STM32F407 chip through the Transfer menu on HyperTerminal. Remember to give 200 ms delay after sending each line of text, so that stm32eforth has enough time to process the commands in the line of text.

After downloading all the application files, select a command you want stm32eforth to execute on boot up, and store its execution address plus 1 into the user variable 'BOOT. Type "HERE ." to find out the size of your application, and erase that many sectors of flash memory. As it is currently configured, stm32eforth uses 64KB of memory. That covers the first 4 sectors of flash memory in STM32F407. Finally, type TURNKEY to copy the entire eForth system to flash memory. When STM32F407 is reset or boots up, your application will run.

Case Sensitivity

Stm32eforth is case sensitive. The names of most commands are in upper case. You can name your commands in either case, or mix the cases. It is possible to make the system case insensitive for novices to learn it, but I don't thing novices could survive very long on STM32F407-Discovery. It is such a complicated system, and you have to have the determination to master it. Case sensitivity is a small pebble on the road.

Case sensitive system has an advantage in that you have greater freedom in naming your commands, and use cases to distinguish classes of commands.

Stm32eforth720 uses all 8 bits in a character. It thus allows the use of other more complicated character sets like Chinese Unicode.

Irreducible Complexity

Stm32eforth720 is the simplest system for Firmware Engineering, or Embedded System Programming. For 25 years, I have worked on it and tried to make it simpler. It is still very complicated, but the complexity is necessary to support this integrated, interactive interpreter/-compiler/debugger/OS. This is what I call Irreducible Complexity. It is the smallest system which can grow to handle any application, however complex it may be.

As Laotze said in his Tao Te Ching, Chapter 48:

> For knowledge, add a bit a day.
> For wisdom, delete a bit a day.
> Delete until there is nothing.
> Then, you can do everything.
>
> 爲學日益，爲道日損。
> 損之又損，以至於無爲。
> 無爲而無不爲。
> 取天下常以無事，及其有事，不足以取天下。

Nothingness is a great idea, but not very practical in firmware engineering. I think Albert Einstein, a practical scientist, said it better:

> Everything should be made as simple as possible,
> but not simpler.

stm32eforth720 is assembled to a 8492 byte image. It seeks and, I believe, has achieved irreducible complexity. Things cannot be made any simpler.

eForth leaves the least footprint in your mind. With this understanding, and of course stm32eforth720, you can make the STM32F407-Discovery do whatever you want it to do.

Grundlegende Experimente mit einer MCU — Taster abfragen

Michael Kalus

Ist der Taster gedrückt oder nicht? Das klingt nach einer einfachen Sache. Um das zu überprüfen, wurde das Verhalten zweier Taster unterschiedlicher Bauart untersucht. In den anschließenden Übungen wird ein Taster abgefragt. Dabei soll erreicht werden, dass der Taster wieder losgelassen werden muss, um erneut aktiv sein zu dürfen.

0.1 Tasterverhalten

Die **Abb.2** zeigt das Verhalten des LaunchPad–Tasters, aufgenommen mit einem Speicheroszilloskop. Da ist eine recht saubere fallende Flanke, wenn der Taster gedrückt wird. Es gibt einen deutlichen Druckpunkt bei der Bedienung, beim Drücken wie beim Loslassen „knackt“ es etwas. Doch das ist nicht bei allen Tastern so. Das Experiment mit einem robusten Taster (**Abb.1b**) zeigt, dass solch ein Taster keinen sauberen Schaltzeitpunkt hat, der Pegel am Portpin fällt nicht nur einmal, sondern springt eine Weile hin und her. Dieser Taster prellt. Für etwa 4.5ms gibt er immer wieder H- und L-Pegel in rascher Folge (**Abb.3**).

Um auch von solch einem Taster ein klares „gedrückt“ oder „nicht-gedrückt“ zu erhalten, muss entprellt werden, was per Software gemacht werden kann. Das folgende kleine Forthprogramm setzt das exemplarisch um. Damit ist es möglich zu ermitteln, ob der Taster erneut gedrückt wurde, oder noch nicht . Bei weiter gedrückt gehaltenem Taster erfolgt keine Reaktion mehr, er muss erst losgelassen werden, um den nächsten Tastendruck einlesen zu können. Das funktioniert also wie ein Auffangregister (Latch) mit Reset.

Abbildung 1: Zwei Typen von Tastern: a) Drucktaster wie auf dem LaunchPad, 24V/DC. b) robuster Drucktaster 250V/AC.

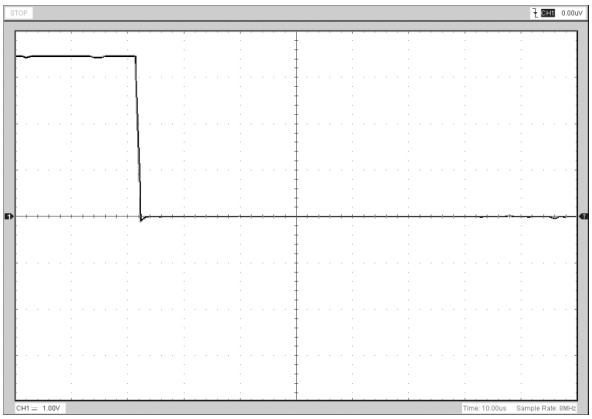


Abbildung 2: Saubere Flanke am Launchpad-Taster Typ a)

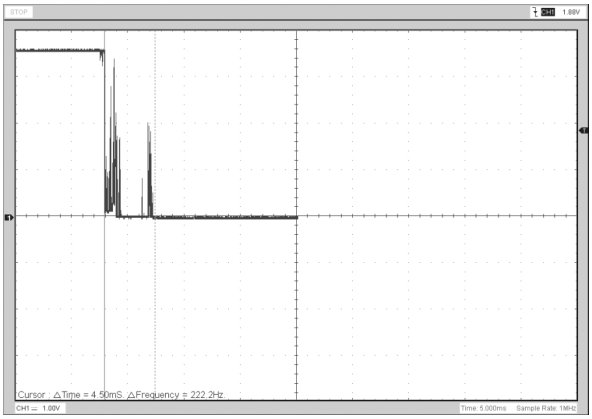


Abbildung 3: Taster Typ b) prellt

0.2 Schaltung

Die Schaltung für das Experiment ist denkbar einfach:

P1.3 —— S2 —— GND

Der Portpin wird als digitaler Eingang geschaltet. Im Beispiel ist Pin3 von Port1 der MCU MSP430G2553 auf dem LaunchPad über den Taster S2 direkt mit Masse (GND) verbunden. Das geht ohne weiteres. Ein interner Pullup-Widerstand liefert den positiven Spannungspegel.

offen	gedrückt
false	true

H	L
3.6V	0V

Tabelle 1: Logik des Tasters

0.3 Digitalen Eingang pollen

Um das Ganze zu demonstrieren, habe ich einen Binärzähler gemacht. Mit jedem Tastendruck wird der Zähler um eins erhöht. Das klappt auch, wenn statt des LaunchPad-Tasters der „olle“ Taster angeschlossen wird. Oder mit einem einfachen Draht an Stelle des Tasters S2, mit dem man die Masse antippt. Probiert's mal aus. Prinzip: Warten bis der Pegel sicher unten ist. Dann ist der Taster stetig gedrückt. Die nötige Wartezeit kann ausprobiert werden. Dann muss noch ein Merker mitgeführt werden, an dem man erkennen kann, das der Taster erneut gedrückt worden ist, also der Pegel zwischenzeitlich wieder hoch war. Der Merker wurde in diesem Beipiel auf dem Stack mitgeführt.

0.4 Interruptflag abfragen

Die Hardware der meisten MCUs unterstützt so etwas bereits, indem sie Flanken am Portpin erkennen kann. Dieses Pin-Verhalten muss per Software angewählt werden. Dazu wird ein Flag in einem Register gesetzt. Im MSP430 heißt es Portpin-Interrupt-Flag-Register P1IFG. Dieses Flag wird dann gesetzt, wenn eine Flanke im Spannungsverlauf am Input-Pin kommt. Man kann einstellen, ob eine steigende oder fallende Flanke der Auslöser sein soll. Einmal ausgelöst, bleibt das Flag solange gesetzt, bis man es per Software wieder zurücksetzt. Die Abfrage, ob der Taster gedrückt worden war, reduziert sich damit darauf, das Flag auszulesen. Ist es gesetzt, war der Taster gedrückt - vermutlich jedenfalls, solange man kein Rauschen oder Störpegel auf der Leitung zum Taster hat, die den Pin auch auslösen (triggern) könnten. Damit ist die Aufgabe also elegant zu lösen. Man muss lediglich etwas warten, bis das Prellen abgeklungen ist, bevor man den Trigger wieder scharf schaltet - reset des flag - um den nächsten Tastendruck mitzukriegen. Auch für diese Variante ist ein Binärzähler-Beispiel angegeben.

Weitere Experimente findet ihr auf der forth-ev Wiki-Seite mit den grundlegenden Experimenten zu einer MCU. Viel Vergnügen.

0.5 Link

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

0.6 Listing

```
\ Hochzaehlen einer Variablen.
\ An den beiden LaunchPad-LEDs darstellen.
\ Zählen jeweils auf einen S2-Tastendruck.

\ LED1 -- P1.0 red
\ LED2 -- P1.6 green
\ S2   -- P1.3 input
\ Die LEDs und S2 sind im 4e4th schon initialisiert.

\ Wert anzeigen an den beiden LEDs
HEX
: SETLEDS  \ n --
  \ Binaerwert an den LEDs darstellen.
  >r  \ in lokale Variable sichern.
  r@ 01 and
  IF green cset ELSE green cclr THEN   \ Bit0
  r> 02 and
  IF red   cset ELSE red   cclr THEN ; \ Bit1
variable OUT
: ?   @ u. ; \ adr --- \ Helfer, Variable zeigen.
: OUT>  \ Wert anzeigen.
  out @  setleds ;

\ Uebung1: Pegel am Pin von S2 abfragen.
HEX
FFFF constant TRUE
: 5XS2?  \ -- f  \ wahr, wenn Taster gedrueckt.
  true  5 zero
  DO 10 ms s2? and LOOP ; \ Zeit ausprobieren.
: TNEW \ x0 f0 -- x1 f1
  \ f1 wird nur wahr, wenn der Taster
  \ erneut gedrueckt wurde.
  2dup >r >r          \ R: -- x0 f0
  invert swap drop    \ f0 --> x1
  r> r>               \ x1 x0 f0 --
  2dup  and    IF 2drop true exit THEN
  2dup  and 0= IF 2drop zero exit THEN
  2drop zero ;
: INC  \ Binaerzaehler, beide LEDs am LaunchPad.
  zero out !  out>  \ Anzeige aus.
  zero BEGIN
  5xs2? tnew           \ Tastendruck abfragen.
  IF 1 out +! THEN  \ Binaer zaehlen
  out>              \ Display aktuell halten.
  key? UNTIL
  drop  key drop ; \ Ende

\ Uebung2: Flanke am Pin von S2 abfragen
HEX
: INITIO \ -- \ I/O initialisation
\ mask adr op
  08 22 ( P1DIR ) cclr \ P1 INs
  08 24 ( P1IES ) cset \ falling edge detect
  08 27 ( P1REN ) cset \ pullup selected
  08 21 ( P1OUT ) cset \ pullup enabled
;   INITIO

023 constant P1IFG
: S2E?  \ -- f   \ set on edge event.
  008 P1IFG cget ;
: S2RS  \ --     \ reset flag
  008 P1IFG cclr ;

: INC2  \ Binaerzaehler, beide LEDs am LaunchPad.
  initio zero out !  out>  \ Anzeige aus.
  s2rs BEGIN
  s2e?           \ Tastendruck abfragen.
  IF 1 out +! 10 ms s2rs THEN  \ binaer zaehlen
  out>           \ Display aktuell halten.
  key? UNTIL
  key drop ; \ Ende

( finis)
```

Recognizer – Interpreter dynamisch verändern

Matthias Trute

Der Titel ist nicht neu[1]. Recognizer sind für die Leser dieses Blattes nichts neues mehr[2]. In den letzten beiden Jahren wurden Erfahrungen gesammelt und ein konsensfähiges System ist entstanden. An der Grundidee hat sich nichts geändert. Darüber hinaus ist ein Forth 201x RFD in Vorbereitung, der einen neuen Wordset „Recognizer“ in den Forth Standard einführen könnte[3].

Warum das Ganze?

Die ursprüngliche Motivation, nachträglich Gleitkommazahlen in ein System, das die nicht kennt, einzubauen, besteht weiterhin. Das war und ist ein kleines Ziel, zumal es kaum jemand nutzen dürfte, da Gleitkommazahlen auf 8–Bit–Controllern einigermaßen exotisch sind. Umso mehr hat es mich erstaunt, dass die gforth-Gurus (Bernd) auf das Thema aufgesprungen sind und wahrhaft revolutionäre Ideen entwickelt haben. Im Zuge der Diskussionen auf dem IRC Channel #forth-ev und durch konkreten Code wurden zudem einige Punkte aus dem bestehenden Konzept hinterfragt und, ja, auch verbessert.

So fiel als Erstes die STATE–Awareness dem Redesign zum Opfer. Die neuen Recognizer benötigen STATE nicht mehr, weniger ist mehr. Der nächste Kritikpunkt war, dass ein Recognizer alles in einem Wort gemacht hat: Text analysieren, Text in Daten umwandeln und die Daten verarbeiten. Sowas ist unforthig, das gehört faktorisiert. Durch einen Umbau des Interpreters konnten diese Punkte gelöst werden, ohne dass sich die grundlegende Idee wesentlich geändert hat[1]. Er hat jetzt zwei Schnittstellen für Recognizer: Die eine ist das Parsen und Umwandeln der Textdaten. Die zweite ist die daran anschließende Nutzung der Daten je nach Laufzeitkontext: Interpretieren, Compilieren und Postponen. Und STATE? Das ist da, wo es hingehört: Im Interpreter und nur dort.

Parsen und Konvertieren

Das Parsen ist der erste Schritt auf dem Weg von den Textdaten des Interpreters hin zu verarbeitbaren Daten. Als Input dient ein einzelnes Wort. Ein Parser liefert nicht nur die Information, dass ein Wort bestimmten Kriterien genügt, sondern auch eine konvertierte Fassung der Ausgangsdaten. Wenn ein Wort den Kriterien für (z.B.) eine Zahl genügt, steht auch die Zahl selbst bereit. Ähnlich die Suche im Dictionary. Neben der Information „Ist enthalten“ gibt es auch das Execution Token (XT) und die Flags. Das *parsing word* eines Recognizers bündelt diese Informationen und liefert zusammen mit den Ergebnissen der Konvertierung auch die Typinformation für das Ergebnis. Die Typinformation wird nachfolgend auch als Methodentabelle bezeichnet. Mehr dazu weiter unten. Damit erhält der Interpreter zweierlei: Die Information, dass das Parsen geklappt hat, und Anweisungen darüber, wie die Daten verarbeitet werden sollen.

Als Beispiel soll der Wort–Recognizer vorgestellt werden. Er prüft, ob das übergebene Wort im Dictionary enthalten ist, und liefert im Erfolgsfall das Execution Token mitsamt Immediate–Flag.

Das *parsing word* heißt `rec:word`. Es wird mit der Adresse und der Länge des aktuellen Worts im `SOURCE` aufgerufen. Diese Information gibt es 1:1 an `FIND-NAME`[2] weiter. Das durchsucht das Dictionary mit seinen aktiven Wortlisten (`GET-ORDER`) und liefert die gewünschten Angaben: Ein Flag und ggf ein Execution Token. Flag 0 heißt nichts gefunden, −1 heißt normales Wort und +1 heißt immediate. Darauf aufbauend wird die Methodentabelle ausgewählt. Flag 0 wird zu `r:fail`, das *parsing word* konnte mit dem übergebenen String nichts anfangen. Anderenfalls wird das Ergebnis von `FIND-NAME` zusammen mit der Methodentabelle `r:word` zurückgegeben.

```
: rec:word
  ( addr len -- xt +/-1 r:word | r:fail )
   find-name ( -- xt +/-1 | 0 )
   ?dup if
     r:word
   else
     r:fail
   then ;
```

Über `r:word` sind drei Methoden erreichbar, die für ein Wort aus dem Dictionary anwendbar sind. Zuerst das bedingungslose Ausführen. Die Flaginformation ist hierbei uninteressant. Für das Compilieren und Postponen ist das Immediate Flag auszuwerten, dass heist „XT ausführen“, „XT abspeichern“ oder das Abspeichern des XT veranlassen.

```
\ Hilfswort
: immediate? 0> ;

\ execute action
:noname drop execute ;
\ compile action
:noname immediate? if
    compile,
  else
    execute
```

[1] Auch die Programmgröße ist nahezu unverändert geblieben, amforth ist da sehr kleinlich.

[2] `FIND-NAME` macht das gleiche wie `FIND`, nur hat es als Input ein addr/len Paar anstelle des counted strings bei `FIND`. Vielleicht haben die Götter ein Einsehen, so ein Wort in den Standard mit aufzunehmen.

```
  then ;
\ postpone action
:noname immediate? if
    postpone postpone
  then
  compile,
;
\ 3 XT auf dem Stack
recognizer: r:word
```

Per Konvention beginnen die parsing words mit `rec:`.

Der Interpreter

Der Interpreter hat zwei Aufgaben. Die erste ist die klassische Zerlegung des `SOURCE` in durch Whitespace begrenzte Worte und das Verwalten von `>IN`. Jedes Wort wird sodann über die Recognizer analysiert und entsprechend deren Ergebnis verarbeitet. Hier kommt der Methodentabelle `r:fail` eine doppelte Bedeutung zu: Sie signalisiert zum einen, dass der Recognizer das Wort nicht erkennen konnte und der Interpreter einen anderen aufrufen soll. Die zweite Bedeutung kommt zum Zuge, wenn kein Recognizer aktiv wurde. Für diesen Fall ist `r:fail` eine Methodentabelle, deren Methoden für alle drei Einsatzfälle das Gleiche machen: Einen Fehler melden, z.B. indem das Wort ausgegeben und Exception -13 ausgelöst wird.

Der zugehörige Interpreter sieht damit ziemlich einfach aus (aus amforth)

```
: interpret
  begin
    parse-name \ SOURCE und >IN
  ?dup while
    ( addr len -- i*x r:table )
    do-recognizers
    state @ if
      1+ \ compile Methode in r:*
    then
    @i execute \ r:* ist im Flash
    \ Housekeeping
    ?stack
  until
  \ parse-name hat immer addr/len
  \ len fehlt schon, da Null
  drop ;
```

Entscheidend ist, dass der Interpreter absolut generisch geworden ist. Er hat für keinen Datentyp eine Sonderbehandlung. `Do-recognizers` ist sehr systemnah gehalten. Eine komplette Beispielimplementierung findet sich im Request For Comments[3].

Die Methodentabelle

Jeder Recognizer verfügt über eine Datenstruktur mit drei Methoden. Diese Methoden nutzt der Interpreter, um die Daten bearbeiten zu lassen. Eine einfache Implementierung ist die Umsetzung als Array von drei Execution Tokens. Per Konvention heißen die Methodentabellen `r:`. Ein parsing word kann durchaus verschiedene Methodentabellen zurückgeben, ebenso kann eine Methodentabelle von verschiedenen parsing words genutzt werden. Ein Beispiel ist weiter unten. Für die Standardfälle sind die Namen der Methodentabelle auch systemübergreifend die gleichen.

Interpret

Die Interpret–Methode wird im Interpreter–Modus aufgerufen. In aller Regel wird sie nicht allzuviel machen, da die Daten auf dem Datenstack liegen und dort auch verbleiben werden. Eine Ausnahme ist natürlich das Ausführen von Worten aus dem Dictionary.

Compile

Die Compilemethode wird aufgerufen, wenn STATE ungleich 0 ist. Damit die Daten in das Dictionary geschrieben werden können, werden sie vom Datenstack gelesen und ggf. zusammen mit einigen Worten für die Runtime versehen. Execution Tokens, deren Dictionaryeintrag das Immediate–Flag aufweist, werden ausgeführt.

Postpone

Warum erfährt POSTPONE eine Sonderbehandlung? Das ist zumindest nicht offensichtlich. Bei Postpone wird die Compilation Semantics eines Wortes ins Dictionary geschrieben. Dabei unterscheiden sich „normale“ und „immediate“ Worte voneinander. Ein Postponing von Zahlen ist, bislang zumindest, nicht definiert. amforth wirft eine Exception, gforth macht das Gleiche wie beim Compilieren.

Recognizer Verwalten

Für die Verwaltung der Recognizer sind weiterhin die Worte `get-recognizers` und `set-recognizers` zuständig. Sie arbeiten in Analogie zu `get/set-order` mit einer variablen Anzahl von XT für die *parsing words* und regeln durch deren Position im Stack die Reihenfolge. Oben auf dem Stack (neben der Anzahl) ist der Wort-Recognizer, gefolgt vom Zahlen-Recognizer (für die Integerzahlen). Im Unterschied zu früher ist kein expliziter Ende-Recognizer mehr vorhanden, dessen Job hat `r:fail` übernommen.

```
' rec-int
' rec-word
2 set-recognizers
```

Für die Definition eigener Recognizer, genauer für die Definition der Methodentabelle ist das Wort `recognizer:` vorgesehen. Es erhält drei XT für die Aktionen des Interpreters und einen Namen. Das Ergebnis ist eine Methodentabelle, deren Token im *parsing word* als Teil des Ergbnisses zurückgegeben wird.

Und EVALUATE?

Natürlich sind alle Regeln des Forth Textinterpreters genauso für `EVALUATE`, `LOAD-FILE`, `INCLUDE-FILE`, `LOAD`, `THRU` usw. gültig. Genauso stellt ein per `MARKER` erstellter Snapshot auch die Recognizer wieder auf den gespeicherten Stand zurück. `FORGET` sollte das eigentlich auch können, aber wie sollte es das tun?

Unterschiede zwischen amforth und gforth

Für die Nutzer gibt es nur einen: gforth hat mehr Worte im Dictionary. Das hat ganz ohne Zweifel Vorteile bei der Stringverarbeitung, da hat amforth aber kürzlich ganz ordentlich aufgeholt. Das unterschiedliche Verhalten beim Postponing von generischen Daten wurde bereits erwähnt, ist aber, da es bislang komplett unmöglich war, eher als Experimentierfeld zu sehen.

Bei der Implementierung gehen die beiden Systeme naturgemäß getrennte Wege. In amforth sind die Methodentabellen hinter `r:*` einfach nur Arrays aus drei Elementen, gforth hat eine objektorientierte Umsetzung für diese Aufgabe.

Zusammenfassung

Die neuen Recognizer sind kein monolithischer Block mehr, sondern setzen sich aus einer Anzahl von für sich genommen kleinen Worten zusammen. Da Recognizer immer im Kontext des Textinterpreters laufen, können sie auch per `>IN` und `SOURCE` auf dessen Daten zurückgreifen. Dies macht sich der String-Recognizer zu Nutze, der das übergebene Wort nur als Ausgangspunkt für eigene, tiefgreifende Aktionen innerhalb von `SOURCE` nutzt. Damit sind Strings als *nativer* Datentyp machbar und das komplexe `S"` kann Legacystatus erhalten.

Desweiteren wird jetzt klargestellt, dass jeder Datentyp im Dictionary abgelegt und auch verzögert (postponed) compiliert werden kann. Das stellt zumindest für die Strings die Frage: Wird der neue String kopiert oder ein Verweis auf den bereits existierenden String im neuen Wort hinterlegt? Wer sich vor der Entscheidung (vorerst) vorbei mogeln will, muss eine Fehlerreaktion hervorrufen.

Wer mag, kann die neuen Recognizer im Amforth ab Version 5.3 ausprobieren, gforth bietet mit den aktuellen Entwicklungsversionen aus dem git Repository[4] ebenso Raum für eigene Experimente. Beide Systeme haben zudem eine Vielzahl von Recognizern geschaffen, deren Quellcode für jedermann zugänglich ist.

Eine formale Beschreibung des Recognizerkonzepts ist um einiges aufwändiger, als sie in Programmcode umzusetzen. Ein Ansatz hierfür ist auf der Homepage von Amforth zu finden[3]. Dort sind alle Details beschrieben, die hier zu kurz gekommen sind, wie etwa der Umstand, dass es mehr als nur den Datenstack geben kann. Kommentare sind ausdrücklich erwünscht!

Referenzen

[1] Matthias Trute, *Recognizer – Interpreter dynamisch verändern*, Vierte Dimension 2011-02

[2] Bernd Paysan, *Recognizer*, Vierte Dimension 2012-02

[3] Matthias Trute, „Forth Recognizer – Request For Comments“ `http://amforth.sourceforge.net/pr/Recognizer-rfc.pdf`

[4] gforth GIT Repository `http://git.savannah.gnu.org/cgit/gforth.git`

Beispiele

Der in [1] vorgestellte Recognizer für Gleitkommazahlen sieht jetzt so aus

```
' noop              \ interpret
:noname fliteral ;  \ compile
:noname fs. -48 throw ; \ no real postpone
recognizer: r:float

: rec:float >float if r:float else r:fail then ;
```

Zum Vergleich der alte Recognizer, noch mit counted strings

```
: rec-float ( addr -- f )
   count >float if
     state @ if postpone fliteral then
     -1
   else
      0
   then ;
```

Auch wenn Morsezeichen in letzter Zeit gerne als Aufhänger für alles genutzt wurden, sind Zeitangaben auch nützlich. Input ist ein String mit den durch Doppelpunkt getrennten Zahlen, das Ergebnis soll eine Zahl sein, die die Anzahl der Sekunden seit 0:0:0 (also Mitternacht) darstellt. Da der Tag 86400 Sekunden hat und der Recognizer auch auf einem 16–bit–System funktionieren soll, ist das Ergebnis eine doppelt genaue Zahl.

```
\ is the character a ':' ?
: ':'? ( addr len -- addr+1 len-1 f )
   over >r 1 /string r> c@ [char] : = ;

\ extract a number from the current string
: get-number ( addr len -- d addr' len' )
  0. 2swap >number ;

\ (hours*60+minutes)*60+seconds
: a+60b 2swap 60 1 m*/ d+ ;

: rec:h:m:s ( c-addr u -- d r:dnum | r:fail )
      get-number ( -- hh. addr len )
      ':'? 0= if 2drop 2drop r:fail exit then

      get-number ( -- hh. mm. addr+1 len-1 )
      2>r a+60b  2r>
      ':'? 0= if 2drop 2drop r:fail exit then

      get-number \ -- (hh*60+mm). ss. addr len
      \ len must now be 0 or its not a time stamp
      if drop 2drop 2drop r:fail exit then drop
      \ add minutes to seconds and finish
      a+60b  r:dnum ;
```

Da das *parsing word* `rec:h:m:s` keine Abhängigkeiten zum Textinterpreter hat, ist das Testen auch ohne Einbeziehung in ihn möglich. `xy` und `ab` sind Zahlen, die die `r:fail` bzw `r:dnum` Methodentabellen repräsentieren, der Wert ist uninteressant.

```
(ATmega328P)> s" 01:00:00" rec:h:m:s . d.
 xy 3600 ok
(ATmega328P)> s" 01:00:0a" rec:h:m:s .
 ab ok
(ATmega328P)> s" 72:00:09" rec:h:m:s . d.
 xy 259209 ok
(ATmega328P)>
```

Und gforth

```
mt@Ayla:~$ ./gforth time-recognizer.frt
Gforth 0.7.9_20140402, Copyright (C) 1995-2013 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type 'license'
Type 'bye' to exit
s" 01:00:00" rec:h:m:s . d. xy 3600 ok
s" 01:00:0a" rec:h:m:s . ab ok
s" 72:00:09" rec:h:m:s . d.  xy 259209 ok
```

Nach der Einbindung in den Interpreter sieht das dann so aus:

```
(ATmega328P)>' rec:h:m:s get-recognizers 1+ set-recognizers
ok
(ATmega328P)> 01:00:00 d.
 3600 ok
(ATmega328P)> 01:00:0a d.
01:00:0a ?? -13 8
(ATmega328P)> 72:00:09 d.
 259209 ok
(ATmega328P)> : test 2:00:00 1:00:00 d- d. ;
 ok
(ATmega328P)> test
 3600 ok
(ATmega328P)>
```

Für Mikrocontroller braucht man gelegentlich eine Vielzahl von Worten, die sich mit einem Subsystem beschäftigen. Damit man bei der Namensvergabe nicht den Überblick verliert, kommen Konventionen zum Einsatz, z.B. `i2c.readbyte` oder `adc.readport`. Forth bietet mit den Wordlists eine weitere elegante Methode zur Strukturierung. Dann muss man nur noch die Reihenfolge in der Search Order im Auge behalten und passend umbauen.

Mit einem Recognizer, der ein Wort in zwei Teile zerlegt und den ersten als Wordlist und den zweiten als Wort behandelt, wird das transparenter und robuster:

```
wordlist constant i2c-wl
: rec:i2c ( addr len -- xt flags r:word | r:fail )
  2dup s" i2c."
  \ etwas amforth-Magie, da Strings im Flash sind
   dup >r here imove here r>
 string-prefix? 0=
   if 2drop r:fail exit then
 4 /string dup 0=
  if 2drop r:fail exit then
 i2c-wl search-wordlist dup 0=
  if drop r:fail exit then
 r:word ;

\ Recognizer anhängen
' rec:i2c get-recognizers 1+ set-recognizers
\ current auf neue Wordlist setzen
get-current i2c-wl set-current

\ Die Worte werden ohne I2C präfix definiert
: tx ." transmitting" ;
: rd ." receiving"  ;
\ ....

\ Voriges Current wieder aktivieren.
set-current
```

Im Programmcode, der die I2C–Routinen nutzt, sieht das dann so aus

```
\ fetch a byte from hwid/addr
: c@i2c ( addr hwid -- c )
   dup i2c.begin
     swap i2c.tx
     i2c.rd i2c.tx
     i2x.rx
   i2c.end ;
```

Die Worte in der I2C–Wortliste kann man sich jederzeit mit `i2c-wl show-wordlist` ansehen. Dann haben sie natürlich kein „`i2c.`“ als Präfix. Amforth hat mit dem Wordlist Scope (`wlscope`) ein weiteres Feature in petto, das als Ergänzung zu den Recognizern gesehen werden kann. Damit kann man zur Compilezeit festlegen, in welche Wordlist ein Wort kommen soll. Default ist natürlich `current`, wenn jedoch ein Wort mit „`i2c.`“ anfängt, könnte man es auch in die i2c–Wordlist einhängen, ohne `current` anzupassen. Das ist eine Geschichte, die vielleicht später erzählt wird...

Goertzel–Algorithmus

Rafael Deliano

Eine Variante der diskreten Fourier–Transformation (DFT) [1], die besonders für DTMF–Empfänger verwendet wurde.

Der Einsatz war ursprünglich auf DSPs beschränkt, die dann aber 24–32 PCM-Kanäle gleichzeitig verarbeiten konnten. Für nur einen Kanal sind heute auch 16–Bit–Controller mit Multiplizierer schnell genug. Es ist damit zwar kein Verfahren, das sich speziell für 8–Bit–CPUs eignet, erreicht. Aber wenn man nur eine Frequenz erkennen will, ist das zumindest nicht viel aufwändiger als ein Allpol–Filter [11].

Detektor

Ausgangspunkt sei der Detektor für eine Frequenz durch konventionelle DFT, wie für DTMF–Empfänger in [10] dargestellt **(Abb.1)**. Neben der korrekten Berechnung der Zeigerlänge wäre auch die rechts dargestellte vereinfachte Variante ohne Berechnung der Wurzel in vielen Anwendungen ausreichend. Wegen der Integrate–&–Dump–Stufe arbeitet diese Implementierung nicht kontinuierlich, sondern liefert nur am Ende eines Datenblocks einen Ausgangswert. Diese Eigenschaft hat auch der Goertzel–Algorithmus **(Abb.2)**, selbst wenn seine Ähnlichkeit mit einem IIR–Filter [12] ersteinmal anderes vermuten lässt.

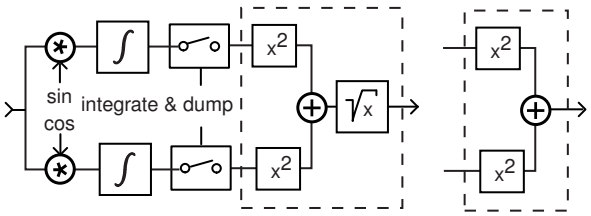


Abbildung 1: DFT Tondetektor, optional reduziert auf squared magnitude

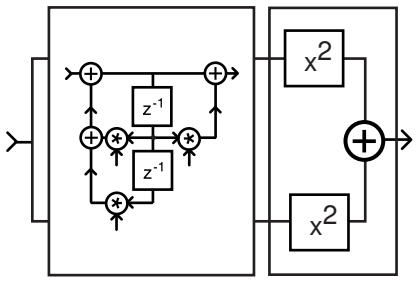


Abbildung 2: Eingangsstufe auf Goertzel geändert

Goertzel

Wegen der komplexen Rechnung müsste man für ihn eigentlich zwei Filter verwenden **(Abb.3)** . Die linke Hälfte des Filters ist aber identisch, also findet sich in der Literatur meist eine kompaktere Darstellung **(Abb.4)**. Man beachte dann aber, dass die rechte Hälfte komplexes Datenformat hat. Dieser Teil wird aber nur einmal, am Ende des Datenblocks, berechnet.

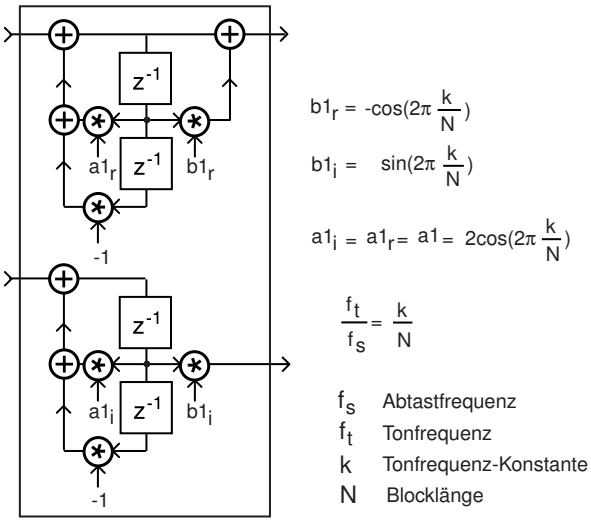


Abbildung 3: Goertzel unoptimiert

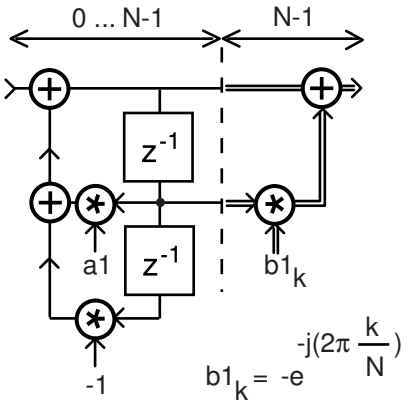


Abbildung 4: Kompaktere Darstellung, beide Ablaufphasen

Allpol

Die linke Hälfte arbeitet, wie bei einem Filter gewohnt, kontinuierlich über jedes Sample. Bei einem DTMF–Empfänger, der als A/D–Wandler ein PCM–Codec verwendet, ist die Samplerate fest fs=8kHz. Die Frequenzen der 8 DTMF–Töne sind auch vorgegeben. Variabel ist dann also nur N, die Zahl der Samples im Datenblock. Große Werte wären wegen der Schmalbandigkeit besser. Für k sind aber ganzzahlige Werte nötig. Mit geringer Frequenzabweichung für alle 8 Frequenzen. Das schränkt die Möglichkeiten ein. Der übliche Wert für fs=8kHz ist

dann N=205 **(Tab.1)** [4], obwohl auch N=105 schon vorgeschlagen wurde **(Tab.2)**. Man kann auch auf fs=4kHz reduzieren, dann empfiehlt sich N=106 [5].

Die Koeffizienten sind so gewählt, dass dieses Filter auf der Stabilitätsgrenze liegt und damit extrem schmalbandig ist. Da der Faktor „a1“ als Festkommazahl nicht exakt darstellbar ist, könnte das Signal entweder Richtung Übersteuerung driften oder abklingen. Aber die Quantisierung der Daten, die bei digitalen Filtern meist für unschöne Grenzzyklen sorgt, wirkt sich hier meist stabilisierend auf die Schwingung aus. Solange das Signal nicht innerhalb der N=205 Samples wegdriftet, ist die Funktion nicht beeinträchtigt.

Tabelle 1: Parameter DTMF–Decoder [4]

N=205 fs=8kHz

ft=	k=	a1=
697Hz	18	27906
770Hz	20	26802
852Hz	22	25597
941Hz	24	24295
1209Hz	31	19057
1336Hz	34	16529
1477Hz	38	12945
1633Hz	42	9166

Tabelle 2: Parameter DTMF–Decoder [5]

N=105 fs=8kHz

ft=	k=
697Hz	9
770Hz	10
852Hz	11
941Hz	12
1209Hz	16
1336Hz	18
1477Hz	19
1633Hz	21

Squared Magnitude

Wenn man die rechte Hälfte des Filters mit der folgenden Bestimmung der Zeigerlänge kombiniert, werden erhebliche Vereinfachungen möglich **(Abb.5)**. Nach Ausgabe des Ergebnisses müssen die beiden Speicher des Filters auf null gesetzt werden und der nächste Datenblock kann beginnen.

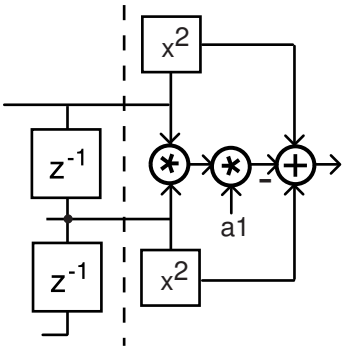


Abbildung 5: Berechnung squared magnitude

Nachverarbeitung

Da der Datenblock mit einem Rechteckfenster gewichtet wurde, ist der Frequenzgang wellig und für Oberwellen etwas durchlässig **(Abb.6)**. Für viele Anwendungen genügt deren Bedämpfung jedoch. Bei DTMF–Signalen z.B. sind kaum Oberwellen vorhanden.

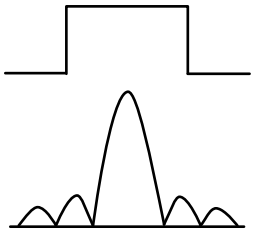


Abbildung 6: Rechteckfenster

Wenn das Eingangssignal z.B. durch ALC einen konstanten Pegel hat, genügt der Vergleich mit einer festen Schaltschwelle, um eine Frequenz zu erkennen **(Abb.7a)**. Andernfalls kann man mit breitbandigen Energiedetektor den Eingangspegel bestimmen und daraus die Schaltschwelle ableiten **(Abb.7b)**.

Bei DTMF–Empfängern auf DSPs werden manchmal pro Frequenz zwei Goertzel verwendet [3] **(Abb.7c)**. Da Sprache im Gegensatz zu DTMF reichlich Oberwellen hat, erleichtert der zusätzliche Detektor die Unterscheidung beider Signale.

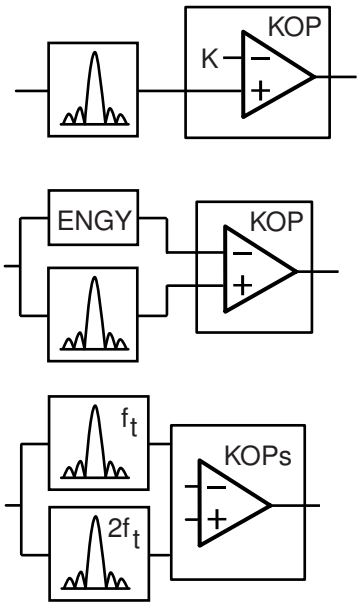
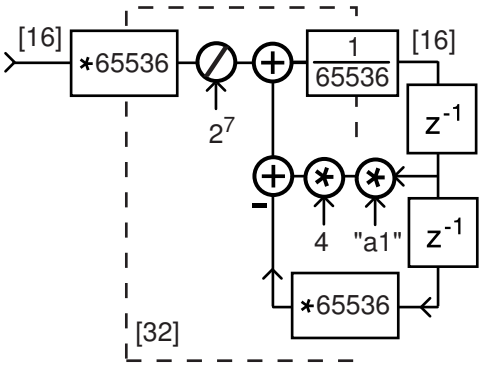


Abbildung 7: Schalter



"1,0" = 32767
a1 = 1,4712 = 2∗0,7356
"a1"= 32767 ∗ 0,7356 = 24103

Abbildung 8: Skalierung Allpol

Und nun zur Implementierung

Als Beispiel wird die gängige DTMF–Variante 941Hz mit N=205 , k=24 verwendet. fs/ft ist dann ca. 8,5 und die Konstante a1 hat den Wert 1,4712.

Wenn man diesen Allpol mit Puls anregt, wird er dauerhaft auf der Sinusfrequenz, d.h mit ca. 8,5 Samples Periode, weiterschwingen.

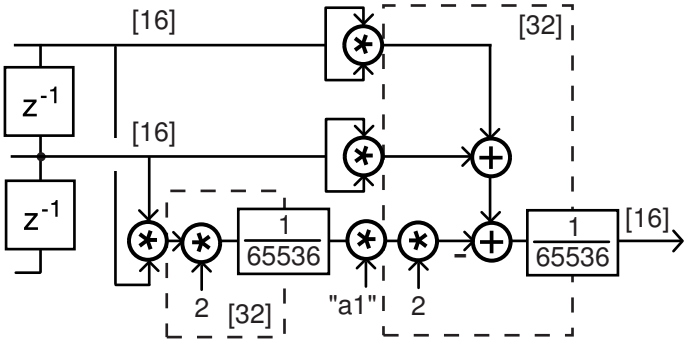


Abbildung 9: Skalierung squared magnitude

Sinusgenerator

Für weitere Tests benötigt man Sinustabellen mit „Frequenzen“ in passendem Bereich **(Abb.10)**. Gerade Längen mit 7 ... 20 Samples enthalten eine einzelne Sinuskurve. Eine Tabelle mit 17 Samples, die zwei Sinusperioden enthält, kann krumme Werte, wie 8,5 , darstellen. Die Tabellen werden hier mit ±7FFF vollausgesteuert verwendet. Die 16–Bit–Dynamik im Allpol–Filter reicht dann nur recht knapp aus. Es empfiehlt sich, wie in [12], ein 32–Bit–Akkumulator, der auf 40 Bit erweitert ist und damit Überlauf erkennt und Sättigung durchführen kann.

Nach Tests wurde der Vorteiler am Eingang **(Abb.8)** auf 2^7 ausgelegt. Dämpft man noch mehr, wird intern ungenügend ausgesteuert. Während mit 2^6 die Overflows bereits Nebenmaxima verschlechtern **(Abb.10)**. Zwar wird der Überlauf durch die Sättigungslogik entschärft, aber eine leichte Verschlechterung des Messwerts tritt trotzdem ein, sodass er nicht unbegrenzt tolerierbar ist. Man sollte also in realen Anwendungen anhand anliegendem Pegel und Frequenzgemisch untersuchen, wie sich intern die Aussteuerung des Allpols verhält. Die abschließende Berechnung der Zeigerlängen **(Abb.9)** ist zwar etwas umfänglich, aber technisch unproblematisch.

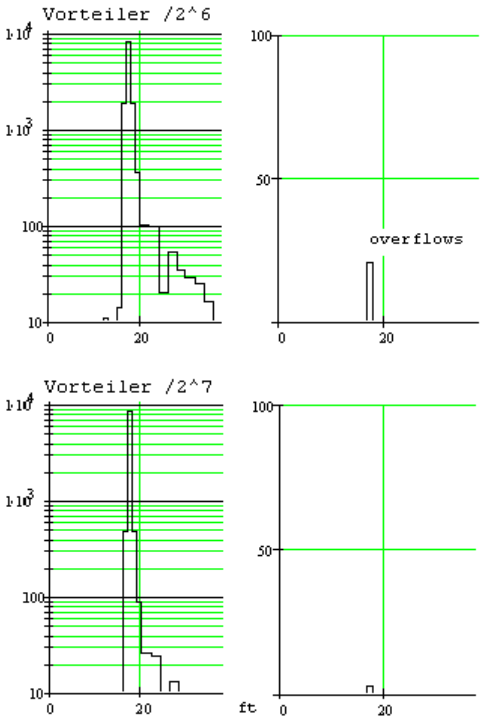


Abbildung 10: Frequenzgang & Overflows

Timing

Das Programm belegt in Assembler auf einem 2,45–MHz–MC68HC908 ca. 400 Bytes FLASH. Der Allpol benötigt 220usec pro Sample, die folgende Berechnung der Zeigerlänge 350usec. Wenn der Allpol z.B. in einem 250usec–Timerinterrupt berechnet wird **(Abb.11)**, wird man dort auch die abschließende Berechnung der Ausgabewerte vornehmen. Die Unterbrechung von 1... 2 Samples ist bezogen auf die Blocklänge N=205 unerheblich.

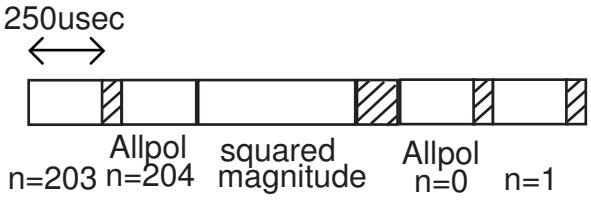


Abbildung 11: Timing

Guard–Bits

Für Simulation und Debugging des Algorithmus ist oft eine portable Variante in Hochsprache wünschenswert. Es ist in Assembler einfach, an das 32–Bit–Datenwort 8 Bit anzuhängen, um einen überlaufsicheren 40–Bit–Akkumulator zu erhalten. In Hochsprache teilt man die Summierung in eine 32–Bit–Addition, bei der das oberste Bit gelöscht ist **(Abb.12)**, sodass man Carry detektieren kann. Gefolgt von zwei Additionen für die oberen 16 Bit und das Carrybit.

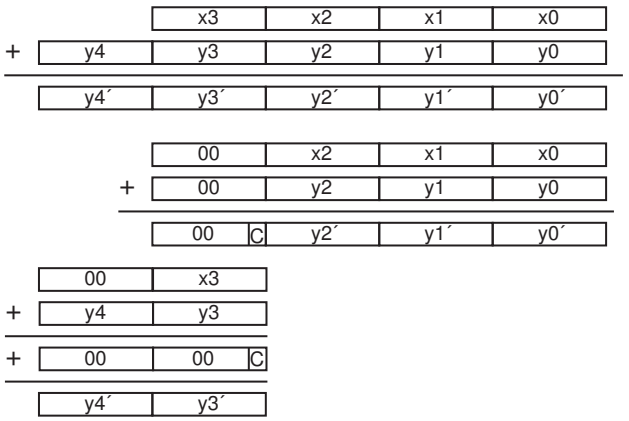


Abbildung 12: Akku mit Guard–Bits in HLL

Referenzen

[1] Goertzel „An Algorithm for the Evaluation of Finite Trigonometric Series“ Amer. Math. Monthly Jan. 1958

[2] Gerdsen, Kröger „Digitale Signalverarbeitung in der Nachrichtentechnik“ Springer 1993

[3] Mock „Add DTMF generation and decoding to DSP–uP designs“ EDN March 1985

[4] Chen "Modified Goertzel Algorithm in DTMF Detection Using the TMS320C80" Application Note SPRA066 1996

[5] Gay, Hartung, Smith „Algorithms for Multi–Channel DTMF Detection for the WE DSP32 Family“ ICASSP 89

[10] emb (12) DTMF–Decoder

[11] emb (13) 2pol Allpol Filter

[12] emb (14) Biquads

[14] (ist im text erwähnt, nicht angegeben ⟶ nachfragen)

Listing

```
<| \ Goertzel.txt

HEX

\ include:
\ D+       \ ( D1 D2 --- D3 ) D1 + D2 = D3
\ D-       \ ( D1 D2 --- D3 ) D1 - D2 = D3
\ D2/      \ ( D1  --- D2 ) D2 = D1 / 2  signed
\ D2*      \ ( D1  --- D2 ) D2 = D1 * 2
\ 2/       \ ( N1 --- N2 ) N2 = N1 / 2  signed
\ 2*       \ ( N1 --- N2 ) N2 = N1 * 2
\ N->D     \ ( N1 --- D1 ) expand 16 Bit to 32 Bit
\ NEGATE   \ ( N1 ---  N2 ) N2 = N1 * -1
\ DNEGATE  \ ( D1 --- D2 ) D2 = D1 * -1
\ ABS      \ ( N1 --- N2 )
\ DABS     \ ( D1 --- D2 )
\ UD>      \ ( UD1 UD2  --- Flag ) flag = 1 if UD1 > UD2
\ 0<       \ ( UN1 --- Flag )  flag = 1 if UN1 < 0

D% 4 ZVARIABLE zRAM      \ storage allpol
D% 24103 CONSTANT A1"    \ filter constant

\ ------------------------------------
```

```

2 ZVARIABLE ERR-CNTR \ overflow counter

: ERR-CNTR+      \ (  ---  ) increment counter
1 ERR-CNTR +! ;

5 ZVARIABLE AKKU
\ 0+        extension-byte to 40 bit
\ 1+ HW HB  register 32 bit
\ 2+    LB
\ 3+ LW HB
\ 4+    LB

5 ZVARIABLE AKKU" \ scratchpad

: AKKU+     \ ( UD1  ---   ) Akku = Akku + UD1
00  AKKU" C!
2DUP AKKU" 1+ D! 00FF AND
     AKKU  1+ D@ 00FF AND D+  \ add lower 3 bytes
2DUP AKKU 2+ C! AKKU 3 + !
SWAP DROP 0100 AND 8SHIFT>  \ extract carry-bit
AKKU @ AKKU" @ + + AKKU !  \ add top 2 bytes plus carry
;

: AKKU-     \ ( UD1  ---   ) Akku = Akku - UD1
```

```
00 AKKU" C! AKKU" 1+ D!
 AKKU  1+ D@ 00FF AND
 AKKU" 1+ D@ 00FF AND D-    \ subtract lower 3 bytes
2DUP AKKU 2+ C! AKKU 3 + !
SWAP DROP 0100 AND 8SHIFT>  \ extract carry-bit
AKKU @ AKKU" @ -
SWAP - AKKU !  \ sub top 2 bytes, sub carry
;

: SAT-AKKU   \ (  ---  ) saturate upper half
AKKU C@ 80 AND
IF
  AKKU @ NOT FF80 AND   \ if not B% 1111 1111 1xxx
  IF 8000 AKKU 1+ !     \ saturate to xx 8000 xxxx
     ERR-CNTR+
  THEN
ELSE
  AKKU @ FF80 AND     \ if not B% 0000 0000 0xxx
  IF 7FFF AKKU 1+ !   \ saturate to xx 7FFF xxxx
     ERR-CNTR+
  THEN
THEN
;

: AKKU@      \ (  --- N1 ) upper half
AKKU 1+ @ ;

: AKKU!      \ (  D1 --- ) clear extension-byte
DUP 8000 AND IF FF ELSE 00 THEN
AKKU C!      \ extend to 40 Bit
AKKU 1+ D! ;

\ ---------------------------------------

2 ZVARIABLE SIGN

:  ABS     \ ( N1 --- UN1 ) patch
DUP 8000 = IF DROP 8001 THEN
ABS ;

: (MAC)    \ ( N1 UN1 --- UD1 ) setup SIGN for AKKU+
OVER 8000 AND SIGN !  \ flag = 1 if result negative
SWAP ABS U* ;

: AKKU+    \ ( UD1 ---  )
SIGN @ IF AKKU- ELSE AKKU+ THEN ;

: MAC      \ ( N1 A1" ---  )
(MAC) AKKU+ ;

: *2-MAC   \ ( N1 A1" ---  )
(MAC) 2DUP AKKU+ AKKU+ ;

: *4-MAC   \ ( N1 A1" ---  )
(MAC) 2DUP 2DUP 2DUP AKKU+ AKKU+ AKKU+ AKKU+ ;

\ ---------------------------------------------

: -1*-MAC  \ ( N1  ---  )
DUP NEGATE 8000 AND SIGN !   \  flag inverted
ABS 0 SWAP AKKU+ ;

\ ---------------------------------------------
: ALP         \ ( N1 ---  ) allpole filter
AKKU 5 00 FILL
0 SWAP        \    --- D1 )
D2/ D2/ D2/ D2/ D2/ D2/
AKKU!
zRAM    @ A1"  *4-MAC
zRAM 2+ @     -1*-MAC
SAT-AKKU
AKKU@
zRAM    @ ZRAM 2+ !
          ZRAM    ! ;

: SM       \ (   --- N1 ) calculate signed magnitude
AKKU 5 00 FILL  0 SIGN !
zRAM    @ ABS DUP DUP U* AKKU+   \  --- |z-1| )
zRAM 2+ @ ABS DUP DUP U* AKKU+   \  --- |z-1| |z-2| )
U* D2* SWAP DROP                 \  --- N2 )
A1" U* D2*
zRAM @ zRAM 2+ @ XOR NOT 8000 AND SIGN ! \ inverted sign
AKKU+
SAT-AKKU
AKKU@ ;

: INIT-ALP    \ (  ---  ) clear allpole filter
0 zRAM ! 0 zRAM 2+ ! ;

|>
```

Der Forth–2012–Standard

Bernd Paysan

2004 regte Anton Ertl an, einen neuen Standard zu entwickeln (unter dem ambitionierten informellen Namen Forth200x), da der alte nun schon 10 Jahre auf dem Buckel hat, und es üblicherweise alle 10 Jahre einen neuen Sprachstandard gibt — so lange dauert das eben. 2014 ist dieses Standard–Dokument fertig geworden, es heißt „Forth–2012." In diesem Artikel sollen die ganzen Veränderungen kurz beschrieben werden.

Warum „Forth–2012", wenn er erst 2014 fertig geworden ist? Weil 2012 so eine Art Feature–Freeze eingesetzt hat, also keine Aufnahme substantieller neuer RfDs, sondern nur noch Kleinigkeiten geändert wurden. Auch das dauert, denn auch wenn das Standard–Komitee auf inzwischen gar nicht mehr so moderne Kommunikationsmittel wie Usenet und E–Mail gesetzt hat, und sogar gegen Ende VoIP und Online–Abstimmungen benutzt hat, sind persönliche Treffen eben wichtig, aber rar. Einmal im Jahr eben, bei der EuroForth.

Dieser Standard gehört keinem Standard–Body, aber die Vorgehensweise ist an den vorherigen ANSI–Standard angelehnt gewesen. Es gibt also Regeln für stimmberechtigte Mitglieder des Komitees, Regeln, wann eine Abstimmung als „Konsens" angesehen wird (ein Standard kodifiziert gemeinsame Praxis, also ist ein Standard ein Konsens–Werk), und Regeln, wie im Vorfeld mit Vorschlägen umgegangen wird. Wir haben kein Design by Committee, sondern die RfDs, die üblicherweise von einem oder wenigen Personen ausgearbeitet werden, gemeinsam diskutiert und bei denen über einen Voting–Prozess CfV ein Stimmungsbild erhoben wird — bei diesem CfV ist jeder stimmberechtigt, bei der Frage, ob das in den Standard kommt, und wenn ja, genau so oder leicht geändert, entscheidet das Komitee. Der Chair, Anton Ertl, hat kein Sonder–Votum.

Der Standard baut auf dem dpANS94–Dokument auf, allerdings hat sich der Editor, Peter Knaggs, gegen Word und für LaTeX entschieden. Was hat sich nun in den letzten 10 Jahren am Standard geändert?

Änderungen zu Forth–94

Alle als „veraltend" markierten Wörter bis auf FORGET wurden aus dem Standard gestrichen, also #TIB, CONVERT, EXPECT, QUERY, SPAN und TIB. Und WORD, dessen veraltete Semantik das Anhängen eines Leerzeichens betrifft (von der ebenfalls veralteten Funktion CONVERT benutzt), hat diese Funktion verloren.

Der separate Floating–Point–Stack, die für portable Programme nützlichere Variante, ist nun Standard. Standard–Programme müssen also nicht mehr für die Option gemeinsamer Stack auch codiert werden (das hat ohnehin niemand gemacht). Die Variante mit gemeinsamem Stack ist als „environmental restriction" noch zugelassen, ist aber aus der Mode geraten.

In Forth–94 wurden Environment–Queries eingeführt, insbesondere für Wordsets, die kaum jemand benutzt hat. Diese gibt es zwar noch, die Queries auf Wordsets sind aber als veraltend markiert. Zudem wurden [COMPILE] und LOCALS| neu auf die Liste der veraltenden Wörter gesetzt, zu FORGET. Mit [DEFINED] und [UNDEFINED] ist die üblichere Variante des Nachfragens, ob ein Wort vorhanden ist, eingezogen.

TO hat neue Ziele bekommen, nämlich Wörter, die von FVALUE und 2VALUE erzeugt werden. Ein System sollte sein TO also für verschiedene Ziele einrichten.

Die Fehlerrückgaben *iors* in Forth–94 waren so gemeint, dass man sie mit THROW werfen kann. Das war aber nur eine Empfehlung. Jetzt ist es verpflichtend vorgeschrieben. Die Anzahl der lokalen Variablen hat sich auf 16 verdoppelt.

Was viele Systeme schon lange nutzen, Number–Prefixe, ist jetzt auch Standard: # für Dezimal, $ für hex und % für binär. Zeichen können als '*c*' geschrieben werden.

FASINH und FATAN2 wurden enger spezifiziert.

Zusätzliche Wörter

Core word sets

Die Einführung deferred words in den Standard hat DEFER, IS, DEFER@, DEFER! und ACTION–OF hinzugefügt. Dann gibt es noch PARSE–NAME, S\", BUFFER: und HOLDS.

Double Number word sets

2VALUE ermöglicht Values auch für diesen Datentyp

Facility word sets

Hier sind zwei Gruppen eingeführt worden: Strukturen und Konstanten zur standardisierten Behandlung von EKEY. Die Strukturen bringen die Wörter BEGIN–STRUCTURE, END–STRUCTURE, +FIELD, CFIELD: und FIELD: mit, die Auswertung von EKEY–Ereignissen ein EKEY>FKEY, K–F1 bis K–F12, K–HOME, K–INSERT, K–LEFT, K–NEXT, K–PRIOR, K–RIGHT, K–UP, K–DOWN, K–DELETE, K–END, sowie die Masken K–ALT–MASK, K–CTRL–MASK und K–SHIFT–MASK.

File–Access word sets

Hier wurden INCLUDE, REQUIRE und REQUIRED hinzugefügt.

Floating–Point word sets

Die Fließkommazahlen haben drei Felder für die Strukturen bekommen, DFFIELD:, SFFIELD: und FFIELD:, ein erzeugendes Wort FVALUE, einen zusätzlichen Rundungsmodus FTRUNC (round to zero), und die auch schon weit verbreiteten Konversionsfunktionen in Single F>S und S>F, da auf 64–Bit–Plattformen die Doubles weniger Sinn machen als früher.

Locals word sets

Eigentlich sollte die geschweifte Klammer als Ersatz für LOCALS| aufgenommen werden, aufgrund Konflikte mit ein paar Systemen ist jetzt {: daraus geworden. In den vielen Systemen, die { als Locals–Block schon länger verwendet haben, wird das sicher auch weiterhin benutzbar bleiben.

Programming–Tools word sets

Hier hat mit TRAVERSE–WORDLIST eine Higher–Order–Funktion Einzug in Forth gehalten, ebenso wie ein neuer Typ, „nt“, das Name Token. Den kann man mit NAME>STRING, NAME>COMPILE und NAME>INTERPRET in xts oder den String umwandeln. N>R und NR> ermöglichen es, n Werte auf dem Stack (abgezählt im TOS) auf den Return–Stack zu schieben, und von dort wieder zurückzubekommen, etwa für GET–ORDER und SET–ORDER. SYNONYM erlaubt einen Alias, und [DEFINED]/[UNDEFINED] das Prüfen, ob ein Wort existiert.

String word sets

Hier ist mit SUBSTITUTE und den Hilfswörtern REPLACES und UNESCAPE ein Vorschlag aus dem Internationalisierungs–Projekt aufgenommen worden: Damit kann man String–Templates mit Makros auffüllen. Als beiderseitigen Delimiter gibt es das von DOS inspirierte '%'.

Extended–Character word sets

Dieses ist komplett neu, es ist die größte Änderung am Standard, die es erlaubt, mit heute üblichen Zeichensätzen variabler Länge (insbesondere UTF–8 hat weite Verbreitung gefunden) umzugehen. Forth hat hier den Vorteil, lang genug gewartet zu haben, die Programmiersprachen, die sich schon Anfang der 90er begeistert auf Unicode gestürzt haben, haben da eine frühe, und unreife Version davon bekommen, von der sie jetzt nicht so leicht wegkommen. UTF–8 und vergleichbare Codierungen sind ASCII–kompatibel, und haben einen sehr viel kleineren Reibungsverlust.

An Wörtern gibt es X–SIZE, XC!+, XC!+?, XC, XC–SIZE, XC@+, XCHAR+, XEMIT, XKEY, XKEY?, +X/STRING und –TRAILING–GARBAGE im Extended–Character word set selbst, und CHAR (als Erweiterung füx xchars), EKEY>XCHAR, PARSE, X–WIDTH, XC–WIDTH, XCHAR–, XHOLD, X\STRING– und [CHAR] (ebenfalls für xchars) in den extended words.

Ausblick

Der Standard selbst wird weiterhin als rollendes Dokument gepflegt und die Arbeit fortgesetzt, diese Ausgabe ist ein Snapshot, auf den sich die Forth–Gemeinde geeinigt hat. Im Embedded–Bereich kleiner Systeme bleiben ein paar Probleme, die die Akzeptanz des Standards erschweren, also etwa CREATE .. DOES> und die Header–Flags, die in Flash–Systemen je nach Beschaffenheit auch Probleme machen können. Insbesondere im inzwischen populär werdenden ARM–Cortex–Bereich machen einige merkwürdige Gebilde den Programmierern das Leben schwer. Insofern werden diese Systeme wohl absichtlich immer die eine oder andere Warze behalten, die sie vom Standard unterscheidet.

Das Komitee hat sich natürlich Arbeit aufgehalst. Es gibt insgesamt 15 Punkte, von denen wir jetzt schon planen, sie umzusetzen:

1. Der Ansatz, über eine Library als Referenz–Implementierung Funktionalität zu standardisieren
2. Einen Vorschlagsprozess für solche Libraries auszuarbeiten
3. Die Bedeutung von Compilation/Interpretation und dem xt zu präzisieren
4. Am Input–Format für Fließkommazahlen zu drehen
5. Ein Wordset für Sockets (oder eine Library)
6. Quotations
7. Zweierkomplement in den Standard verankern, nachdem alles andere ausgestorben ist
8. Ein Zeichen als eine Adress–Unit festschreiben, so wie das fast überall verwendet wird
9. Recognizers einführen (ja, die hier im Heft)
10. Multitasking und –threading sowie message passing und wie sich Nebenläufigkeit inzwischen darstellt
11. Internationalisierung, bzw. das, was als dritter Teil übrig geblieben ist: Das Managen von übersetzten Texten und Meldungen
12. Cross Compiler, ein Ansatz, der schon vor 15 Jahren mal angefangen wurde, aber inzwischen wieder veraltet ist, und erneut erarbeitet werden muss
13. IEEE floating point library, das ist ein sehr umfangreiches Projekt
14. Directories/filenames, nachdem der Widerstand der Menschen aus Backslashistan inzwischen gebröckelt ist
15. Das Memory–Access–Wordset, das einfach auch nach der Überarbeitung immer noch zu groß war

Es soll an neuer Technologie auch noch ein Wiki geben, in dem man den Standard nachlesen und bearbeiten kann.

Interrupts, Scheduler, Multitasking

Karsten Roederer

Statt eines Erfahrungsberichtes der Bromfiets-Zündung in VD 4/2013 möchte ich gerne auf das Thema Multitasking im Embedded Bereich eingehen. Die Zündung wird im Bromfiets nicht zum Einsatz kommen, da mir die Schneckenwelle schon mit einer einfacheren Zündveränderung gebrochen ist. Das Getriebe würde eine leistungsfähigere Zündanlage nicht verkraften.

Beim Grübeln über ein sinnvolles Beispiel, was zeigt, wo ein Multitasker wünschenswert wäre, ist mir just die Software für die Zündverstellung eingefallen[1]

Warum Überhaupt?

Im Embedded Bereich, gerade wenn es sich um kleinere Aufgaben handelt, erschließt sich die Notwendigkeit eines Multitaskers erst einmal nicht. Bei der Zündverstellung haben wir 2 wirklich unabhängige Haupttasks, für die man einen Multitasker verwenden könnte:

1. Zeitgerechte Zündung
2. Handverstellung

Dazu kommt noch forthtypisch die Interpreter/Kompiler-Umgebung der Terminalschnittstelle, die gerne mit `key? ... (UNTIL)` aus der Mainloop heraus aktiviert wird.

Da bei der Auswahl der geeigneten Hardware jene den Zuschlag erhält, die das oder die Echtzeitproblem(e) schon in Hardware löst, hat man sich von der didaktisch sinnvollen Vorgehensweise — eins nach dem anderen — schon gelöst und steht eigentlich vor dem Problem, einen Multitasker zu benötigen.

Interrupt

Richtig performant wird eine Applikation, wenn man die Interrupts, die die eingebaute Hardware generieren kann, auch nutzt. Das fällt den meisten Anwendern schwer, da man ja das „eins nach dem anderen“ quasi schon genetisch in die eigene Vererbung eingebaut hat und man einfacher halber den Schritt, den man gerade programmiert, gerne mit `BEGIN ...` flag `UNTIL` „zu Ende führt“, bevor der Softwaredenkapparat sich dem *nächsten* Schritt widmet (aus eigener Erfahrung).

Aber zunächst einmal eine kurze Erklärung, was ein Interrupt ist: Egal, was der Prozessor gerade macht, wird bei Auslösung eines Interrupts (z.B. extern durch Drücken eines Klingelknopfes, dessen Signal auf einen Interrupt–Pin des Chips geführt ist) eine spezielle Adresse angesprungen. D.h., es wird das Programm ausgeführt, welches an dieser Adresse beginnt. Meine gerade zuvor ausgeführte Applikation befindet sich dann in einem sehr gefährdeten Zustand. Der Interrupt–Programmierer muss jetzt, je nach zu programmierender Hardware, ev. auf Dinge achten, die nichts mit seinem Anliegen zu tun haben. So muss z.B. gern mal das Statusregister gerettet werden und nach erfolgter Interrupt–Routine zurückgeschrieben werden, damit die ursprüngliche Applikation wieder an der Stelle weitermachen kann, wo sie unterbrochen wurde. Ein Schludern bei der Programmentwicklung wird sicher keinen volkswirtschaftlichen Schaden anrichten, da diese Fehler zugleich offenbar werden. Ich sag mal „Blue Screen“ gleich nach dem Einschalten oder bei Auslösung des Interrupts.

Schlimmer sind da schon nicht dokumentierte Ungereimtheiten der Hardware. Ich denke da gerade an den Novix NC4000, der bei Ausführung eines 2–Clock–Cycle–Befehls den Interrupt verwarf oder an den RTX2000, der bei einem höher priorisierten externen Interrupt den gleichzeitigen internen Timer–Interrupt vergaß. Das sind dann so Überraschungen, die einen erst in einer späten Designphase ereilen und ein ganzes Projekt umstürzen können, wenn die Auswahl der Hardware das Funktionieren dieses Features voraussetzt. Beim Novix reichte es, die obersten Datenbits per Hardware zu dekodieren und dann den Interrupt so lange zu latchen, wie es Klaus Schleisiek seinerzeit veröffentlicht hat. Beim RTX2000 war es für mich dramatischer, da dieser Fehler erst einen Tag vor Fahrtantritt mit einem Forschungsschiff entdeckt wurde. Zum Glück hatte ein A/D–Wandler noch einen eingebauten Timer, sodass die geophysikalischen Laufzeitmessungen am Meeresboden mit dem vorhandenen Design ohne Hardwareänderungen durchgeführt werden konnten. Die Software musste aber *schnellstmöglich* grundlegend geändert werden — endlich eine Herausforderung für Ulrich Hoffmann, dem die Pflege (Neuprogrammierung) meines ursprünglichen Spaghetti–Codes oblag. Die asynchronen Interrupts scheint man in den neuen Prozessorgenerationen mittlerweile in den Griff bekommen zu haben. Zumindest bei den 18er PICs. In einem geclocktem Design ist es halt nicht so einfach asynchrone Flankenwechsel sicher zu händeln.

Der Scheduler

Der Scheduler ist das zentrale Element für das Multitasking. Er treibt sein Unwesen in einem Multitaskingsystem unscheinbar im Hintergrund. Entweder als Timesharer oder präemptiv (könnte man so übersetzen: April April ich bearbeite jetzt aber nicht Deine Task), wie ihn z.B. Arndt Klingelnberg für ein Forth-System beschrieben hat [2] oder als kooperativer, welcher durch anarchistische Programmierkunst ausgehebelt werden kann.

Das präemptive Multitasking hat sich deshalb zu Zeiten, an denen mehrere User sich einen Prozessor teilten, als einzig gangbarer Weg durchgesetzt. Nur frag ich mich: Ist es heute noch nötig den User, der mittlerweile Herr über mindestens 2 Cores ist, noch so zu gängeln mit einem Betriebssystem, das jeglicher sicheren Vorhersagbarkeit entbehrt? Denn das ist der entscheidende Nachteil des präemptiven Multitaskings: Ich kann nicht vorhersagen oder berechnen, wann meine Task spätestens zur Ausführung kommt im laufenden Betrieb. Somit ist ein solches Betriebssystem für sicherheitsrelevante Anwendung ein NOGO, worauf mich ein guter Safetymann/frau auch hoffentlich hinweisen wird, falls ich so etwas ins Auge gefasst hätte.

Das kooperative Multitasking kann nur funktionieren, wenn der Programmierer seine Applikation sorgfältig ausführt. D.h.: keine „`BEGIN ... AGAIN`“–Konstrukte ohne irgendeine zeitliche Begrenzung, kein „`BEGIN ... flag UNTIL`“, ohne zeitliche Begrenzung, kein r> drop — jetzt mal im Scherz — sowieso.

Ein „handprogrammierter Scheduler“ nach folgendem Muster:

```
BEGIN
flag1 @ IF task1 THEN
flag2 @ IF task2 THEN
...
key? UNTIL
```

ist sehr übersichtlich und der Safetymann/frau muss nicht in den Tiefen des Betriebssystems rumwühlen, um die Machart des Schedulers zu ergründen. Im Artikel von Klaus Schleisiek [3] ist ein kooperativer Multitasker beschrieben, der quasi invers zu einer in der sonstigen Prozessorwelt vorzufindenden Art funktioniert. Sonst hab ich keine bessere Veröffentlichung in den 4d's gefunden bezüglich der Implementation eines kooperativen Multitaskers.

Der Interrupt–Scheduler

Der Interrupt-Scheduler wird da benötigt, wo die verschiedenen Interruptquellen auf dieselbe Adresse springen. Er arbeitet in gleicher Weise, wie oben beschrieben. Nur wird man, ob der Eile, die man beim Interruptservice an den Tag legt, ihn gleich in Assembler ausführen. Damit ist er schneller und die Performance des Prozessors wird nicht so geschmälert. Das sieht dann für den PIC–FORTH–Assembler z.B. ein bisschen kryptisch so aus:

```
: irqserv ( -- ) \ interrupt service routine
  [i
  ccp1ifm pir1 mtst \ is it ccp1if?
  if
    [ pir1 2 a, bcf, ] \ clear ccp1if
    [ ccp1con 0 a, bsf, ]
    \ make him high hopefully
  then
  ccp2ifm 1 pir2 mtst \ 17° Trigger
  if [ pir2 0 a, bcf, ] \ clear ccp2if
    [ portb 4 a, bsf, ] \ set for test
    RB4 ccpr2h c@ 8 lshift ccpr2l c@ +
    dup dup \ messe Periode
    capt3 @ dup rot >
    if swap invert 1+ + period !
    else - period ! then \ war Überlauf
    capt3 ! \ store new capture
    period @ dup 360 / dup onedegree !
    degrees @ * - \ new igntcomptime
    capt3 @ + dup $ff00 and 8 rshift
    ccpr1h c! \ new compare high
    $ff and ccpr1l c! \ new compare low
    [ portb 4 a, bcf, ] -1 igcountdown !
    \ clear for foreground routine
  then
  adifm pir1 mtst \ is it the ADC-Interrupt?
  if
    adresh c@ 8 lshift adresl c@ +
    channel @ sampel ! \ store sample
    adifm pir1 mclr channel @ 1+ dup channel !
    \ increment A/D-Channel
    dup 1 > if drop 0 dup channel ! -1 allad !
    then \ only channel 8 & 9
    $8 + 2 lshift \ here only channel 8 & 9
    adcon0 c@ $c1 and or adcon0 c!
    [ adcon0 1 a, bsf, ] \ go
  then
  i]
;i
```

Das folgende Bild möge veranschaulichen, was mit dem Timer angestellt wird.

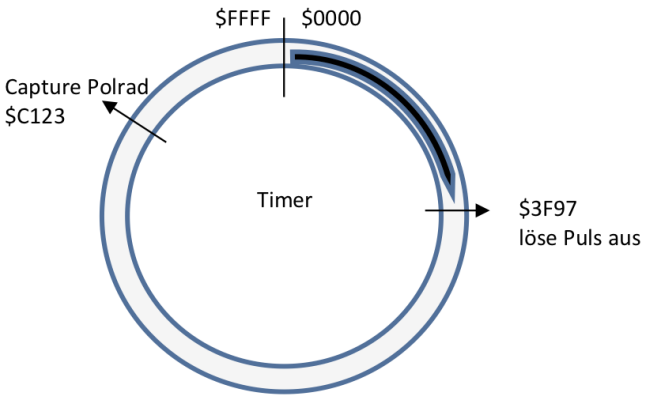


Abbildung 1: Entnommen: [1]

Es gibt also hier gleich 3 Timer–Interruptquellen:

1. 1. Der 17° vor OT–Interrupt (oberer Totpunkt des Kolbens) — extern
2. 2. Timerinterrupt für die Zündung
3. 3. Interrupt für die Zündpulslänge

Für den A/D–Wandler wird die Flag–Variable „allad“ verwendet, die ankündigt, wenn ein neuer Set von frisch gewandelten A/D–Daten da ist. Sie wird in der Vordergrundroutine „resettet“. Die Interrupt–Routine ist gleich

für einen Set an Kanälen ausgelegt, wobei in dieser Applikation bisher nur der eine Kanal benötigt wird. Der angeschlossene Temp–Sensor wird noch nicht verarbeitet. Für den Timer wurde die Flag–Variable „igcountdown“ spendiert, die aber in der Vordergrundroutine nicht verwendet wird.

Der Vordergrund

Der Vordergrund sieht jetzt sehr entspannt aus. Es können hier jetzt noch weitere Programmteile eingebaut werden, ohne dass die „Echtzeit-Performance“ leidet. In der Regel Outputs visueller Art.

```
: start ( -- )
  begin
    allad @
    if decode 0 allad ! then
    \ so wait for new ADC-inputs
  key? until ;
```

Für den MSP430 wird in der Regel kein Scheduler in der obigen Art (die Interruptroutine ist gemeint) benötigt, da fast jede Interruptquelle eine eigene Adresse anspringt und somit die Frage: „Wer war das?“ meistens entfällt. Aber in beiden Fällen, wie im PIC wie auch im MSP430, führen die Programme über die Worst-Case-Analyse zu berechenbaren späteste Ausführungszeiten (und das Stirnrunzeln bei Safety verschwindet).

Multitasking?

Bei der Frage, ob man einen im Betriebssystem eingebetteten Multitasker verwenden sollte, möge man mir die folgende Abschweifung verzeihen: Bei Airbus ist Safety ein von Airbus bezahlter Personenkreis, ohne den kein Airbus mehr fliegen darf. Selbst jedes je verkaufte Flugzeug von Airbus darf von Stund an nicht mehr fliegen, wenn Airbus, von der Pleite überrascht, seine Safety–Leute nicht mehr bezahlt. Somit sind der Airliner in der Pflicht dafür zu sorgen, dass Airbus solvent bleibt. Gibt es denn eine bessere Lizenz zum Gelddrucken? Boeing war auch mal in dieser Situation. Nur haben die es sich verscherzt durch Safety–Seilschaften, die enttarnt, dazu führten, dass Safety da jetzt eine Behörde ist mit Uniform und allem Drum und Dran. D.h., wenn Boeing insolvent würde, darf jeder ausgelieferte Jet weiterfliegen. Das macht die Boeing Jets wertvoller, egal, wie schrottig das Design sonst auch sein mag. Das nur mal am Rande. Um Safety bei der Fehlerbaumanalyse nicht unnötig zu verwirren, plädiere ich dafür, einen ev. eingebetteten Multitasker nicht zu verwenden und lieber einen Scheduler in der oben beschriebenen Art zu verwenden. Gerade wenn man auch selber die Implementation nicht so genau nachvollziehen (sprich Safety erklären) kann. Der Scheduler hat auch noch den Vorteil, dass er schneller ist als ein eingebetteter Multitasker.

Referenzen

[1] K. Roederer, *„Zündung mit Handverstellung,“* Vierte Dimension, 2013/4.

[2] A. Klingelnberg, *„Forth von Triangel,“* Vierte Dimension, pp. 4-10, 1993/4.

[3] K. Schleisiek, *„µCore anwenden,“* Vierte Dimension, 2005/2.

ARM–Cortex und die Steckplatine

Matthias Koch

Wer schon immer einmal zum Kennenlernen mit einem kleinen ARM–Cortex auf Lochraster oder Steckplatine basteln wollte, für den sei dieser Artikel gedacht — der auch eine „Vorspeise“ zum großen ARM–Cortex–Sonderheft darstellt, welches um Weihnachten erscheinen wird.

Ganz zu Beginn: Keine Angst!

Steckplatine und Drähte sind bestimmt vorhanden. Die restliche Zutatenliste ist kurz:

1 LPC1114FN28 mit achtundzwanzig lochrasterfreundlichen Beinchen (gibt's bei `http://www.watterott.com/`)
1 Kondensator 100 nF
2 Taster (Schließer)
1 USB–Seriell–Adapterkabel mit 3,3 V Pegeln plus
1 sechspolige Stiftleiste, um es anzuschließen.

Und zur Stromversorgung etwas zwischen 3,0 V und 3,6 V. Wenn das USB–Seriell–Kabel wie jenes von FTDI eine 5V–Versorgungsspannung anbietet, genügen:

2 Si–Dioden, z.B. 1N4148
1 Leuchtdiode, gerne grün
1 Widerstand 100 Ohm

Sowie für die hier beschriebenen Experimente noch

1 RGB–Leuchtdiode mit gemeinsamer Kathode.

All dies wird so zusammengebaut, siehe Schaltplan Abbildung 1 und Steckbrett Abbildung 2.

Die beiden Dioden sorgen für je 0.7V Spannungsabfall und so bleiben von den 5 V aus dem USB–Anschluss 3,6 V übrig. Die Leuchtdiode und ihr Vorwiderstand sind wirklich wichtig, weil sonst die Spannung ohne Last langsam ansteigen könnte. Wer mag und hat, kann natürlich stattdessen einen Spannungsregler verwenden. Wenn während des Loslassens des

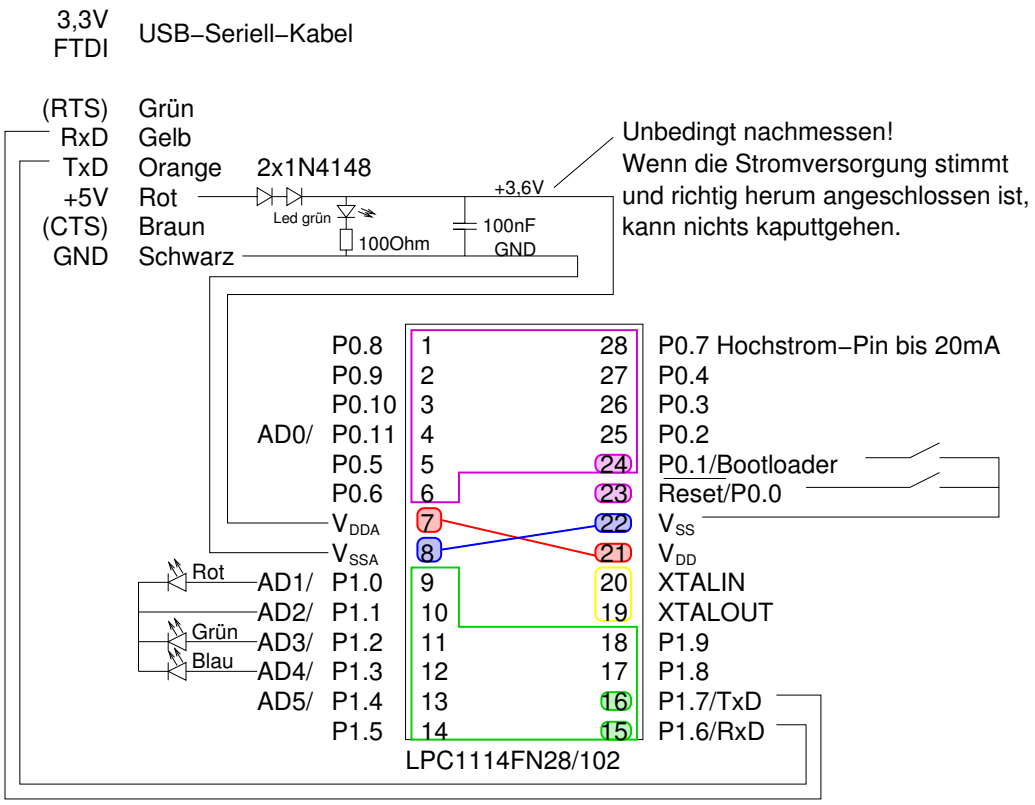


Abbildung 1: Schaltplan

Reset–Tasters der Bootloader–Taster gedrückt ist, springt der Chip in den Bootloader–Modus und erwartet unsere Wünsche. Jetzt kann bereits Mecrisp–Stellaris geflasht werden. Dafür gibt es unter Linux LPC21ISP (`http://sourceforge.net/projects/lpc21isp/`) welches so aufgerufen wird: `lpc21isp mecrisp-stellaris-lpc1114fn28.hex /dev/ttyUSB0 9600 12000` Wer Windows benutzt, wird mit Flash Magic (`http://www.flashmagictool.com/`) fündig. Anschließend ist ein Terminal mit 115200 Baud 8N1 zu öffnen und Mecrisp–Stellaris begrüßt uns nach einem weiteren Druck des Reset–Tasters.

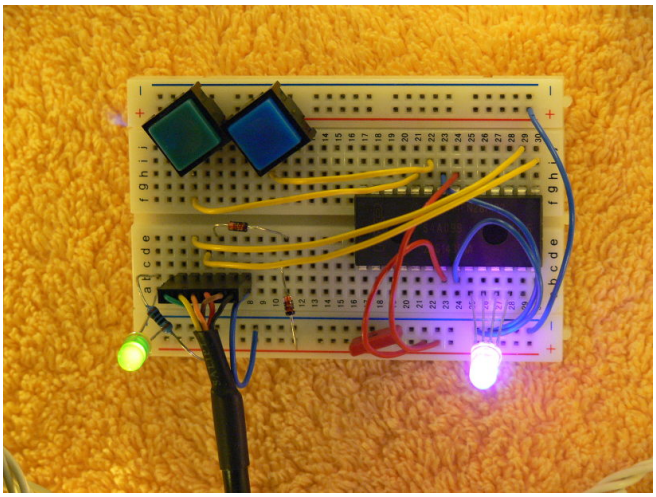


Abbildung 2: Steckbrett

Auf Entdeckungsreise

Ein Teil soll noch zur Zutatenliste hinzugefügt werden: Eine RGB–LED mit gemeinsamer Kathode, die einfach so neben den Chip in die Steckplatine eingefügt wird, dass die Anode für Rot an P1.0, die gemeinsame Kathode an P1.1, Grün an P1.2 und Blau an P1.3 angeschlossen ist. Wer mag, kann natürlich auch andere Leuchtdioden je nach Wunsch wählen. Vorwiderstände sind nicht nötig, da die Pins des Microcontrollers jeweils 4 mA liefern — mit Ausnahme von P0.7, wo 20 mA fließen. Beide Taster können nach Programmstart umkonfiguriert werden. Während der Bootloader–Taster nach seiner Abfrage beim Start frei ist, ist es für die ersten Experimente jedoch empfehlenswert, den Reset–Taster in seiner Funktion beizubehalten.

Jetzt geht es an die Vorbereitung der Leitungen — dafür sind die IOCON–Register zuständig, die für jeden Pin festlegen, welche Funktion er erfüllen soll. Ein Blick ins Datenblatt für die Pinbelegung und ins Referenzhandbuch, das „User manual“, Kapitel 8, ist an dieser Stelle sehr nützlich, weil es für unterschiedliche Pins viele Varianten gibt. Doch für jene Leser, die gleich loslegen wollen, seien hier die passenden Werte für die ersten Experimente fertig herausgesucht:

```
$D0 $40044010 ! ( P0.1 als digitalen IO-Pin mit )
( internem Pullup für den Taster konfigurieren )

$C1 $40044078 ! ( P1.0 als digitalen IO-Pin ohne )
( Sonderfunktionen konfigurieren )
$C1 $4004407C ! ( P1.1 ... )
```

```
$C1 $40044080 ! ( P1.2 ... )
$C1 $40044090 ! ( P1.3 ... )
```

Jetzt müssen noch die Leuchtdioden–Pins als Ausgänge geschaltet werden, was im Referenzhandbuch in Kapitel 12 beschrieben wird. Da diese Register oft verwendet werden, geben wir ihnen hier der Bequemlichkeit halber Namen:

```
$50003FFC constant P0DATA
$50008000 constant P0DIR

$50013FFC constant P1DATA
$50018000 constant P1DIR
```

Der Taster soll Eingang sein, was mit 0 P0DIR ! erreicht werden kann, und wir machen die Leuchtdiodenanschlüsse alle auf einmal mit `%1111 P1DIR !` zu Ausgängen. Anschließend können die einzelnen Farben in P1DATA zum Leuchten gebracht werden: `%0001 P1DATA !` ergibt Rot, `%0100` lässt Grün aufleuchten und `%1000` lässt blaues Licht scheinen.

Um den Taster zu lesen, ist eine kleine Definition nützlich `: taster? ( -- ? ) 2 P0DATA bit@ ;` mit der sogleich ein Farbwechsel programmiert werden kann

```
: rotgrün ( -- ) begin taster?
    if %0001 P1DATA ! else %0100 P1DATA ! then
  key? until ;
```

Ein kleiner Trick

Aufmerksame Leser des Ledcomm–Artikels (VD 2/2013) wissen bereits, dass Leuchtdioden auch als Photodioden nützlich sind und sich so für die Kommunikation nutzen lassen. Diesmal jedoch soll es einfacher zugehen und wir verwenden die RGB–Led als drei kleine farbempfindliche Solarzellen, die mit dem Analog–Digital–Wandler ausgelesen werden sollen.

Für den Analog–Digital–Wandler, der in Kapitel 25 des Referenzhandbuches beschrieben ist, müssen wir zunächst einmal den Strom einschalten und den Takt aktivieren, was für die GPIOs und die serielle Schnittstelle ja schon beim Start von Mecrisp–Stellaris erledigt wird. Anschließend bekommt die Kathode einen Low–Ausgang und alle Anoden werden zu analogen Eingängen. In einer Schleife können nun die Spannungen an den Anoden nacheinander gemessen und ausgegeben werden.

```
$40048080 constant SYSAHBCLKCTRL
  \ Clock control register
$40048238 constant PDRUNCFG
  \ Power-down configuration register

$40044078 constant IOCON_PIO1_0
$4004407C constant IOCON_PIO1_1
$40044080 constant IOCON_PIO1_2
$40044090 constant IOCON_PIO1_3

$50013FFC constant P1DATA
$50018000 constant P1DIR

$4001C000 constant AD0CR
  \  AD-Wandler Control Register
$4001C004 constant AD0GDR
  \            Global Data Register


: farbsensor-init ( -- )
       \ Vorbereitungen für den Farbsensor
  1 13 lshift SYSAHBCLKCTRL bis!
       \ Takt für AD-Wandler einschalten
  1  4 lshift PDRUNCFG bic!
       \ Stromaus-Bit des AD-Wandlers löschen

  $C1 IOCON_PIO1_1 !
       \ P1.1 sei digitaler IO-Pin
       \ ohne Sonderfunktionen,
  2 P1DIR !      \     Ausgang
  0 P1DATA !     \           low

  $42 IOCON_PIO1_0 ! \ P1.0 sei analoger Eingang
  $42 IOCON_PIO1_2 ! \ P1.2 ...
  $42 IOCON_PIO1_3 ! \ P1.3 ...
;

: analog ( Kanal -- Messwert )
\ Einen analogen Eingang wandeln
  1 swap lshift \ Kanal wählen
  4  8 lshift or
  \ CLKDIV = 4 --> 12 Mhz / 4
  \        = 3 MHz < 4.5 MHz Maximum.
  1 24 lshift or  \ Messung starten
  AD0CR !

  begin 1 31 lshift AD0GDR bit@ until
      \ Auf das Wandlungsende warten
  AD0GDR @ 6 rshift $3FF and
      \ Messergebnis passend schieben
;

: farbsensor ( -- )
  farbsensor-init
  begin
    1 analog ." Rot: " u.
    3 analog ." Grün: " u.
    4 analog ." Blau: " u.
    cr
  key? until
;
```

Die gemessenen Werte sind relativ zur Versorgungsspannung, die in diesem Beispiel 3,6 V beträgt, und lassen sich leicht umrechnen: $Spannung = Messwert * 3,6\,V/1024$. Die Umrechnung der Verhältnisse der Messwerte untereinander in Farben muss für die verwendete RGB–Leuchtdiode experimentell ermittelt werden. Der Gedanke dabei ist, dass eine Leuchtdiode für ihre eigene abgestrahlte Farbe am empfindlichsten ist und sich manchmal auch von „blauerem“ Licht, also mit größerer Photonenenergie, anregen lässt, jedoch nicht von „roterem“ Licht, also mit geringerer Photonenenergie. Das heißt, dass rotes Licht nur die rote LED anregt, grünes Licht die grüne LED und ein bisschen auch die rote LED, während blaues Licht alle drei Leuchtdioden anregen kann.

Vintage Computer Festival Berlin 2014

Carsten Strotmann

Am Wochenende 4. und 5. Oktober fand zum ersten Mal das Vintage Computer Festival in Berlin statt, nach 15 Jahren in München (als Vintage Computer Festival Europe) gibt es nun einen Ableger in der Bundeshauptstadt.

Organisiert von Prof. Stefan Höltgen und dem Team des AFRA (ein Akronym für „Abteilung für Redundanz-Abteilung“ :)) lockte das VCFB trotz des sonnigen Herbstwetters über tausend Besucher in die Räume des Pergamon–Palais.

Die Austellung war auf drei Räume aufgeteilt: in dem Hauptraum gab es allerlei historische Rechner zu sehen. Ein Raum war für die Geschichte der Apple-Rechner reserviert, es gab vom Apple II bis zum neusten MacBook die komplette Ahnengalerie von Apple-Geräten zu besichtigen (inkl. Lisa, dem Ur–Macintosh und dem Newton). In einem dritten Raum kamen Spielernaturen auf ihre Kosten, im Spieleraum waren Arcade–Spielautomaten und Homecomputer mit bekannten Spiele–Klassikern zum Losspielen aufgebaut.

Abbildung 1: Apple–Halle

In einem Vortragsraum gab es das ganze Wochenende interessante Vorträge, von denen viele auch im CCC–Media–Archiv (`http://media.ccc.de/browse/conferences/vcfb/2014/`) noch nachträglich angeschaut werden können. Im Rechnerarchiv der FU–Berlin wurden ganztägige Workshops angeboten, z.B. eine Einführung in die Spiele–Programmierung unter Assembler auf einem 6502–Homecomputer.

Wie auch schon auf den vergangenden VCFe–Veranstaltungen in München, gab es auf der Berliner Ausgabe den Forth–Benchmarkwettbewerb. Ziel dieses Wettbewerbes war es wieder, die Teilnehmer zu animieren, aktiv an den alten Rechnern zu arbeiten. Es gab eine Reihe an vorgefertigten, simplen Forth–Benchmark–Programmen (für FIG–Forth, Forth–83 und ANSI–Forth), welche schnell eingetippt werden können. Insgesamt gab es 18 neue Benchmark–Ergebnisse (die Benchmarkergebnisse aus Berlin, und alle Ergebnisse vorheriger Wetttbewerbe können auf der Webseite `http://theultimatebenchmark.org` gefunden werden).

Abbildung 2: Ein sehr früher „Laptop“, auch schon mit Forth

Besonders spannend war ein Benchmarktest auf einem mit einer 13Mhz–ULTRAWARP–Karte aufgerüsteten Apple II. Unter aktiver Mithilfe des Entwicklers der Karte (Michael Mengel) konnten zwei Forth–Systeme (GraForth und Apple II Forth) aufgetrieben werden, welche sowohl auf dem Standard–Apple, wie auch auf dem per Turbo–Karte aufgerüsteten Rechner liefen. Eines der Forth–Systeme war leider sehr „non–standard“, die Entwickler des Forth–Systems hatten versucht, das Forth sehr „Basic“–artig zu gestalten, was das Übertragen der Benchmark–Programme ohne Dokumentation des Forth–Systems zu einer Herausforderung machte. Belohnt wurde die Mühe bei der Preisverlosung unter den Teilnehmern des Benchmark–Wettbewerbs mit dem ersten Platz für Michael Mengel, der sich als Gewinn das Forth–Buch „Die Programmiersprache Forth“ von Albert Nijhof (Übersetzung von Fred Behringer, `http://mv-buchhandel.de/wissenschaft/naturwissenschaften/672/die-programmiersprache-forth`) aussuchte.

Den zweiten Platz bei der Verlosung hat Robert Clausecker gewonnen, einen Plüsch–Swap–Drachen. Robert hat mit einem MSP430–Board und 4€4th am Wettbewerb teilgenommen. Beide Preise wurden von der Forth–Gesellschaft gestiftet.

Abbildung 3: Swappie auf der VCFB

Eine weitere sehr interessante Forth–Geschichte brachte Forth–Gesellschafts–Mitglied Helfried Schürer zum VCFB. In der 80er Jahren hatte Helfried ein Fig–Forth–System (FOSY) für die U808D–CPU (ein Klon der Intel–8008–CPU) auf einen DDR–Industriecomputer, den K–1510, portiert. Nach der Wende wurde dieser Rechner nicht mehr benötigt und geriet in die Vergessenheit. Heute gibt es warscheinlich nur noch sehr wenige lauffähige Geräte, davon eines in einem Museum in Halle/Saale.

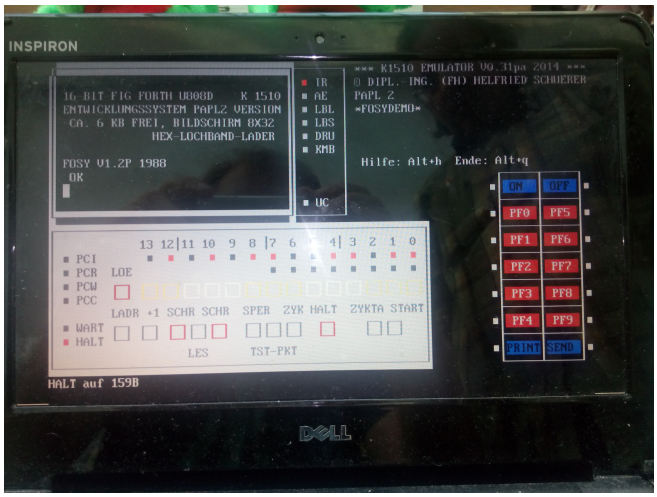


Abbildung 4: Fig–Forth auf dem U808D

Helfried hat in den vergangenen Jahren mit F–PC (unter MS–DOS innerhalb des DOS–Emulators „DOSBOX“) einen Emulator für den K180 geschrieben, komplett mit Darstellung des Bedienpanels. Und auf diesem Emulator wurde das in den 80er Jahren portierte Forth wieder zum Leben erweckt. Helfried hat die Wettbewerbs–Benchmarks auf dem emulierten K180 nachgereicht, diese sind nun auf der Benchmark–Webseite zu finden. Der K180–Emulator findet sich unter `http://fosy808.de/h/aktuelles-de.html`.

Aufgrund der guten Resonanz und Besucherzahlen wird es im kommenden Jahr sicher eine weitere Auflage des VCF in Berlin geben.

Referenzen

[1] *VCFB Homepage,* `http://www.vcfb.de/2014/`

EuroForth 2014

Bernd Paysan

Die EuroForth 2014 fand zur Abwechslung im Hotel Amic Horizonte in Palma de Mallorca statt. Mallorca ist recht günstig erreichbar, und im Prinzip sind die Hotels auch nicht teuer... allerdings war dann die Schwierigkeit, dass es dort nur ganz wenige Hotels mit Konferenzräumen gibt.

Das Ergebnis war dann das Hotel Amic Horizonte, das einen geräumigen Konferenzraum hatte, einen schönen Blick auf den Hafen, natürlich auch einen Pool und die für Mallorca üblichen Gäste. Wer in den Pausen mit T–Shirts und Shorts herumlief, war definitiv schon overdressed. Wie immer für die Konferenzen, die weder in England noch Deutschland stattfinden, waren weniger Gäste als üblich da.

Forth200x—Standard–Meeting

Wie üblich, findet die zwei Tage vor der Konferenz das Standard–Meeting statt, welches den letzten Feinschliff am Forth–2012–Standard [1] durchführte, mehr zu diesem Standard in einem eigenständigen Artikel in diesem Heft.

EuroForth–2014–Konferenz

Ab Freitag Mittag ging's dann mit der eigentlichen Konferenz und den Vorträgen und Workshops los. Proceedings [2] und Videos [3] der Vorträge (bis auf die ersten zwei, bei denen die Videokamera noch nicht im Raum war) wie immer auf den üblichen Adressen im Internet.

Freitag, 26.9.

14:00	Anton Ertl „Region-based Memory Allocation in Forth“
14:45	Stephen Pelc „Compiling to Flash“
16:00	Paul Bennet „High Integrity System — CODE“
16:40	Bernd Paysan „net2o Command Language“
17:20	Andrew Read „Concept and implementation of an extended return stack to enhance subroutine and excepton handling in Forth“

Samstag, 27.9.

9:00	Bill Stoddard „HiTEX — LATEX gets a helping hand from Forth“
9:30	Ulrich Hoffmann „Saturating Arithmetic“
10:00	Peter Knaggs „Report from the Forth 200x standard committee“

Sonntag, 28.9.

9:00	Stephen Pelc „VFX Forth for ARM Linux“
9:30	Gerald Wodni „Forth — the next generation“
10:00	Peter Knaggs „Java Forth“
10:30	Klaus Schleisiek „Doing C style structs on cell addressed uCore“
11:30	Anton Ertl „Getting rid of C“
12:00	Paul Bennet „Forth in education“

Abbildung 1: Teilnehmer mit Blick auf den Hafen

Abbildung 2: Teilnehmer vor dem Pool

Referenzen

[1] Forth 200x Standards Committee #10, *Forth 2012 document,* `http://www.forth200x.org/documents/forth-2012.pdf`

[2] *EuroForth 2014 Proceedings* `http://www.complang.tuwien.ac.at/anton/euroforth/ef14/papers/`

[3] *EuroForth 2014 Videos* `http://www.forth-ev.de/wiki/doku.php/events:euroforth-2014`

Forth–Gruppen regional

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

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

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

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

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

Gruppengründungen, Kontakte

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

μP–Controller Verleih

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

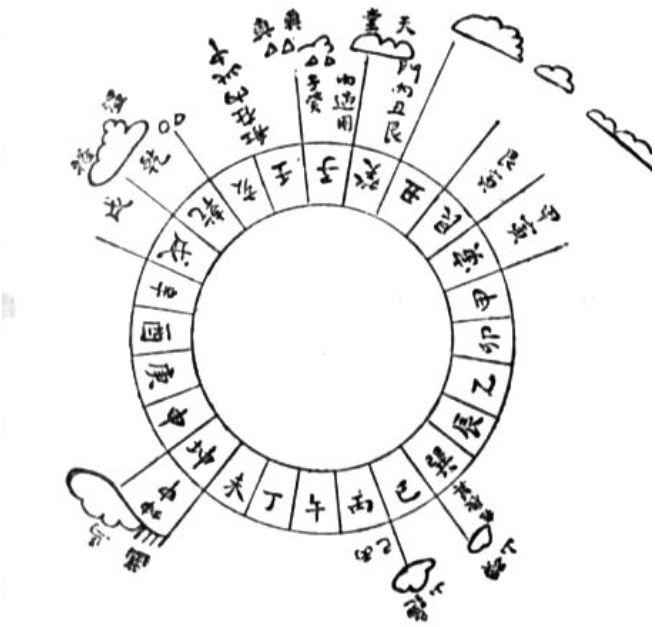
Spezielle Fachgebiete

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

Termine

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

8.–9. November 2014:
OpenRheinRuhr http://openrheinruhr.de
27.–30. Dezember 2014:
31C3 Chaos Communication Congress Hamburg
http://ccc.de
9.–12. April 2015:
Forth–Tagung 2015 in Hannover
Details zu den Terminen unter http://forth-ev.de



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

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

Einladung zur

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

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

Anreise

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

Anmeldung

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

Programm

Donnerstag

ab 13:00	Frühankommer(–Workshops)

Freitag

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

Samstag

vormittags	Vorträge und Workshops
nachmittags	Exkursion

Sonntag

09:00	Mitgliederversammlung
13:00	Ende der Tagung