Vierte Dimension

1/2013

Das Forth–Magazin

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

In dieser Ausgabe:

Dienstleistungen und Produkte fördernder Mitglieder des Vereins

tematik GmbH
Technische Informatik

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

Gegründet 1985 als Partnerinstitut der FH–Wedel beschäftigten wir uns in den letzten Jahren vorwiegend mit Industrieelektronik und Präzisionsmeßtechnik und bauen z. Z. eine eigene Produktpalette auf.

Know–How Schwerpunkte liegen in den Bereichen Industriewaagen SWA & SWW, Differential–Dosierwaagen, DMS–Messverstärker, 68000 und 68HC11 Prozessoren, Sigma–Delta A/D. Wir programmieren in Pascal, C und Forth auf SwiftX86k und seit kurzem mit Holon11 und MPE IRTC für Amtel AVR.

LEGO RCX–Verleih

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

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

Inhalt:

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

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

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

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

KIMA Echtzeitsysteme GmbH

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

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

FORTecH Software GmbH
Entwicklungsbüro Dr.–Ing. Egmont Woitzel

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

Ingenieurbüro
Klaus Kohl–Schöpe

Tel.: 07044/908789
Buchenweg 11
D–71299 Wimsheim

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

Titelbild: Funkmodul RFM12 Sende-/Empfangsmodul, Quelle: `www.pollin.de`

Impressum

Name der Zeitschrift

Vierte Dimension

Herausgeberin

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

Redaktion & Layout

Bernd Paysan, 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,

hier in Hamburg klirrt noch der Frost, aber die Sonnenstrahlen versuchen schon vehement, den Winter zu vertreiben. Ich begrüße Euch zu einer weiteren Ausgabe unseres Forth–Magazins, und wenn man auf der Titelseite nachsieht, dann stellt man fest, dass wir nun mittlerweile im 29. Jahr Neuigkeiten und Interessantes rund um Forth präsentieren.

`SWAP` hat es im vergangenen Jahr geschafft, sich im täglichen Sprachgebrauch zu etablieren. Die ökonomisch angespannte Situation in vielen Ländern hat dazu geführt, dass viele (junge) Frauen unter dem Motto *SWAP till you DROP* so genannte SWAP–Parties veranstalten, bei denen gebrauchte aber neuwertige Kleidungsstücke getauscht werden. Das stillt die Kauflust auf sparsame Weise und beruhigt die Sorge um das Verbleiben der ausgedienten Kleidung: Eine Win–Win–Situation.

`SWAP` überall (Quelle `http://www.expatica.com/upload`)

Was hat dieses Heft nun zu bieten? Den Anfang macht Michael Kalus, der uns nahebringt, wie man mit dem MSP430 Schrittmotoren ansprechen kann. Auch beleuchtet er im Artikel über *Do–It–Yourself*–Assemblieren, wie Forth–Worte Maschinencode generieren können.

Teil 13 (!) der Forth–Abenteuer von Erich Wälde, hier mit Martin Bitter als Co–autor, führt uns in die Welt der Funk–Kommunikation ein. Die beiden beschreiben, wie man Funk–Sensoren einsetzen kann, um auch an unwegsamen Stellen zu messen und die erhobenen Daten dann zentral zu sammeln.

Auch Fred Behringer ist mit zwei Artikeln vertreten. Er erinnert uns zum einen an das Wort `LOCAL`, um lokale, temporäre Änderungen an Variablen zu signalisieren. Zum anderen haucht er den LEDs an PC–Tastaturen neues Leben ein, so denn der PC noch unter DOS läuft.

Matthias Trute adaptiert den 1–Wire–Bus auf Amforth und diskutiert Unterschiede zwischen MSP430– und ATmega–Prozessoren.

Mecrisp, eine weitere Forth–Implementierung für den MSP430, wird von Matthias Koch, dem Mecrisp–Erfinder, beschrieben. Interessant ist insbesondere, welche speziellen Eigenschaften Mecrisp auszeichnen.

Und zu guter Letzt — Johannes Teich verbessert nach der Devise *einfacher ist besser* seine Wave Engine.

Ich wünsche allen Lesern viel Freude beim Lesen, gute Anregungen und neue Impulse.

Hamburg im März 2013,

Ulrich Hoffmann

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
`http://fossil.forth-ev.de/vd-2013-01`

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

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

Neuer Termin für den wöchentlichen IRC chat

Die *Stammkunden* des IRC chats haben beschlossen, den Termin von Mittwoch auf **Donnerstag 20 h** zu verschieben. Zugang zum IRC channel `#fort-ev` bekommt man über die Server des `de-IRC.net`, z.B. `irc.fu-berlin.de`. Weitere Informationen befinden sich auf den Webseiten [1] und [2].

1. http://www.forth-ev.de/article.php/20080120222945626
2. `http://irc.fu-berlin.de/ircmap.html`

ew

Neue vereinsinterne Mailingliste

Auf der Jahrestagung 2012 in Biezenmortel wurde angeregt und befürwortet, eine geschlossene, vereinsinterne Mailingliste zu schaffen, auf der alle Mitglieder der Forth–Gesellschaft e.V. eingetragen sind. Die Idee ist, dass über diese Liste die Mitglieder öfter über Aktivitäten des Vereins informiert werden können, als nur durch das Erscheinen der Vierten–Dimension. Es wird vermutlich eher wenig Verkehr auf der Liste sein, es sei denn, die Mitglieder lassen sich zum Mitmachen animieren — was natürlich gerne gesehen wird. Wer dennoch nicht auf der Liste sein möchte, kann die Mitgliedschaft jederzeit über das Mailman–Webinterface kündigen, wer eine neue E–Mail–Adresse hat, kann sich dort ummelden. Und wer keine oder eine nicht mehr genutzte E–Mail–Adresse angegeben hat, meldet sich beim Forth–Büro.

Die Liste ist unter `mitglieder@forth-ev.de` für registrierte Mitglieder erreichbar.

Dank geht an Bernd, der die widerspenstige Mailman–Software gezähmt hat.

ew

Forth auf den Chemnitzer Linuxtagen

Die Linuxtage in Chemnitz sind wohl eine der am besten organisierten Konferenzen rund um freie Software. Interessante Vorträge und viele interessierte Besucher. Daher ist es kein Wunder, dass die Forth-Gesellschaft auch in diesem Jahr am Wochenende des 16. und 17. März bei arktischen Temperaturen (-10 C°) wieder den Weg in die Technische Universität Chemnitz gefunden hat.

Am Stand wurde über neue Entwicklungen in der Forth-Welt berichtet: neues amForth, das 4€4th auf dem TI-Launchpad, gForth auf Android und im Chrome-Browser. Am Stand haben Erich Wälde, Gido Baumann, Sabine Hornig und Carsten Strotmann viele Fragen zum Thema Forth beantwortet. Die Forth-Workshops der letzten Jahre in Chemnitz haben Wirkung gezeigt, es musste weniger erklärt werden „was dieses Forth denn ist“, stattdessen wurde bei Gesprächen nun öfters in die Tiefe gegangen.

Anstatt eines Workshops hat Carsten einen Vortrag zum Thema *Forth überall* gehalten, um das weite Spektrum der Forth-Anwendungsgebiete dem Publikum vorzustellen.

Aus der Präsenz auf den Linuxtagen in Chemnitz haben sich einige neue Kontakte entwickelt, die nun in der Zukunft vertieft werden können. So fragt eine Gruppe aus München einen Informationsvortrag zum Thema *Forth programmieren* an. Wir sagen: machen wir doch gerne!

Chemnitzer LinuxTage 2013
`http://chemnitzer.linux-tage.de/2013`

cs

noForth 1210

Seit Oktober 2012 gibt es für MSP430 das *noForth* von Albert Nijhof und Willem Ouwerkerk. Es ist ein 16–bit–Forth und in zwei Versionen erhältlich. Verson 1210 für den MSP430G2553 im Launchpad, und Version 1210R für MSP430FR5739 im FRAM–Starterkit von TI.

Ihr noForth haben sie von Grund auf neu geschrieben. Es lässt 8kB Flash im LaunchPad für Anwendungen übrig, und rund 100 bytes RAM. Ist also etwas genügsamer als das *4e4th* oder *Camelforth*. Fehlende Worte aus dem Standard–Core–Word–Set gibt es als Quelle dazu. Dafür hat es einen Decompiler, was ich für eine nützliche Sache halte.

Eingebaut sind zudem zwei `MARKER` und `SHIELD` genannte forget–Funktionen. Sie funktionieren, ohne dass Speicherplatz im flash vergeudet wird. Es hat ein schnelles `FIND`, weil acht threads verwendet werden. *INSIDE*–Worte haben einen eigenen Pfad. *' APPLICATION TO APP FREEZE* erzeugt ein turnkey–system aus der Anwendung. Die Anwendung kann mittels Schalter S2 des LaunchPads umgangen werden. Der Prompt kann reguliert werden, um zusätzliche Informationen anzuzeigen. `VALUE` mit *TO* +*TO* und `INCR` ist auch dabei. Die Fehler– und Ausnahmebehandlung ist um `CATCH` und `THROW` aufgebaut. Und es benutzt floored division. Praktisch ist, dass nach einem Fehler beim Compilieren der Rest der Quelldatei ignoriert wird.

Dazu gibt es einen Assembler und Disassembler, die als Source–Code nachgeladen werden können, um direkt auf dem LaunchPad assemblieren zu können. Außerdem gibt es den *aux430ass*, einen Assembler, der in Win32forth läuft und ein Skript assembliert, das vom noForth direkt compiliert werden kann. Und es gibt schon eine ganze Reihe von Beispiel–Programmen sowie eine ordentliche Dokumentation der Speicherbelegung.

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

mk

Schrittmotor ansteuern

M.Kalus

Man besorge sich einen kleinen Schrittmotor, z.B. aus einem alten Drucker. Ich hatte einen Fuji Stepper Motor SM25 2005-A herumzuliegen, ähnlich dem im Bild **Abb. 1** . Der hatte 6 Anschlussdrähte - rot, blau, weiß, gelb, grün, grün. Die Achse kann man bei diesem kleinen Motörchen von Hand leicht drehen, sie springt dabei von einer Position zur nächsten - 20 Positionen für eine volle Umdrehung.

Dieser Beitrag ist entnommen aus dem Forthwiki [1]. Dort werden grundlegende Experimente zu MCUs gezeigt.

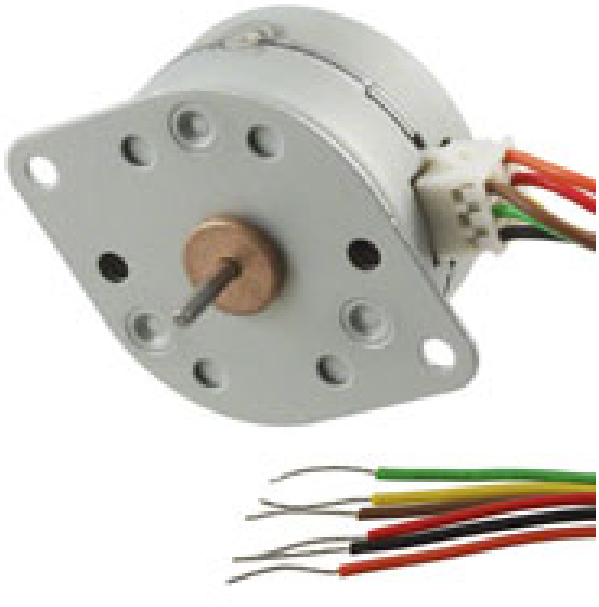


Abbildung 1: 20M Portescap Standard Unipolar Schrittmotor, 5V DC, 20 Schritte, 18° Schrittwinkel

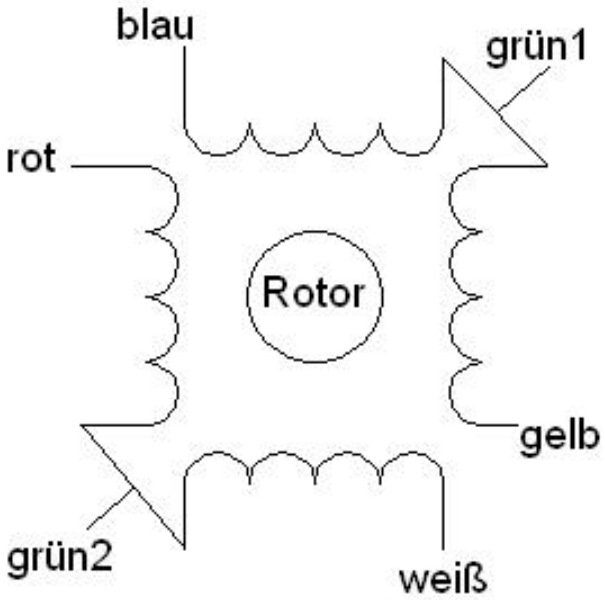


Abbildung 2: Spulenschema

Motortyp identifizieren

Mit dem Ohmmeter testet man erstmal die Verbindungen der Drähte. Dabei fand ich folgende Werte und das Schema:

[htdp]

Tabelle 1: default

	rot	blau	weiß	gelb	grün1	grün2	
rot		OL	94Ω	OL	OL	45Ω	A
blau			OL	93Ω	46Ω	OL	B
weiß				OL	OL	47Ω	C
gelb					45Ω	OL	D
grün1						OL	
grün2					OL		

(OL = keine Verbindung)

Es fällt auf, dass 4x fast die gleichen Werte, und 2x davon das Doppelte zu messen war. Das sieht ganz nach zwei Spulen mit Mittelanzapfung aus **Abb. 2**.

Ein Test an 3V ergab keine Zuckungen des Motors, egal wie man kombinierte. Mit einer 9V–Block–Batterie hingegen tat sich dann schon was! Man verbinde grün2 fest mit dem Pluspol, und nehme einen Draht in die Hand, der vom Minuspol kommt (Krokoklemmen z. B.). Und tippe dann damit abwechselnd mal an rot und dann an weiß, dann zuckt der Motor deutlich. Mit etwas Geschick findet man heraus, dass er dann mal einen Schritt vorwärts, und dann wieder einen rückwärts macht. Desgleichen mit grün1 und den Anschlüssen blau/gelb. Es sieht also so aus, als könne man durch kurzes Einschalten einer Spule den Motor um einen Schritt weiterstellen. Nun müsste man ausprobieren, welche Abfolge zu einer Drehung in eine Richtung führt. Auch das ist manuell machbar. Man schließt dazu die grünen Drähte zusammen an den Pluspol der Batterie, und tippt dann einfach der Reihe nach von außen nach innen alle vier Anschlüsse mit dem Draht vom Minuspol kurz an - rot, blau, weiß, gelb. Das gibt vier Schritte in die gleiche Richtung. Macht man das in der umgekehrten Reihenfolge, dreht der Motor rückwärts. So einfach geht das.

Nun machen wir das mit der MCU (einem Texas–Instruments–MSP430G2335 auf LaunchPad–Board). Ermutigt durch das Lautsprecherexperiment (siehe Wiki) wird eine Stepperspule direkt zwischen zwei Portpins angeklemmt, mal sehen, was passiert. Hm, nix - schade. Am Oszilloskop sieht man, dass die Pegel an den Pins runtergezogen werden, aber der Motor tut keinen Schritt. Tat er an der 3V–Batterie ja auch noch nicht. Dazu ist der Strom aus dem FET der MCU also wohl zu schwach.

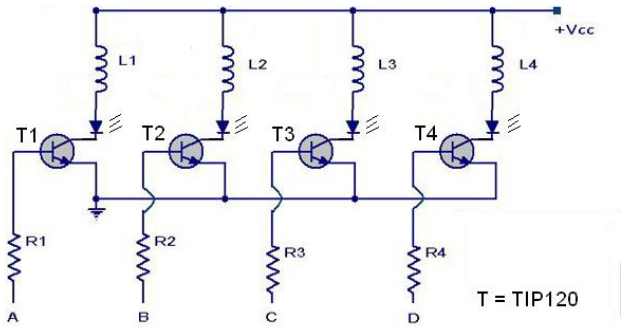


Abbildung 3: Schaltplan der Schrittmotor–Treiber

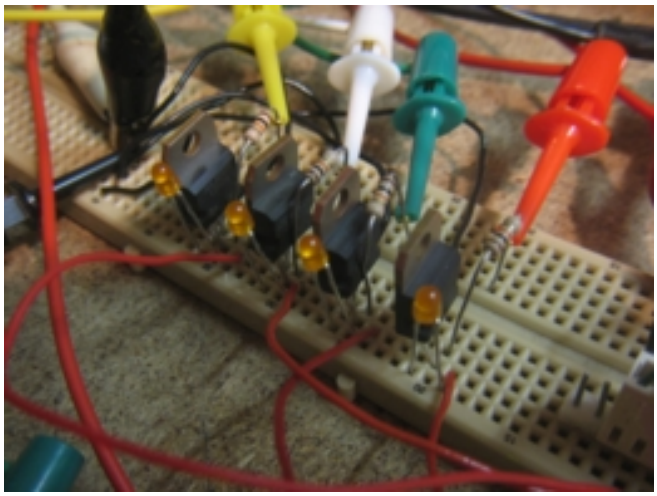


Abbildung 4: Testaufbau der Treiber

Da wir offensichtlich etwas mehr Strom dafür benötigen, als ein Port hergibt, muss ein TIP120 ran. Damit lässt sich das schön schalten. Mit vier Stück davon und je einem Basiswiderstand ist schon alles aufgebaut, was man braucht (**Abb. 3**). Die Freilaufdiode ist bei diesem Darlington-Transistor schon eingebaut. Damit ich sehen kann, welcher Transistor gerade durchschaltet, wurde jedem eine LED spendiert (**Abb. 4**). (Keine Sorge, das halten die aus an der 9V–Blockbatterie, weil die Spule schon einen Vorwiderstand darstellt, siehe oben.) Das Ergebnis wurde in kurzen Videoclips festgehalten, die auf der Internetseite zum Experiment aufgerufen werden können, siehe Links.

Mein Forthprogramm für den Schrittmotor ist dem Beitrag angehängt. Damit werden in experimenteller Weise drei digitale Antriebsmuster für den Stepper realisiert, wie sie im englischsprachigen Wikipedia erklärt sind. Im deutschen Wiki war das leider nicht so klar dargelegt [2].

Danach gibt es vier grundlegende Ansteuerungsmuster.

- Wave drive
- Full step drive (two phases on)
- Half stepping
- Microstepping

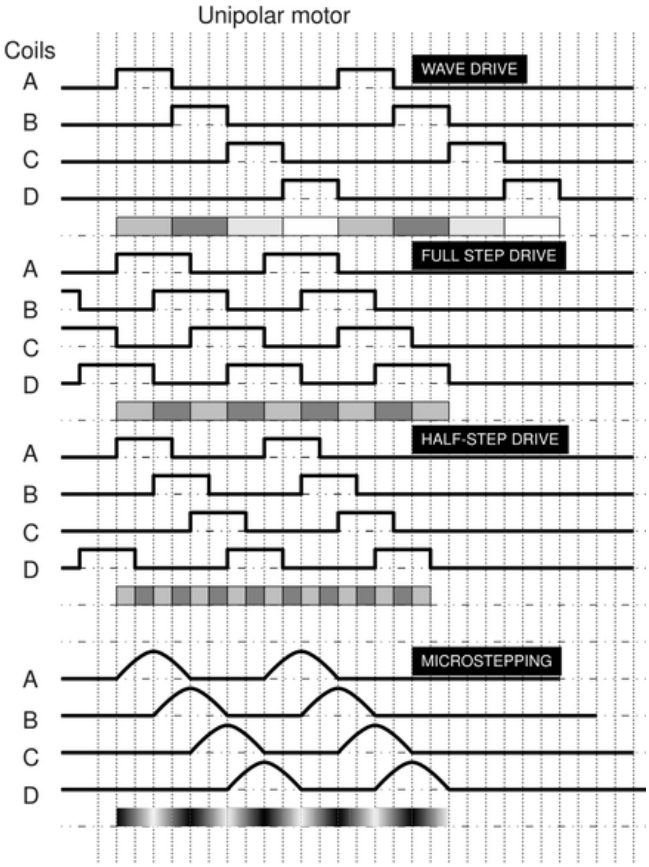


Abbildung 5: Schema der Ansteuerung

Die ersten drei sind digitale Formen, und einfach zu machen mit Forth. Im *full–step–drive* werden ganze Schritte ausgeführt. Der *wave–drive* macht das ebenso, hat aber nur immer eine Spule an, ist daher schwächer im Drehmoment. Beim *half–step–drive* ergeben sich halbe Schrittweiten, ohne dass man am elektrischen Aufbau etwas ändern müsste; allerdings sind dabei abwechselnd mal zwei, und dann nur eine Spule an, was unterschiedliche Drehmomente der Schritte zur Folge hat.

Es gibt sicherlich ausgefuchstere Steuerungen, um Schrittmotoren anzusteuern, und Bausteine, die das unterstützen, z. B. Piccolo Motor Control. Das herauszufinden, überlasse ich aber eurem Spieltrieb. Hier ging es mir um ein grundlegendes Experiment. Um daran die Arbeitsweise des Motörchens zu studieren. Statt der einzelnen Transistoren kann man auch eine integrierte Schaltung nehmen. z. B. einen L293D. Aber diskret aufgebaut, sieht man besser, wie das alles geht, finde ich. Viel Vergnügen beim Experimentieren.

Nachtrag: Schrott

Passt bitte auf beim Nachmachen!

Überspannung hilft beim Schrotten sehr. Bei meinen Experimenten mit den Spulen und Transistoren (z.B. TIP120, s. oben) wurde auch ein 12V/7Ah Bleiakkumulator an die Transistoren angeschlossen. Der kann kurzzeitig 2.1 A liefern. Aus Versehen wurde beim Zusammenstecken der Schaltung mit dem Pluspol vom Akku der Portpin P1.5 kurz berührt. Das genügte, um den Chip funktionsunfähig zu machen. Danach ging das Forthprogramm nicht mehr, keine Reaktion mehr der MCU. Aber beim dritten Versuch, das Programm neu zu flashen, also nach dreimaligem Löschen des Flash–Speichers, nahm die MCU das Programm doch wieder an, und lief brav los, als sei nichts gewesen. Lediglich P1.5 ließ sich nicht mehr schalten, zeigte konstant 918 mV Pegel. Die Programmer–Schaltung auf der Platine hatte wohl nichts abbekommen, USB seriell und Jtag gingen nach wie vor. Aber trauen kann man der MCU nun nicht mehr, wer weiß, was innen noch alles hin ist von all der Peripherie, die da drinsteckt. Glücklicherweise ist das LaunchPad billig. Und die MCU auch.

Also: Aufpassen, dass die Schaltung stimmt, bevor die Spannungsquelle angeschlossen wird — doppelt und dreifach checken!

Ihr bastelt auf eigene Gefahr — keine Garantie!

Referenzen

1. http://www.forth-ev.de/wiki/doku.php/projects:4e4th:4e4th:start:msp430g2553_experimente
2. http://en.wikipedia.org/wiki/Stepper_motor

Listing

```
1   \ Schrittmotor testen (Unipolarmotor)
2
3   \ Portpins als Ausgaenge betreiben und auf H oder L setzen,
4   \ um damit Transistoren zu schalten, die die Spulen im Motor treiben.
5
6   \ P2 ist eine Konstante, legt die Port2-Adresse des MSP430G225 auf den Stack.
7   BIN
8   : init   1111 P2 1+ cset ; \ P2.0..3 = out
9   : aus    1111 P2 cclr ;
10  : PULS ( x m adr -- x )  c! dup ms ;
11  : _A_    0001 p2 puls ;  \ Einen Puls am Port ausgeben.
12  : _B_    0010 p2 puls ;
13  : _C_    0100 p2 puls ;
14  : _D_    1000 p2 puls ;
15  : _AD_   1001 P2 puls ;
16  : _AB_   0011 P2 puls ;
17  : _BC_   0110 P2 puls ;
18  : _CD_   1100 P2 puls ;
19  DECIMAL
20
21  \ wave drive mode
22  : WDM ( x -- ) init aus
23    BEGIN  _A_ _B_ _C_ _D_ key? UNTIL  drop aus ;
24
25  \ full step drive
26  : FSD ( x -- ) init aus
27    BEGIN _AD_ _AB_ _BC_ _CD_ key? UNTIL  drop aus ;
28
29  \ half step drive
30  : HSD ( x -- ) init aus
31    BEGIN _AD_ _A_ _AB_ _B_ _BC_ _C_ _CD_ _D_ key? UNTIL  drop aus ;
32
33  ( finis )
```

Adventures 13: Eigene Funk–Sensoren empfangen

Erich Wälde und Martin Bitter

Es kommt der Tag im Leben eines Kontrollettis[1]*, wo es was zu messen gibt, an einer Stelle, an der ein Kabel (Strom, Kommunikation) keine Option ist. Das ist der Tag, wo er sich vielleicht daran erinnert, dass es bei Pollin [8] so nette kleine Module mit dem Namen* rfm12 *gibt, die auf den ISM–Funkbändern senden und empfangen. Also flugs welche bestellt und bekommen. Damit geht der Ärger aber erst richtig los. Stellt sich heraus, dass die Module kleine Primadonnen sind, die richtig gestreichelt werden wollen, bevor das mit der drahtlosen Datenübertragung halbwegs funktioniert.*

Im ersten Artikel [4] beleuchteten wir die Grundlagen. Im zweiten Artikel [5] wurde gezeigt, wie man mit der Betriebsart OOK (on–off–keying) käufliche Außensensoren für Luftfeuchte und –temperatur belauschen kann. Im dritten Artikel [6] wurde ein Füllstandssensor für die Zisterne vorgestellt, der periodisch Daten in der Betriebsart FSK (frequency–shift–keying) versendet. Dieser Artikel zeigt, wie die Daten des Füllstandsensors empfangen und verwaltet werden. Die Daten werden über die serielle Schnittstelle von einem Programm eingesammelt, welches auf einem weiteren (Linux–)Rechner läuft. Alternativ kann man sie direkt auf einem LCDisplay oder einer Reihe LEDs anzeigen.

Übersicht

Im ersten Teil dieser Artikelserie wurde beschrieben, wie man rfm12–Module an einen ATmega–Mikrokontroller anschließt, und wie man Datenpakete in der Betriebsart FSK (*frequency shift keying*) überträgt. Für diesen Teil benötigen wir zwei betriebsbereite Stationen. Die Sendestation (Adresse $22) verschickt regelmäßig ein Datenpaket (mit echten oder erfundenen Daten) über das Funkmodul. Die Empfängerstation (Adresse $60) belauscht den Funkverkehr. Erkennt das Empfänger–Funkmodul den Anfang eines Datenpakets, dann tut es das über einen Pegelwechsel an Pin `nIRQ` kund, welcher mit Pin `INT0` am Kontroller verbunden ist. Der Kontroller sammelt in der zu `INT0` gehörigen Interrupt–Service–Routine die empfangenen Bytes ein und legt sie im RAM ab (`D.in`). Ist ein vollständiges Datenpaket eingegangen, dann wird das Bit `wFcomplete` gesetzt. An diesem erkennt das Hauptprogramm, dass neue Daten vorliegen. Diese werden dann ausgewertet und in eine andere Datenstruktur eingepflegt.

Die Sendestation verschickt für jeden lokalen Sensor ein separates Datenpaket. Ein Sensor wird durch die Stationsnummer (z.B. `$22`) und die Sensornummer (z.B. `$01`) gekennzeichnet. Die Empfängerstation verwaltet eine Liste mit erwünschten Kombinationen der Stations– und Sensornummer. Für die erwünschten Sensoren wird Platz reserviert, der je einen Datensatz aufnehmen kann. Die erwünschten Sensoren und der dazugehörige Platz werden in einer Liste von Datenstrukturen verwaltet. Die darin enthaltenen Nutzdaten werden regelmäßig vom Dateneinsammler (`collector`) abgeholt.

Daher dreht sich der Großteil dieses Artikels um die Verwaltung der Daten.

Datenhaltung: `filter_mean`

Auf all meinen Stationen werden die Daten in einer Datenstruktur abgelegt, die es erlaubt, den Mittelwert und die Extrema auszugeben, ohne die einzelnen Messwerte aufzuheben. Zur Erinnerung: der Mittelwert $\bar{x}$ ist die Summe der Messwerte geteilt durch deren Anzahl:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

Also bietet es sich an, statt der Messwerte eben die Summe und die Anzahl aufzuheben, und auf Anfrage den Mittelwert zu berechnen und auszugeben. Die Summe wird in einer doppelt breiten Variablen (2 Zellen) abgelegt. Außerdem werden Minimum und Maximum abgespeichert. Damit kommt man auf die Größen N, $\sum x$, x_{min} und x_{max}, welche in 5 Zellen oder 10 Byte abzuspeichern sind. Zwar reagiert der Mittelwert recht empfindlich auf Ausreißer in den Messwerten, aber der Platzbedarf ist unschlagbar gering.

`filter_mean`: definiert die notwendige Datenstruktur, `mean_reset` löscht deren Inhalt, `mean_addup` fügt einen Messwert in die Datenstruktur ein und `mean_eval` berechnet den Mittelwert und legt alle Daten auf dem Stack ab. Weitere Details finden sich in [1].

Für die Übertragung per Funk habe ich den Mittelwert vorgesehen. Ich könnte zwar auch statt dessen die Summe übertragen (2 Byte mehr), dann könnte ich auf der Empfangsstation die Daten weiter kumulieren (Die Anzahl der enthaltenen Messwerte wird immer übertragen). Aber das gefiel mir nicht. Die Empfangsstation soll kein Wissen über die Bedeutung der Daten haben.

Datenhaltung: `data.frame`

Zu den gerade geschilderten 10 Byte Nutzdaten kommen 4 Byte als Kopf und 2 Byte für die Checksumme hinzu.

[1] a. zwanghaft Kontrolle ausübender Mensch; b. Mikrokontroller programmierender Mensch

Die einfachste Lösung bestand darin, Datenpakete mit einer konstanten Länge von 16 Byte zu verwenden.

Offset	Größe	Verwendung
0	1	Adresse Sender
1	1	Adresse Empfänger
2	1	derzeit unbenutzt (Bitflags)
3	1	derzeit unbenutzt (Datenlänge)
4	10	Daten
14	2	Checksumme

Für die einfachere Handhabung der Datenpakete (frames) habe ich ein paar Konstanten und Worte definiert. Die Größe und Positionen der Bestandteile im Datenpaket werden in Konstanten gepackt, deren Namen besser zu merken sind.

```
$10          constant df.size
df.size &6 - constant df.payloadsize
: data.frame:
  variable df.size allot
;
\ define header offsets
\ and add-offset functions
 &0 constant df:src  : df:src+ ( df:src + ) ;
 &1 constant df:dst  : df:dst+   df:dst + ;
 &2 constant df:len  : df:len+   df:len + ;
 &3 constant df:flg  : df:flg+   df:flg + ;
 &4 constant df:sid  : df:sid+   df:sid + ;
 &4 constant df:dat  : df:dat+   df:dat + ;
&14 constant df:crc  : df:crc+   df:crc + ;
```

Zum Löschen eines Datenpakets sowie zur Ausgabe der Daten gibt es weitere Funktionen. Zum einen wird das Paket schlicht in hexadezimalen Werten ausgegeben (`df.dump`), zum anderen wird die Struktur des Pakets lesbar dargestellt (`df.show`). Die Daten werden hier noch nicht weiter dekodiert.

```
: df.erase ( addr -- )  df.size erase ;
: df.dump  ( addr -- )
  df.size 0 do
    dup i + c@ 2 u0.r space
  loop
  drop
;
: df.show  ( addr -- )
  hex
  ." .....source:" space dup df:src+ c@ 2 u0.r cr
  ." destination:" space dup df:dst+ c@ 2 u0.r cr
  ." .....length:" space dup df:len+ c@ 2 u0.r cr
  ." ......flags:" space dup df:flg+ c@ 2 u0.r cr
  ." ...sensorid:" space dup df:sid+  @ 4 u0.r cr
  ." .......data: $" df.payloadsize 2/ 0 ?do
    dup df:dat+ i cells + @ space 4 u0.r
  loop cr
  decimal
  ." ............: &" df.payloadsize 2/ 0 ?do
    dup df:dat+ i cells + @ space .
  loop cr
  hex
  ." ........crc:" space dup df:crc+  @ 4 u0.r cr
  drop
;
```

Damit kann man ein zu verschickendes oder empfangenes Datenpaket schon einmal inspizieren (`D.tmp` beinhaltet die Adresse des Datenpakets im RAM):

```
> D.tmp df.dump cr D.tmp df.show
22 60 80 0A 01 22 08 00 FC 08 47 09 60 09 56 B6
.....source: 22
destination: 60
......flags: 80
.....length: 0A
...sensorid: 2201
.......data: $ 2201 0008 08FC 0947 0960
............ & 8705  8  2300  2375  2400
........crc: B656
 ok
>
```

Das Paket stammt von einer Station mit der Adresse `$22` und ist an die Station `$60` adressiert. Die Sensor–ID ist `$2201`.

Der Datensatz beinhaltet 8 Messwerte. Die Checksumme ist eine Fletcher–16–Checksumme [10]. Sie ist zwei Byte groß und lässt sich einfach prüfen: Der Datenblock inclusive der Checksumme ergibt als Checksumme immer null. Es kann natürlich jeder andere Algorithmus eingesetzt werden.

Es braucht jetzt noch ein Wort, welches die Nutzdaten in einem Datenpaket so ausgibt, dass der Dateneinsammler die Buchstaben interpretieren kann. Die Ausgabe soll gleich aussehen, wie die Ausgabe aus einem `filter_mean` Datensatz: $\mathrm{ID}_{\mathrm{station+sensor}}$, N, $x_{\min}$, x_{mean}, $x_{\max}$, also etwa so:

```
2201:8,+23.00,+23.75,+24.00
```

Für die Ausgabe braucht man noch eine weitere Information: wieviele Nachkommastellen sind in den Daten vorgesehen? Das ist eine Information, die im Datenpaket so nicht vorliegt. Die physikalische Einheit wäre ebenfalls erwünscht. Die nächste Iteration dieser ganzen Angelegenheit sollte ich vielleicht in dieser Richtung überdenken. Andererseits braucht nur der Dateneinsammler zu wissen, welche Einheit und welcher Skalierungsfaktor zu einer bestimmten Sensor–ID gehört.

Das Wort `.Dn` nimmt die SensorID, die Adresse der Nutzdaten und die Zahl der Nachkommastellen vom Stapel und gibt die Daten im gewünschten Format aus.

```
: .Dn ( addr prec -- )
  >r
  hex
  dup @ 4 u0.r   colon \ ID_{station+sensor}

  decimal
  dup 2 + @ s>d 1 du0.r  comma \ N
  dup 4 + @ r@ +.f       comma \ min
  dup 6 + @ r@ +.f       comma \ mean
  dup 8 + @ r> +.f             \ max
  drop
;

> D.tmp df:src+ 2 .Dn
2201:8,+23.00,+23.75,+24.00 ok
>
```

Daten gültig?

Wie schon erwähnt, habe ich mich für die Fletcher–16–Checksumme entschieden, um die Unversehrtheit eines Datenpakets zu prüfen. Der Algorithmus ist auf Wikipedia [10] erklärt.

```
variable crc.fl.sum1
variable crc.fl.sum2
: crc.fletcher16 ( addr len -- checksum )
  0 crc.fl.sum1 !
  0 crc.fl.sum2 !

  ( len ) 0 ?do
    dup i + c@
    crc.fl.sum1 @ + $00ff and
    dup crc.fl.sum1 !
    crc.fl.sum2 @ + $00ff and
    crc.fl.sum2 !
  loop
  drop

  crc.fl.sum1 @ crc.fl.sum2 @  +
  0 swap -  $00ff and
  crc.fl.sum2 @
  swap >< +
;
```

Empfangene Daten aufräumen

Im ersten Artikel dieser Serie [4] wurde beschrieben,

- wie das Funkmodul an den Mikrokontroller anzuschließen ist
- wie das Funkmodul für FSK–Empfang und die Erkennung von Datenpaketen konfiguriert wird
- wie der Kontroller die Daten vom Funkmodul abholt (Int0 Interrupt Service Routine)
- dass das komplette Datenpaket im RAM abgelegt wird.

Nachdem die korrekte Anzahl von Bytes empfangen wurde, muss der Empfang (Lauschen auf die `magic bytes`) wieder aktiviert werden. Es empfiehlt sich, vorher den gerade empfangenen Block von `D.in` nach `D.tmp` zu kopieren, und ihn danach in aller Ruhe auszuwerten. Dies geschieht in der Funktion `job.tick`, welche 128 mal pro Sekunde aufgerufen wird.

Nach dem Kopieren prüft das Programm, ob das Datenpaket intakt ist (Checksumme). Ist das der Fall, dann werden StationID+SensorID aus dem Datenpaket geholt. Ist diese ID bekannt, d.h. wird die ID in der Liste der bekannten Funk–Sensoren gefunden, dann wird das Datenpaket in den Speicherbereich eines solchen Sensors kopiert. Es gibt für jeden Funk–Sensor Platz für genau einen solchen Datensatz. Wird ein neuer Datensatz empfangen bevor der vorhandene abgeholt wurde, dann wird der alte Datensatz überschrieben.

```
0 value dfList
: >dfList  ( addr -- ) dfList list.add to dfList ;
\ station-22,
\     sensor-01, temperature, lm75
data.frame: D.2201   2 D.2201 $2201 df.head D1
D1 >dfList
```

Die Adressen der `data.frame`–Datenbereiche werden zusammen mit der SensorID und der Anzahl der Nachkommastellen in einer einfach verketteten Liste geführt. Der Anfang der Liste wird in der EEPROM–Variablen `dfList` gespeichert. Im letzten Element der Liste enthält der Zeiger auf das nachfolgende Element den Wert null (beschrieben in [3]).

```
data.frame: D.in
data.frame: D.tmp
...

variable Ncrcerror
: job.tick
  wFcomplete fset? if
    ledsensor low

    D.in D.tmp df.size cmove \ copy frame
    0 df.index !             \ clear counter
    D.in df.size erase       \ clear block

    wc? drop
    w.rx.clearfifo
    wc? drop
    wFcomplete fclr
    wFactive fset

    D.tmp df.size crc.fletcher16 0= if
      D.tmp df:dat+ @   ( ID_station+sensor )
      dfList df.list.find
      dup 0 > if \ sensor wanted
        D.tmp over df.size cmove
      then
      drop \ addr
    else
      1 Ncrcerror +!
    then
    ledsensor high
  then
;
```

In `job.tick` wird geprüft, ob ein kompletter Datensatz eingegangen ist. Wenn ja, dann wird dieser von `D.in` nach `D.tmp` kopiert und das Funkmodul aktiviert. Danach wird die Checksumme berechnet. Ist sie gut, dann wird die SensorID in der oben erwähnten Liste der gewünschten Sensoren gesucht. Ist sie vorhanden, dann wird der Inhalt von `D.tmp` in den zugehörigen `data.frame:` umkopiert.

Daten abholen

Wie schon erwähnt, werden die Daten periodisch von einem Rechner abgeholt, der die Empfangsstation über einen RS485 Bus an der seriellen Schnittstelle erreicht. Der Bus wurde in [2] beschrieben. Der Rechner agiert als Busmaster. Der Busmaster adressiert die Station und schickt im Wesentlichen den Befehl `~data`. Dieser gibt die vorhandenen Daten in dem oben erwähnten Format als Text aus.

```
: df.list.reset ( addr -- )
  1+ @i df.size erase
;
: df.list.eval ( addr -- )
  dup 2 + @i              \ -- addr prec
  swap 1+ @i              \ -- prec df
  dup @  0= not if        \ -- prec df
    \ frame with data
    df:dat+ swap          \ -- df:dat prec
    .Dn
  else
    \ frame erased
    drop drop
  then
;

: data.reset
  0 data_samples !
  ['] df.list.reset dfList list.walk
;
: data.ls
  ['] df.list.eval  dfList list.walk
;
: ~data
  leddata low
  ." __Q"       \ datagram start
  .id+ver       \ stationID + swVersion
  decimal       \ uptime
  space zero colon .uptime_sec space
  data.ls       \ sensor data
  data.reset
  space ." C--" \ datagram end
  leddata high
  lederr  high
;
```

Die folgende Ausgabe stammt von der Station $60, die Daten von der Station $22 empfängt, welche Daten von einem Temperatur–Sensor versendet. Die Ausgabe wurde von Hand umgebrochen, die dritte Zeile (0:0) besagt, dass kein fifo-overflow entdeckt wurde (s. So Sachen, 3.):

```
> ~data
__Q 60:0004 0:846s
  0:0
  2201:8,+21.00,+21.00,+21.00
  C-- ok
```

Die Zeichenketten __Q und C-- dienen lediglich dazu, die Daten zu markieren. Das macht dem Einsammler die Arbeit leichter.

Die Datensätze für jeden einzelnen Sensor werden in einer sqlite–Datenbank gespeichert. Ein Anzeigeprogramm liest die Daten aus der Datenbank und malt daraus Diagramme. Das Anzeigeprogramm ist in perl geschrieben und benutzt die alte pgplot–Bibliothek zum Zeichnen der Diagramme. Das hat mir natürlich schon den entsprechenden Seitenhieb eingebracht: *Warum machst Du das denn nicht in Forth?* Also ergeht hier der Aufruf an die, die das besser können, mal einen Artikel zu spendieren, in dem das vorgeführt wird: Daten aus einer sqlite–Datenbank abfragen und in Diagramme verwandeln, z.B. in Bigforth/Minos. Zweite Aufgabe: den Einsammler ersetzen, also für jede Station periodisch die Daten per serielle Schnittstelle abfragen und die erhaltenen Daten in die Datenbank eintragen. Es kann vorkommen, dass eine Station nicht sofort korrekt antwortet. Davon soll sich der Einsammler nicht stören lassen.

Sicher kann in dem vorliegenden Programm noch einiges verbessert werden. Man gelangt aber recht schnell in das Fahrwasser der Protokolle. Sollte man den Empfang bestätigen? Oder lediglich die Aussendung wiederholen? Sollte der Sender zuerst empfangen, und gucken, ob die Luft rein ist? Was wenn der Datensatz oder die Bestätigung nicht empfangen wird? Damit haben sich schon klügere Leute beschäftigt. Für meine Zwecke reicht das so, für verkaufbare, selbstkonfigurierende Mesh–Sensor-Netzwerke ist das sicher zu wenig.

So Sachen

1. Zum Testen des Programms für diesen Artikel hatte ich eine andere Platine genommen. Und es wollte und wollte nicht klappen, obwohl ich doch das gleiche Programm geladen hatte, welches seit Wochen klaglos seinen Dienst tut. Aber Software lebt bekanntlich nicht im luftleeren Raum: eine (zum Glück sichtbar) schlechte Lötstelle auf der rfm12–Adapterplatine sorgte für eine Unterbrechung. Den Kontakt nachgelötet, und schon schnurrt alles bestens.

2. Manchmal blieb die Station mit dem einwandfrei funktionierenden Programm dennoch einfach stecken. Zwar konnte ich mit dem Kontroller an der seriellen Schnittstelle reden, aber von Datenempfang (oder –versand) keine Spur. Das ließ sich durch mehr Speicherplatz für die Stapel in den Hintergrundprozessen (tasks) entschärfen. Vorher reservierte ich $20 Byte, jetzt $40. Damit ist das Symptom weniger geworden — immer nach der Devise *man sieht nur bis zum nächsten Baum!*

3. Langwieriger zu finden war die Ursache dafür, dass das Funkmodul gelegentlich einfach den Empfang verweigert und dem Dateneinsammler dann auch nichts Neues mehr berichten kann. Nach längerem Grübeln kam ich drauf, dass das Funkmodul aus irgendwelchen Gründen einen *fifo–overflow*–Fehler anzeigt und dann den weiteren Empfang von Daten unterlässt, und seien sie noch so schön. Die Kur war dann eigentlich recht einfach: Falls dieser Zustand vorgefunden wird, werden die Zähler, Flags und Datenbereiche der Empfangsroutinen, sowie das Funkmodul neu initialisiert. Dieser Test wird einmal jede Sekunde durchgeführt.

```
variable dbg.fsk.restar
&13 bv constant wS.fifo-overflow
...
: rfm12.fsk.restart
  0 wFlags !
  D.in df.size erase
  0 df.index !
  +rfm12.rx.isr
;
...
: job.sec
  ++uptime
  wc? wS.fifo-overflow and if
```

```
    rfm12.fsk.restart
    1 dbg.fsk.restart +!
  then
;
: init
  ...
  0 dbg.fsk.restart !
;
```

Dieses Vorgehen führt dazu, dass der Empfang weitergeht. Allerdings erklärt es kein bisschen, wie dieser Zustand zustandekommt. Ich lasse auch mitzählen, wie häufig das passiert. Diese Zahl wird an den Dateneinsammler übertragen, wandert jedoch nicht in die Datenbank.

4.

Zum Datenübertragen gibt es nichts Besseres als eine Kabelverbindung. Diese Weisheit bestätigt sich immer wieder, wenn man mit Funkverbindungen experimentiert. Über den Einfluss der Antennen hatte ich schon einmal berichtet. Antennen sind ein sehr weitläufiges Thema in Funkerkreisen. Antennen sind das schwächste Glied in der Übertragunsgkette und daher entsprechend wichtig. Ein Bekannter berichtet, dass seine Funkverbindung erst mit dem Einsatz einer primitiven *ground–plane–monopol*–Antenne zuverlässig wurde. Ein Bandfilter zwischen der Antenne und dem Empfänger könnte auch helfen, aber das habe ich nicht ausprobiert. Und selbst wenn der Datenverkehr eine Weile (mehrere Wochen) recht zuverlässig funktioniert, dann setzt er möglicherweise dennoch über Stunden oder Tage aus — ohne sichtbaren Grund. Und genau da liegt das Problem: sichtbar sind die Funkwellen bei 70 cm halt für unsere Augen nicht. Beim kabelgebundenen Erfassen von Wetterdaten kommt man dann allerdings recht schnell mit dem Thema Blitzschutz in Berührung — nichts ist umsonst.

Referenzen

1. E. Wälde, Adventures in Forth 5, Die 4. Dimension 4/2008, Jahrgang 24
2. E. Wälde, Adventures in Forth 6, Die 4. Dimension 1/2011, Jahrgang 27
3. E. Wälde, eine einfache Liste, Die 4. Dimension 2/2011, Jahrgang 27
4. E. Wälde und M. Bitter, Funklöcher!, Die 4. Dimension 3/2011, Jahrgang 27
5. E. Wälde und M. Bitter, Funksensoren belauschen, Die 4. Dimension 4/2011, Jahrgang 27
6. E. Wälde und M. Bitter, Funk–Füllstandsensor für die Zisterne
7. `http://amforth.sourceforge.net/`
8. `http://www.pollin.de`
9. `http://www.hoperf.com`
10. `http://en.wikipedia.org/wiki/Fletcher%27s_checksum`

Listings

Wie schon häufiger ist auch diesmal der komplette Programm–Text viel zu lang zum Abdrucken. Deswegen zeige ich hier lediglich eines der beiden verwendeten Testprogramme, `main-rx.fs`. Dieses Programm kann die Daten des Füllstand–Sensors aus dem letzten Artikel ebenfalls empfangen, wenn sie in die Liste der erwünschten Sensoren eingetragen werden. Alle Dateien finden sich im Downloadbereich der VD. Die Programme wurden auf einem atmega32 unter amforth-4.6 entwickelt und getestet.

```
\ 2012-09-20  ew  main-rx.fs

\ receiver for FSK dataframes (not OOK)
\ StationID: 0x60

\ amforth 4.6 atmega32
\ incl. stationid,rs485,mpc,rec-quiet
\ NO timeup clock!
\ onlytask:   cmd loop
\ task_intro: diagnostic stuff, then awake
\             task_job
\ task_job:   collect and send sensor data

\ included by make marker:
\ lib/misc.frt
\ lib/bitnames.frt
\ lib/ans94/marker.frt
\ $(AMFORTH)/devices/$(MCU)/$(MCU).frt
\ first.fs

marker --start--
include lib/ans94/postpone.frt
include ewlib/format.fs \ sign! .i +.f
\ bitvalue, convert bit number [0..7] to mask
: bv    ( bit# -- mask )
  1 swap lshift ;
: or!   ( mask addr -- )
  dup c@ rot or  swap c! ;
: and!  ( mask addr -- )
  dup c@ rot and swap c! ;
: comma ( -- )   [char] , emit ;
: colon ( -- )   [char] : emit ;
: dot   ( -- )   [char] . emit ;
: .id   ( id -- )   space .i colon ;

\ --- values and variables ------------------
$60    value EEstationID
$0004  value swVersion
\ needed by +mpc7:
\ USART Control and Status Register C
\ shared with UBRRH!
$40 constant UCSRC
&65 constant data_samples_not_collected
    variable data_samples
    variable dbg.fsk.restart

decimal

\ --- additional exception numbers ----------
-200 constant #timeout

\ --- pin definitions -----------------------
PORTA 0 portpin:  vsupply
\   A 1
\   A 2
PORTA 3 portpin:  sw.user
PORTA 4 portpin:  pwr.leds
\   A 5
\   A 6
\   A 7

PORTB 0 portpin:  lederr
PORTB 1 portpin:  ledsec
PORTB 2 portpin:  ledsensor
PORTB 3 portpin:  leddata
PORTB 4 portpin:  /ss
PORTB 5 portpin:  _mosi
PORTB 6 portpin:  _miso
PORTB 7 portpin:  _clk

\   C 0           _scl
\   C 1           _sda
PORTC 2 portpin:  _rfm12
PORTC 3 portpin: sht_clk
PORTC 4 portpin: sht_dat
\   C 5
\   C 6           32 kiHz Quarz
\   C 7           .

\   D 0           RxD
\   D 1           TxD
PORTD 2 portpin:  _int0
PORTD 3 portpin:  _int1
\   D 4
PORTD 5 portpin:  pwr.ping
PORTD 6 portpin:  ping
PORTD 7 portpin:  /rs485.rw

\ --- bus addresses -------------------------
$96 constant a_th1              \ lm92

\ --- sensor offsets ------------------------
0     value  c0_T1              \ 1/100 C


\ --- famous includes -----------------------
include ewlib/flags.fs
: blank bl fill ;
: or!  dup c@ rot or swap c! ;
include lib/ans94/2x.frt
include ewlib/mstarslash.fs      \ m*/

\ --- clock tick, sleep ---------------------
2variable uptime
: .uptime_sec uptime 2@ d.i [char] s emit ;
: .uptime
  uptime 2@
  2dup decimal d.i [char] s emit space
  &60 ud/mod  &60 ud/mod  &24 ud/mod
  d.i    [char] d emit space    \ days
  2 u0.r [char] : emit          \ hours
  2 u0.r [char] : emit          \ minutes
  2 u0.r                        \ seconds
;
```

```
: ++uptime  1 s>d uptime 2@ d+ uptime 2! ;
: zero  ( -- )   [char] 0 emit ;
: comma ( -- )   [char] , emit ;
: colon ( -- )   [char] : emit ;
: dot   ( -- )   [char] . emit ;
: .id   ( id -- ) space hex .i colon ;
: .id+ver ( -- )
  EEstationID  .id swVersion 4 u0.r
;
include ewlib/clock_tick.fs
include ewlib/timeup.fs

include ewlib/spi.fs
: +spi.lsb.first
  SPCR c@  %00100000 or         SPCR c! ;
: +spi.msb.first
  SPCR c@  %00100000 invert and SPCR c! ;

include ewlib/twi.fs    \ +twi -twi twi.ping?
                        \ twi.scan
include ewlib/i2c.fs    \ >i2c <i2c
include ewlib/i2c_lmXX.fs \ lm.get
                        \ lm{75,92}.decode

\ --- data handling -------------------------
include ewlib/filter_mean.fs   \ filter_mean:
            \ mean_reset mean_addup mean_eval
include ewlib/filter_mean_show.fs \ .F1
      \ .F2 ( id max mean min N>0 | id 0 -- )

include ewlib/list.fs \ list.add,show,walk

include ewlib/crc-fletcher16.fs
       \ crc.fletcher16 ( addr len -- chsum )

include ewlib/spi.fs    \ /ss +spi -spi

include lib/multitask.frt
include ewlib/task.fs

\ include ewlib/mpc.fs
include ewlib/case.fs

decimal
\ --- sensors + filters ---------------------
include ewlib/data-frame.fs  \ data.frame:
                   \ df.show df.erase df.dump
data.frame: D.in
data.frame: D.tmp
variable df.index
: df.index++
  df.index @ 1+  df.size mod   df.index !
;
include ewlib/dot-d-n.fs

0 value dfList
: >dfList  ( addr -- )
  dfList list.add to dfList ;

include ewlib/data-frame-list.fs

\ station-22,
\     sensor-01 temperature, lm92
data.frame: D.2201   2 D.2201 $2201
    df.head: D1   D1 >dfList

: dd ( D.x -- )
  hex
  df.index @ 4 u0.r space
  dup df.dump cr
  dup df.show  ." ...checksum:" space
  dup df.size crc.fletcher16
  dup 4 u0.r space
  0= if
    ." ok"
  else
    ." CRC ERROR"
  then cr
  ." ...data.set:" space
  dup df:dat+
  2 .Dn cr \ print id:N,min,mean,max record
  drop ( D.x )
;

\ --- wireless: fsk -------------------------
include ewlib/rfm12.fs

: rfm12.init-ook ( -- )
  $8017 >wc
  $C431 >wc
  $a608 >wc
  $94c2 >wc
  $C220 >wc
  $82D8 >wc
  $CA00 >wc
  $CC67 >wc
  $C623 >wc
;
: rfm12.off      ( -- ) $8201 >wc ;
: rfm12.tx.on    ( -- ) $8238 >wc ;
: rfm12.tx.off   ( -- ) $8208 w? >wc ;
: rfm12.rx.on    ( -- ) $82c8 >wc ;
: w.rx.clearfifo ( -- ) $ca81 >wc $ca83 >wc ;

&13 bv constant wS.fifo-overflow \ in status

\ wireless: fsk, receive data ---------------


PORTD 2 portpin: _int0
variable wFlags           \ status flags
wFlags 0 flag: wFactive   \ receiving
wFlags 1 flag: wFcomplete \ receive complete
wFlags 2 flag: wFtmp      \ new data in D.tmp
include ewlib/rfm12_rx_int0.fs
: +rfm12.rx.isr
  ledsec low
  wc? drop
  rfm12.rx.on
  wc? drop
  w.rx.clearfifo
  wc? drop
  wFcomplete fclr
  wFactive fset
\  +int0
  ledsec high
;

: rfm12.fsk.restart
  0 wFlags !
  D.in df.size erase
  0 df.index !
```

```
  +rfm12.rx.isr
;

\ wireless: fsk, process data ---------------
\ --- data: reset, collect, show ------------
: data.reset
  0 data_samples !
  ['] df.list.reset dfList list.walk
;
: data.collect
  noop
;
: data.ls
  ['] df.list.eval  dfList list.walk
;

\ --- tilde commands ------------------------
\ : ~end  -emit +mpc7 ;
: ~id   .id+ver ;
\ : ~call mpc_call ;
: ~status
  .id+ver space
  .uptime space
  dbg.fsk.restart @ u.
;
: ~info
  ." __I 2201:ffff:T[C/100]:600:Adv13-Test: C-33
  cr
;
: ~data
  leddata low
  ." __Q"       \ datagram start
  .id+ver       \ stationID + swVersion
  decimal       \ uptime
  space zero colon .uptime_sec space
  space zero colon dbg.fsk.restart @ u. space
  data.ls       \ sensor data
  data.reset
  space ." C--" \ datagram end
  leddata high
  lederr  high
;

\ --- timeup jobs ---------------------------
variable Ncrcerror
: job.tick
  wFcomplete fset? if
    ledsensor low

    D.in D.tmp df.size cmove \ copy frame
    0 df.index !             \ clear counter
    D.in df.size erase       \ clear block

    wc? drop
    w.rx.clearfifo
    wc? drop
    wFcomplete fclr
    wFactive fset

    D.tmp df.size crc.fletcher16 0= if
      D.tmp df:dat+ @   ( ID_station+sensor )
      dfList df.list.find
      dup 0 > if \ sensor wanted
        D.tmp over df.size cmove
      then
      drop \ addr
    else
      1 Ncrcerror +!
    then
    ledsensor high
  then
;
variable N
: job.sec
  ++uptime \ ledsec toggle

  wc? wS.fifo-overflow and if
    rfm12.fsk.restart
    1 dbg.fsk.restart +!
  then
;
: job.min   noop ;
: job.hour  noop ;
: job.day   noop ;
: job.month noop ;
: job.year  noop ;

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

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

\ --- init ---------------------------------
: init-io
  lederr    high  lederr    pin_output
  ledsec    high  ledsec    pin_output
  ledsensor high  ledsensor pin_output
  leddata   high  leddata   pin_output
;
: init-mem
  1 jobCount !
  0 tuFlags !
  0 s>d uptime 2!
  timer2 @ cycles/tick + newtimer !

  0 N !
  0 Ncrcerror !
  D.in df.erase
  D.tmp df.erase
  0 df.index !
  0 wFlags ! \ clear wireless flags
  $ff PORTA c! \ debug: port A output
  $ff DDRA c!  \ debug
  0 dbg.fsk.restart !
;

: init
  init-io
  init-mem
  init-timeup

  lederr low

  +ticks-slow-blink
```

```
  +twi
  -jtag

  +spi
  +rfm12 &1000 ms rfm12.init
  int0.enable +int0
  +rfm12.rx.isr
  wc? drop \ w.status
  data.reset

  500 ms
  lederr high
;

\ --- task handling -------------------------
\ create task data structure and a word to
\ fill it
task: t_intro   0 value tid_intro
task: t_job     0 value tid_job

Edefer switch-tasks
' noop is switch-tasks

\ --- task: intro ---------------------------
: run-intro
  &10 0 do
    lederr low
    100 ms
    lederr high
    900 ms
  loop
  -ticks
  lederr    high
  ledsec    high
  ledsensor high
  leddata   high
  switch-tasks
  +ticks
  stop
;

: start-intro
  t_intro          \ fill user area
  dup to tid_intro \ store tid
  activate         \ add indirected call to
  run-intro        \ <-- this statement
;

\ --- task: job -----------------------------
\ parallel form of timeup:
\ do not call job.sec ... job.year in one go.
\ always call
\     tickover? if timeup ... then
\ after every job.$n
: run-job
  begin

    \ these are run often
    tickover?
    if
      timeup
      job.tick
      1 jobCount ! \ start with job.sec
    then

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

    pause
  again
\  key? until
;

: start-job
  t_job
  dup to tid_job
  activate
  run-job
;

\ --- tasks: switch + start -----------------
: (switch-tasks)
  leddata low
  tid_job task-awake
  &100 ms
  leddata high
;
: starttasker
  \ init Edefer
  ['] (switch-tasks) is switch-tasks

  \ start tasks
  start-intro
  start-job

  \ organize task handling
  onlytask
  tid_intro  alsotask
  tid_job    alsotask   tid_job task-sleep

  \ start tasker
  multi
;

\ --- run-turnkey ---------------------------
: run-turnkey
  applturnkey

  init
  starttasker
;

\ ' run-turnkey is turnkey
```

Die lokalen Variablen von Hansen in High-Level-Forth und ohne Bracket-Tick

Fred Behringer

Sieben Primitives aus der Vorschlagsliste von Stricker [6] reichen zur Einrichtung von lokalen Variablen im Sinne von Hansen [5] in High-Level-Forth.

Früherer Versuch

Im VD-Heft 3/2012 hatte ich einen Vorschlag von Hansen [5] zur zeitweisen Verwendung einer schon bestehenden Variablen als lokale Variable (innerhalb einer Colon-Definition) besprochen [3]. Es sollte alles in High-Level-Forth und ohne Immediate-Worte erledigt werden. Dabei schrak ich vor dem von Hansen [5] eingesetzten Bracket-Tick `[']` zurück, der mir zu komplex erschien, als dass er sich leicht aus den von Stricker [6] vorgeschlagenen 26 Primitives herleiten ließe. Um Zeit zur Überlegung zu gewinnen, wich ich im VD-Heft 3/2012 [3] auf eine Klammerung der Form local[...]global aus. Im Vorliegenden zeige ich, dass man Hansens ursprüngliche Idee nur mit der schon bestehenden `Variablen` `xxx` und der Konstruktion `: www xxx local <name1> <name2> ... ;` durchaus auch ohne den Bracket-Tick `[']`, allein mit den Primitives `(lit) >r r> dup swap ! @` aus der von Stricker [6] angegebenen Liste, durchziehen kann.

Aus dem Leserbrief von Hansen [5]

Ich übersetze: „Meine Version, (d. h. die Version von Henning Hansen [5]) einer „Lokalen Variablen“ verwendet eine ganz normale Forth-Variable, die außerhalb der Colon-Definition definiert wurde und dort i. A. auch verwendet wird. Das einzige Wort, das hinter dem Namen der Variablen (also `xxx`) ins Spiel kommt, ist `local`. Das Wort local muss am Anfang einer Colon-Definition stehen und darf nicht innerhalb von Kontroll-Strukturen oder anderer Art von Returnstack-Manipulationen auftreten.“

Ganz bin ich (der Autor des vorliegenden Artikels) mit dem bis zu diesem Punkt von Hansen [5] Gesagten nicht einverstanden: Einfachste Beispiele (siehe Listing) zeigen, dass das Wort `local` durchaus auch weiter hinten in der zugrundegelegten Colon-Definition stehen kann, ohne dass dadurch an Hansens Idee (nur mit einem einziges Wort zur Kennzeichnung des lokalen Bereichs auszukommen) gerüttelt werden müsste. Weiter sagt Hansen [5]:

„Die besagte Variable kann danach frei innerhalb der Colon-Definition verwendet werden und wird nach deren Verlassen auf ihren ursprünglichen Wert zurückgesetzt. Da der natürliche Wirkungsbereich einer lokalen Variablen in Forth eine Colon-Definition ist, können lokale Variablen über den Returnstack eingebunden werden.

Das Wort `local` sichert zunächst die Adresse und den Wert der Variablen auf dem Returnstack. Dann richtet es einen Aussprung über `(local)` ein, wobei es die Adresse von `(local)` auf den Returnstack legt.

Beim Aussprung aus der Colon-Definition, die den lokalen Bereich der Variablen bildet, greift `(local)` dann auf den Returnstack zu und stellt mit dem dort bereitstehenden Wert den ursprünglichen Wert der Variablen wieder her.“

Soweit die Erklärungen von Hansen [5]. Ich hoffe, dass ich den Sachverhalt mit meiner Übersetzung hinreichend genau wiedergegeben habe. Ich verzichte darauf, die eigentliche Idee von Hansen ausführlicher zu kommentieren, und konzentriere mich auf eine Erkenntnis, mit deren Hilfe es mir gelingt, den Bracket-Tick von [5] durch Primitives aus [6] auszudrücken.

Mein Ziel: Vermeidung des Bracket-Ticks

Hansens Verwendung des Forth-Wortes `[']` (siehe Listing) sprang ins Auge, störte mich aber. Ich wollte neben der ausschließlichen Verwendung von nur High-Level-Forth-Worten (wie ja auch schon bei Hansen) kein Wort zulassen, das sich nicht in einfacher Weise aus dem Basis-Set von Primitives von Willi Stricker [6] herleiten ließe. Ich wollte `[']` aus den Vorschlägen von Hansen [5] beseitigen. `[']` hat außerdem den nur bedingt zu verteidigenden Nachteil, ein Immediate-Wort zu sein — und das störte mich ganz besonders. (Wozu sind Immediate-Worte gut? Könnte man Forth nicht auch ganz ohne Immediate-Worte aufbauen?) Ich gehe vom Inneren einer Colon-Definition aus und zeige hier, dass ich mit der Wortfolge

```
(lit) (local) 1 + 1 +
```

anstelle der von Hansen eingesetzten Folge

```
['] (local) >body
```

mein Ziel der Vermeidung von `[']` ganz einfach erreiche. Dabei ist `(lit)` das in Turbo-Forth so bezeichnete Primitive, das das (im Quelltext) unmittelbar hinter ihm stehende Wort (bei späterer Ausführung des Compilats) als 2-Byte-Wert auf den Datenstack legt — und das Programm unmittelbar hinter diesem 2-Byte-Wert (im Compilat) fortsetzt.

Was sagt Turbo-Forth zum Gesagten?

Geht man die Turbo-Forth-Literatur durch, dann wird man beim Bracket-Tick `[']` stutzig: In einer (offenbar) frühen Version wird `[']` (im Kernel) in High-Level-Forth (unter Verwendung von literal) implementiert und

scheint nichts mit (lit) zu tun zu haben. (lit) ist eine ganz normale Code-Definition, die nicht immediate ist. (lit) greift, wir befinden uns innerhalb einer Colon-Definition zur Laufzeit, auf den hinter ihm liegenden 2-Byte-Wert zu, legt ihn auf den Datenstack und setzt das Programm hinter diesem 2-Byte-Wert fort. ['] dagegen ist (in der frühen Version von Turbo-Forth) ein Immediate-Wort. Was macht dieses Immediate-Wort? Es legt beim Compilieren die cfa von (lit) ins Dictionary (die cfa von (lit) — natürlich unter dem Namen von ['] !) und verhält sich dann aber bei seiner Ausführung genau so, wie man es von (lit) (von (lit) !) erwartet. Wozu dann also der Umstand mit ['] und der damit verbundenen Immediacy-Eigenschaft? Das hat man bei der Implementation von Turbo-Forth offensichtlich auch gesehen und in späteren Versionen von Turbo-Forth das Wortpaar ['] (local) durch das Wortpaar (lit) (local) ersetzt. Man könnte fast sagen, es wurde einfach ['] durch (lit) ersetzt. Ganz reicht das nicht, da (lit) — wie auch schon ['] — nur im Verein mit dem darauffolgenden 2-Byte-Wert (hier die cfa von (local)) einen Sinn ergibt. Aber der Bracket-Tick ist dann weg!

Was bleibt noch zu sagen?

Bei Hansen [5] steht noch >body. Das wird in Turbo-Forth (und in anderen Sytemen auch) durch das Wort : >body 2+ ; wiedergegeben. Hier kann man 2+ durch 1 + 1 + ersetzen. Und + lässt sich in Strickers Basic-Kernel auf Primitives aus der Liste [6] zurückführen.

Wo der Festwert 1 herkommt, d. h., wie ein solcher Wert erzeugt wird, soll mich hier nicht interessieren. Ich halte mich für den Moment einfach an [6], wo auch andere Festwerte vorkommen, über deren Herkunft nichts gesagt wird.

Alles Übrige läuft wie bei Hansen [5] und ich darf (auch) im Listing auf weitere Erläuterungen verzichten.

Geht es auch ganz ohne Immmediate-Worte?

In den ANS-Forth-Empfehlungen [1] sind die Aussagen über Immediate-Worte nicht sehr kräftig. Das interessante ANS-Forth-Programmierer-Handbuch [4] stützt sich stark auf [1] und weist den Bracket-Tick ['] ausdrücklich als Immediate-Wort aus. ['] gehört zum ANS-Core-Word-Set und damit zu Forth als Sprache. Hm...! In [2] versuche ich, Kontroll-Strukturen ohne Immediate-Worte auszudrücken. Übrigens wird auch in den Handbüchern von Turbo-Forth der Bracket-Tick als Immediate-Wort geführt, obwohl in den neueren Versionen von Turbo-Forth (siehe oben) von Immediacy keine Rede mehr ist.

Ist ['] (über (lit) hinaus) überflüssig?

In den ANS-Forth-Empfehlungen [1] steht: "Die interpretativen Eigenschaften von ['] sind nicht definiert". Und: „Außerhalb von Colon-Definitionen bedeutet ['] <wort> dasselbe wie ' <wort>“.

Ulrich Hoffmann verdanke ich einen redaktionellen Hinweis darauf, dass auch VolksForth schon ohne ['] bei LOCALS ausgekommen ist. Ich selbst habe mich im vorliegenden Artikel nach vielen Hin- und Her-Überlegungen (vergleiche [3]) darüber gefreut, schließlich zu erkennen, dass ich mit nur 7 Primitives aus der Vorschlagsliste von Willi Stricker [6] durchkomme. 6 davon kommen auch schon im Leserbrief-Vorschlag von Henning Hansen [5] vor. Was also (über [5] hinaus) zur Lösung meiner Unschlüssigkeit aus [3] als einziges zusätzliches Primitive berücksichtigt werden müsste, wäre der ebenfalls in Strickers Primitive-Liste enthaltene Literal-Handler (lit) aus Turbo-Forth oder F83 oder ZF (in Turbo-Forth und ZF als Code-Definition, also als Inbegriff eines Primitives).

Literatur

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

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

[3] Behringer, Fred: Lokale Variablen in High–Level–Forth, ohne Immediacy und ohne Bracket–Tick. Vierte Dimension 3/2012, S. 12-14.

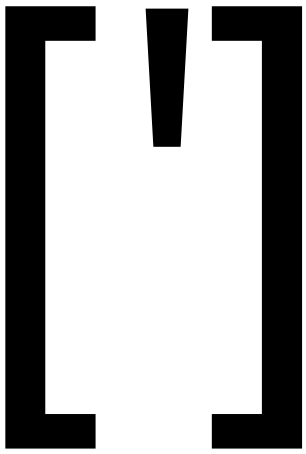
[4] Conklin, E.K., and Rather, E.D.: Forth Programmer's Handbook. Forth. Inc., California, USA (1997).

[5] Hansen, Henning: Leserbrief in der Vierten Dimension 1/1988, TU–Dänemark, Lynby.

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

Listing

```
1   hex
2
3   variable xxx             \ Zur Vorbereitung. Jedes vorhandene xxx kann dafuer genommen werden.
4   4711 xxx !               \ Als Diskussionsbeispiel fuer die Belegung mit einem globalen Wert.
5
6
7   : (local) r> r> ! ;
8
9   : local ( xxx -- )
10     r> swap dup @ swap
11     >r >r
12  \ ------------------------------------------------------------------------------------
13  \  ['] (local) >body     \ Hier kommt der von Hansen [5] urspruenglich eingesetzte Teil.
14  \ ------------------------------------------------------------------------------------
15     (lit) (local) 1 + 1 + \ Das ist der Teil,  den ich (zur Vermeidung von ['']) einsetze.
16  \ ------------------------------------------------------------------------------------
17     >r >r ;
18
19
20  \ Generelles Beispiel einer Colon-Definition (namens www) mit eingebauter lokaler Variabler xxx,
21  \ die aus einer schon bestehenden Variablen heraus durch Aufruf von xxx local bis zum Ende der
22  \ Colon-Definition www lokal frei verwendet werden kann. Ausserhalb des lokalen Bereichs wird
23  \ der urspruengliche Wert von xxx jeweils immer wiederhergestellt. Als Operationen, die nach dem
24  \ Aufrufen von www zur leichteren Analyse der Vorgaenge verwendet werden koennen, wird (hier) an
25  \ den markanten Stellen das Wortpaar @ . verwendet.
26
27  \ Aus diesem Beispiel folgt, dass local nicht "am Anfang der Colon-Definition" www zu stehen
28  \ braucht.
29
30  : www ( xxx -- ) xxx @ . xxx local xxx @ . 1234 xxx ! xxx @ . ; xxx @ . ( [ret] )
31
32  \ Ein Aufruf (www [ret]) dieses Analyse-Beispiels liefert:
33
34  \ 4711 = globaler Wert von xxx
35  \ 4711 = globaler Wert von xxx (noch nicht lokal veraendert)
36  \ 1234 = Wert von xxx nach lokaler Veraenderung
37  \ 4711 = wieder urspruenglicher globaler Wert von xxx
38
39
```



„Bracket-Tick“

Biester im System

Carsten Strotmann

Simulationen von unabhängigen "Lebewesen" im Rechner gibt es schon seit langem, man denke nur an Conways "Game of Life"[1] oder auch "Krieg der Kerne (Core War)"[2]. Diese Simulationen laufen in einem abgeschlossenen Programm auf einem einzelnen Rechner. Die Rechnerarchitektur ist für alle "Agenten" (Protagonisten in der Simulation) identisch, bei Core War ist auch die Programmiersprache durch die Simulation vorgegeben (RedCode).

Studenten der Hochschule Augsburg haben eine einfache Simulations-Umgebung entwickelt, bei der die Agenten auf unterschiedlichen Rechnern(-architekturen) ablaufen und auch in unterschiedlichen Programmiersprachen geschrieben sein können. Ein Agent, in Forth geschrieben, laufend auf einem MSP430-Launchpad, kann gegen einen in Java geschriebenen Agenten auf einem Apple-MacBook antreten. Unfair, aber möglich[3].

Die Simulations-Umgebung heißt "Beast-Arena" (`http://www.beast-arena.de`). Die Agenten (-Programme) verbinden sich per TCP/IP oder RS232 an den zentralen Arena-Rechner. Die Kommunikation erfolgt über ein einfaches, textbasiertes Protokoll.

Die Spielregeln

Die Protagonisten des Spiels sind die kleinen "Biester", welche in einer kleinen quadratischen Welt leben (z.B. 26 x 26 Spielfelder). Je nach Anzahl der Biester (Mitspieler) im Spiel wird die Größe der Spielwelt bestimmt. Die Spielwelt ist "endlos", d. h., der rechte Rand schließt nahtlos an den linken Rand, der obere Rand an den unteren Rand an. In der Spielwelt wird zufällig das Futter für die Biester verteilt. Das Spiel ist rundenbasiert, pro Runde kann jedes Biest eine Aktion ausführen.

Ein Biest (oder das Programm, welches den Agenten steuert) sieht die 25 Felder direkt um den eigenen aktuellen Standpunkt. Von diesem Standpunkt aus kann sich das Biest gerade oder diagonal bewegen. Bewegung kostet Energie. Zum Spielanfang wird jedes Biest mit 30 Energiepunkten ausgestattet. Fällt die Energie des Biests auf null, so stirbt es und verschwindet vom Spielfeld (siehe Abbildung 1).

Ein Zug um ein Feld in gerader Zugrichtung (nicht diagonal) verbraucht 2 Energiepunkte (links in Abbildung 2 auf der nächsten Seite).

Ein diagonaler Zug in ein Feld kostet 3 Energiepunkte (rechts in Abbildung 2 auf der nächsten Seite).

Gerade Züge über zwei Felder verbrauchen gleich 5 Punkte, diagonale Züge über zwei Felder kosten 7 Punkte. Ein einfaches "Stehenbleiben" kostet einen Energiepunkt (Abbildung 4 auf Seite 23).

Ein spezieller Zug ist das "Verstecken". Beim "Verstecken" bewegt sich das Biest nicht von der Stelle, verhindert aber die Anzeige des aktuellen Energievorrates für gegnerische Biester. Gegnerische Biester können dann nicht sehen, ob das eigene Biest kleiner oder größer ist (wichtig für Angriffe).

Biester können neue Energie bekommen, indem sie Nahrung aufnehmen. Nahrung ist das Futter, welches zu Spielbeginn zufällig auf dem Spielfeld verteilt wurde. Die Nahrungsaufnahme erhöht den Energievorrat eines Biests um 5 Punkte.

[1] https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

[2] https://de.wikipedia.org/wiki/Core_War

[3] Ich überlasse es dem Leser, zu bestimmen, in welche Richtung dieses Beispiel unfair ist :)

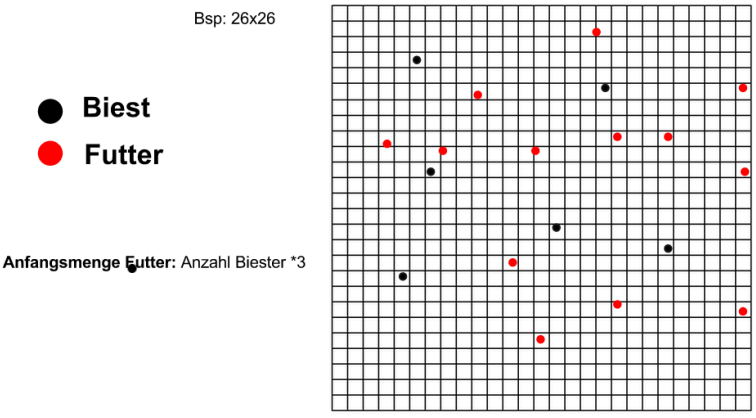


Abbildung 1: Das Spielfeld der Biester

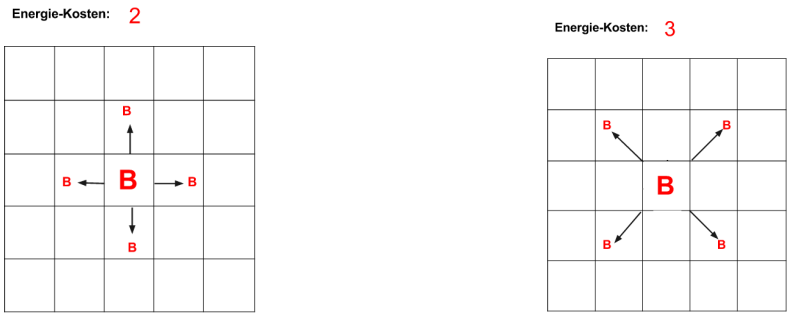


Abbildung 2: Energiekosten Regel 1 und 2

Biester können kleinere (gegnerische) Biester auffressen und sich damit die Energie des kleineren Biestes einverleiben. Ein Biest ist "kleiner", wenn es über weniger Energie als das angreifende Biest verfügt. Angegriffen wird, indem ein Biest auf ein schon von einem anderen Biest besetztes Feld zieht. Kommen zwei Biester mit gleicher Energie auf einem Feld zu stehen, gewinnt der Verteidiger (d.h. das Biest, welches das Feld als erstes betreten hat).

Generische Biester im Sichtfeld werden mit einem "<" oder ">" dargestellt. Diese Zeichen geben den relativen Energiestand des gegnerischen Biests an (größer oder kleiner als das eigene Biest).

Das Protokoll

Das Protokoll zwischen dem "Biester"-Programm und dem zentralen "Arena"-Server ist sehr einfach gehalten, wie in Abbildung 3 zu sehen ist. Die Kommunikation kann über TCP/IP oder über eine serielle Verbindung erfolgen (ggf. auch mit Protokollübersetzern). Die Kommunikation erfolgt über 7-Bit-ASCII-Zeichen.

Zu Beginn meldet sich das Biester-Programm am Server an und bekommt einen eindeutigen Namen (oder Nummer) zugewiesen. Dieser Name wird bei weiterer Kommunikation zur Identifikation benutzt. Alle Nachrichten zum und vom Server werden mit "@" abgeschlossen.

Im Spiel sendet das Biest pro Runde die gewünschte neue Position (0-24, siehe Abbildung 5 auf der gegenüberliegenden Seite). Das Biest darf bei der Rückmeldung eine bestimmte Zeit nicht überschreiten. Diese Zeit wird am Anfang des Spieles mitgeteilt und kann je nach Wettbewerb abhängig von der Rechnerklasse sein (Mikrokontroller bekommen mehr Bedenkzeit als ein 4-Core-PC).

Der Server sendet dem Biest eine Zeichenkette mit 25 Zeichen zurück, welche die "Sicht" des Biestes an der neuen Position beschreibt:

Zeichen	Bedeutung
*	Futter
<	kleineres Biest
>	größeres Biest
=	gleich starkes Biest
?	verstecktes Biest
.	leeres Feld

Ein Biester-Programm liest und interpretiert pro Runde die 25 Zeichen lange Zeichenkette, ermittelt aus den gegebenen Informationen den besten Zug und teilt diesen Zug dem Server mit.

Gewonnen hat das Biest, welches zuletzt allein auf dem Spielfeld zurückbleibt. Da jeder Zug (auch das "Nichtstun") Energie verbraucht, kann ein Spiel nicht ewig dauern.

Turnier auf dem Vintage Computer Festival

Auf dem Vintage Computer Festival im letzten Jahr (2012) wurde das Konzept der Biester-Arena vorgestellt. Auf dem VCFe in diesem Jahr (27. und 28. April 2013 in München[4]) wird es ein Test-Turnier geben, bei dem Autoren eigene Biester-Programme im Wettbewerb mit anderen Teilnehmern testen können. Die Organisatoren wollen diesen Testlauf nutzen, um noch Verbesserungsvorschläge einzuarbeiten und Fehler in den Regeln oder

[4] http://vcfe.org/D/index.html

Client	Server
"Anmeldung moeglich?@"	"Ja@" oder "Nein@"
"Anmeldung!@"	"<kennung>@", z.B. "c@" oder "Fehler@"
"Weltgroesse?@"	"<hoehe>x<breite>@"
"Spielbeginn?@"	"Wed Dec 7 17:46:54 2011@"
"Bedenkzeit?@"	Beispiel: "10s" (10 Sekunden)
andere Werte	"Fehler@"

Abbildung 3: Das Protokoll zwischen Client und Server

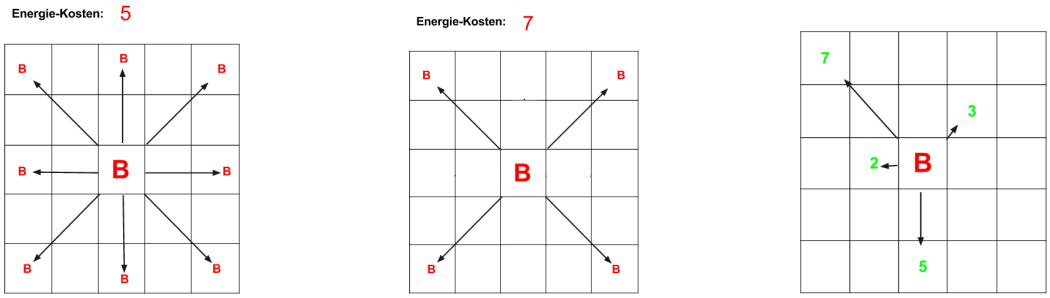


Abbildung 4: Energiekosten Regel 3, 4 und 5

dem Server-Programm zu beheben. Im kommenden Jahr (2014) soll es dann das erste richtige "Biester"-Turnier geben, auch wieder auf dem VCFe.

Da die Tagung der Forth-Gesellschaft immer kurz vor dem VCFe stattfindet, wäre es möglich, auf der Tagung ein "Team Forth" zusammenzustellen und ein Forth-Biest mit in das Rennen um den Biester-Pokal zu senden.

Auf der Webseite des Beast-Arena-Projektes (www.beast-arena.de) gibt es weitere Informationen und auch den Quellcode des Server-Programms (in Python) zum Testen und Optimieren der eigenen Biester.

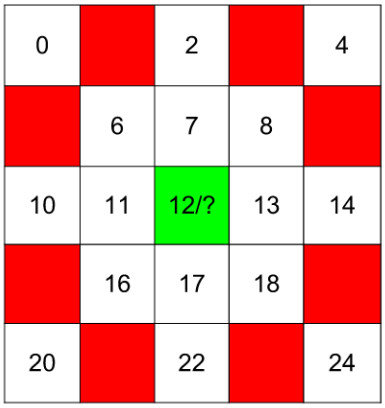


Abbildung 5: Die möglichen Bewegungen der Biester

Adaption des 4€4th–1–Wire–Codes an AmForth

Matthias Trute

Mitunter haben auch andere Väter tolle Töchter. In diesem Fall hat der MSP430 vom 4€4TH–Projekt eine kleine Portion Assemblercode und viel Forthcode, um eine ganze Welt von Peripheriebausteinen anzusprechen: Den 1–Wire–Bus.

Motivation

Neulich fiel mir beim Stöbern in diversen RSS–Feeds ein Checkin im 4€4TH–Projekt auf:

```
Aktuelles Verzeichnis: /
Revision:   2517
Autor:  michael
Letzte Änderung:    Revision 2517 - 2012-12-14 11:45:52
Logeintrag: camelforth dallas 1-wire von Brad angefuegt
  Neue Dateien: 4e4th/onewire430-1.zip
  4e4th/4e4th-onewire.a43
```

Die Kontaktaufnahme mit den beiden 4€4th–Maintainern (Dirk und Michael) verlief unproblematisch und sie gaben mir viele interessante Informationen. So zum Beispiel, dass der Code urprünglich auf eine Bitte aus Australien von Brad (in Kanada) erstellt wurde und dass der Code unter der GNU–Lizenz speziell für das 4€4TH–Projekt veröffentlicht wurde.

Mein AmForth hat eine lange, wenngleich informelle, Wunschliste, was so alles damit funktionieren soll. Der 1–Wire–Bus stand dabei recht weit oben. So ganz in reinem Forth erschien der nicht machbar (siehe unten) und die im Netz auffindbaren Codeschnipsel waren alle irgendwie nicht motivierend genug, es anzugehen. Der MSP430–Code sah hingegen sehr interessant aus...

Grundlagen

1–Wire ist die Bezeichnung für einen sehr genügsamen Peripheriebus, der von der Firma Dallas (jetzt MAXIM) erfunden wurde. Mit ihm lassen sich mehrere Bausteine mit wenigen Leitungen mit einem Controller verbinden. Häufig findet man ihn bei Temperatursensoren, es gibt auch andere Dinge, die das Verfahren nutzen. Damit man die Geräte voneinander unterscheiden kann, haben sie einen 8–stelligen garantiert eindeutigen Code, der sie identifiziert: die ROM-ID. Diese kann durch einen Busscan abgefragt werden und dient dazu, mit einzelnen Geräten gezielt zu kommunizieren.

Bernd Paysan hat den 1–wire–Bus vor einiger Zeit hier bereits vorgestellt. Er hat dabei die PC–Seite stärker beleuchtet. Bei ihm gibt es demzufolge ein nettes GUI–Programm, das mit der Maus funktioniert. Die Hardwareseite läuft unter der Rubrik „vorhanden und funktioniert“. Für Mikrocontroller hingegen sind die Bits und Bytes wichtig. Da ist der 1–wire–Bus nicht ganz so hübsch. Er erfordert zwar nur zwei Verbindungen für Daten/Energie und Masse (eine separate Spannungsversorgung, also 3 Verbindungen, sind besser), aber was darauf abgeht, muss bis auf wenige Mikrosekunden genau getimed werden.

1–Wire ist ein asynchroner serieller Bus. Er arbeitet mit einer strikten hostorientierten Kommunikation. Der Host initiiert den Datentransfer und wertet die Antworten der Clients anhand des Timings aus. Zwischen zwei Datentransfers muss genügend Zeit verstreichen, damit über die Datenleitung auch genug Energie an die Clients geliefert werden kann. Dies wird erreicht, indem die Datenleitung im Ruhezustand automatisch über einen Pullup–Widerstand auf 1 (Highpegel, meist 3–5 Volt) gestellt wird. Die Clients laden damit einen kleinen eigenen Kondensator auf, der die lokale Energieversorgung während der Kommunikationsphasen übernimmt.

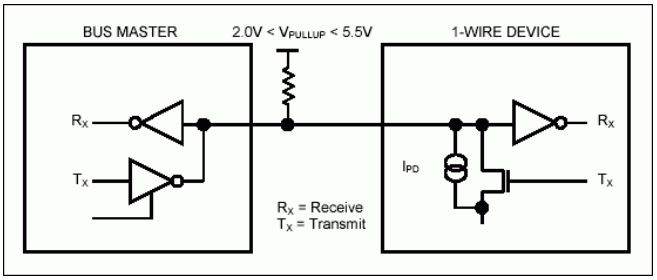


Abbildung 1: 1–Wire–Bus–Interface–Schaltung

Der Host signalisiert über ein LOW auf der Datenleitung den Start einer Kommunikation. Wird die Signalleitung nur kurz (wenige Mikrosekunden) auf LOW gezogen, wird ein Bit übertragen. Je nach Dauer des LOW–Pulses wird zwischen 0 (lange LOW) und 1 (kurz LOW) unterschieden. Damit die Clients auch Daten senden können, können sie ihrerseits den Buspegel in einem engen Zeitfenster auf LOW ziehen, was der Host dann als 1 erkennen kann. Für eine Null wird der Bus nicht auf LOW gezogen, bleibt also bei HIGH. Um zu prüfen, ob überhaupt ein 1–wire–Device angeschlossen ist, ist ein RESET/PRESENCE–Test vorgesehen. Dabei geht der Bus für eine vergleichsweise lange Zeit auf LOW. Wenn dann innerhalb einiger Mikrosekunden nach der Freigabe des Bussignals der Pegel wieder auf LOW geht, ist wenigstens ein Device angeschlossen und reagiert.

Das Timing basiert auf sogenannten Slots von 60 Mikrosekunden Länge, innerhalb derer jeweils ein Bit übertragen wird (Lesen und Schreiben sind sich sehr ähnlich). RESET sind 8 (oder mehr) derartige Slots am Stück. Zwischen zwei Slots ist das Timing erheblich entspannter.

Ein Mikrocontroller muss diese Abläufe hinreichend genau abbilden, sofern er nicht einen dedizierten Hilfschip benutzt, der das als Co-Prozessor übernimmt (es gibt da nette I2C Bausteine). Da eine Taktfrequenz von 1MHz

genau einer Mikrosekunde entspricht, und ein (moderner) Mikrocontroller einen CPU–Befehl pro Taktzyklus ausführt, wird deutlich, dass das Timing nicht von den eher langsamen Indirect–Threaded–Forth–VMs erzielt werden kann. Ein klein wenig Assemblercode ist unumgänglich. Aber es sollte wirklich nur wenig sein.

Implementierung

Der Autor von Camelforth, Bradford J. Rodriguez hat Ende 2012 ein kleines Modul für den MSP430 geschrieben und dem 4€4TH–Projekt gespendet. Sein Code hat nur zwei in Assembler geschriebene Worte und alles andere in Forthcode umgesetzt. Die beiden Assemblerworte sind zudem recht kurz, so dass ich ungeachtet der mir unbekannten Assemblersyntax und in Unkenntnis der Prozessordetails des MSP430 den Versuch wagte, eine AVR–Atmega–Variante für AmForth zu erstellen.

Die erste Herausforderung war das präzise Timing. Das hat erst mal nichts mit dem 1–wire–Bus zu tun. Bislang konnte AmForth Zeiten so ab einer Millisekunde hinreichend korrekt bearbeiten. Jetzt stiegen die Anforderungen in den Mikrosekundenbereich, wobei das kürzeste Intervall sportliche 6 Mikrosekunden umfasst. Solche Aufgaben können Microcontroller im Wesentlichen auf zweierlei Wegen machen: CPU–Zyklen verbraten oder ein Timermodul bemühen. Ersteres ist einfach, Letzteres bindet Hardwareressourcen, die vielleicht anderweitig besser genutzt werden können. Für das 1–wire–Timing ist prinzipiell noch der serielle USART–Baustein geeignet, die sind jedoch noch knapper. Also Variante 1.

4€4TH ist auf das Launchpad–Board von Texas Instruments zugeschnitten. Das ist genau ein Prozessortyp mit genau einer festgelegten Taktfrequenz (8MHz). AmForth ist mittlerweile für mehr als 100 Atmegavarianten mit Taktfrequenzen zwischen 1 und 20MHz (oder auch mehr) gedacht. Damit musste ein Mechanismus gefunden werden, der aus den in der Spezifikation des 1–Wire–Busses angegebenen Mikrosekunden die genau passende Anzahl CPU–Zyklen macht.

Im Netz finden sich einige Beispiele, es gab aber keines, das alle Zeitspannen bei allen denkbaren Taktfrequenzen überdeckte. Der aktuelle Code basiert auf Beispielen von www.mikrocontroller.net, die allerdings stark überarbeitet wurden.

```
.macro   delay
 .set cycles = ( ( @0 * F_CPU ) / 1000000 )
 .if (cycles > ( 256 * 255 * 4 + 2))
  .error "MACRO delay - too many cycles to burn"
 .else
   .if (cycles > 6)
    .set  loop_cycles = (cycles / 4)
    ldi   zl,low(loop_cycles)
    ldi   zh,high(loop_cycles)
delay_loop:
    sbiw  Z, 1
    brne  delay_loop
    .set  cycles = (cycles - (loop_cycles * 4))
   .endif
   .if (cycles > 0)
     .if   (cycles & 4)
       rjmp  pc+1
       rjmp  pc+1
     .endif
     .if   (cycles & 2)
       rjmp  pc+1
     .endif
     .if   (cycles & 1)
       nop
     .endif
   .endif
 .endif
.endmacro
```

Zum Vergleich der MSP430–Code für die gleiche Kernaufgabe, zugeschnitten auf die Verhältnisse des Launchpads:

```
DELAY   MACRO   usec
        LOCAL   loop
        MOV     #(((usec*MCLK)-2)/3),X  ; 2
loop:   SUB     #1,X                    ; 1n
        JNZ     loop                    ; 2n
        ENDM
```

Der Makrocode für den Atmega macht mehr als der entsprechende Code für den MSP430. So prüft er zur Compilezeit, ob das angegebene Zeitintervall überhaupt erreicht werden kann. Zudem wird deutlich, dass der Atmega zwar 16–Bit–Operationen beherrscht (SBIW ist Subtract Immediate Word), aber trotzdem immer noch eine 8–Bit–CPU bleibt. Der MSP430 ist hingegen ein 16–Bit–Prozessor. Im Quellcode sehen beide Makros wie ein Maschinenbefehl aus, werden aber in beiden Fällen zu einem Loop mit vorab berechneter Durchlaufzahl expandiert.

```
...
delay 6
...
```

Das war der einfache Teil der Portierung. Jetzt ging es darum, den eigentlichen 1–wire–Code umzusetzen. Erster Befehl ist der OWRESET. Hierfür muss der Bus 500 Mikrosekunden auf LOW gezogen werden, dann auf 1 gesetzt und nach weiteren 60 Mikrosekunden der Zustand des Busses eingelesen werden. Wenn dann eine Null gelesen wird, ist ein 1–wire–Device vorhanden und ansprechbar. Anderenfalls bleibt der Bus auf 1 und nichts funktioniert.

Atmegas können wie der MSP430 ihre Pins dynamisch zwischen Ein- und Ausgabe umschalten. Die Wechselwirkungen zwischen der Datenrichtung, den Werten in den Ein- und Ausgaberegistern und den resultierenden Pegeln auf den Datenleitungen können sehr verwirrend werden. Dabei sind auch neutrale, sog. Tristate Zustände möglich. Die sind hier aber störend, so dass wirklich jedes Bit zu jedem Zeitpunkt exakt bestimmt werden muss.

Wenn dann noch ein gewisses Misstrauen gegenüber der Testverkabelung (Steckbrett mit Arduino daneben) und

den Timings besteht, wird die Fehlersuche zu einem Stochern im Nebel. Ein Oszilloskop hätte helfen können, war aber nicht verfügbar. Irgendwann lieferte OWRESET dann doch eine 0 wenn kein DS1820 (ein Temperatursensor, der aussieht wie ein Transistor) angeschlossen war und eine -1 wenn doch. Und das sogar reproduzierbar.

Das zweite Wort, OWSLOT, ist für die Bittransfers zuständig. Brads Code ist eine ausgesprochen intelligente Umsetzung der Vorgaben aus dem Datenblatt, da er die getrennt dokumentierten Vorgänge „1 schreiben“, „0 schreiben“ und „Bit einlesen“ in einem Modul zusammengefasst hat. Alle Codebeispiele, die sich so im Netz finden lassen, implementieren diese drei Aufgaben auch in getrenntem Code. Nachdem der OWRESET die grundlegenden Abläufe bei der Ansteuerung des Buspins geklärt hatte, sollte es jetzt eher einfach sein. War es aber nicht, da die Pegel wohl nicht immer so sauber gesteuert wurden, wie es erforderlich ist.

Das Debugging ohne Oszilloskop war der reine Blindflug. Vom OWRESET her war sichergestellt, dass die Hardware korrekt funktioniert. Es war aber unbekannt, warum immer nur Nullen oder immer nur Einsen als Ergebnis kamen. Der Fehler konnte beim Bitsenden oder beim Bitlesen stecken, oder bei beiden. Der Code war schließlich immer fast der gleiche. Ziel war, dass die ROM-IDs sinnvoll aussahen. Die stehen nicht auf den Bausteinen drauf, also war irgendwas, was nach einer Mischung von 0 und 1 aussah, erst mal gut. Ob es auch stimmte, war erst mal nicht klar.

Beim OWRESET war der Debugzyklus „Assemblerfile editieren“, „Hex File erzeugen und auf den Controller flashen“ und „OWRESET .“ ausführen. Beim OWSLOT kam jetzt noch ein „#include 1wire.frt“ und „owshowid“ hinzu. Die Datei „1wire.frt“ hat den ganzen highlevel Code für den byteweisen Transfer und auch Kommandos wie „Bus scannen und ROM ID ausgeben“ (owshowid). An dieser Stelle hat sich die amforth-shell bewährt. Zum einen weil sie den Reconnect zum Controller nach dem Neuflashen ohne Weiteres gemacht hat, zum anderen aber wegen der (controllerunabhängigen) Kommandohistorie. Zweimal Cursor–UP und Enter drücken geht sehr viel schneller als alles andere.

Jetzt kam Brads Arbeit so richtig zur Wirkung: Ich hatte die Forth–Datei vom 4€4TH übernommen und zunächst um alles verkürzt, was den Testzyklus verlangsamt hätte. Als dann irgendwann alle Bugs im OWSLOT beseitigt waren, habe ich die restlichen Teile wieder aktiviert und es funktionierte sofort alles. Ich konnte den Bus auf alle angeschlossenen Geräte scannen (der Algorithmus dafür ist abenteuerlich) und von einem DS1820 die Temperatur einlesen. Alles, was man so haben will. Ein zweiter Sensor parallel zum ersten funktionierte auf Anhieb. Ebenso war die ermittelte Temperatur plausibel und damit die gesamte Codekette korrekt.

Anpassungen

So ganz ist Brads Code dann aber doch nicht unverändert geblieben. Zum einen hat er alle Schlüsselworte standardkonform in Großbuchstaben geschrieben. AmForth ist aber nur unter Protest bereit, Großbuchstaben zu verstehen, es bevorzugt die leiseren Töne der Kleinbuchstaben.

Zum anderen ist die Wahl der Wortbezeichner nicht immer glücklich. So gibt es CONVERT, was bei den alten Hasen sicher andere Assoziationen als die Umrechung von Temperaturwerten weckt. So haben jetzt alle relevanten Worte ein Präfix „ow“, das ihre Zuständigkeit für den 1–wire–Bus deutlich macht.

Ausblick

Brads Kerncode wird sicher Bestand haben. Ob die Routinen für die einzelnen Gerätetypen so bleiben werden, wird sich zeigen. So sind elegantere Abläufe als

```
hex create sensor2
  28 , 4C , 75 , CC , 2 , 0 , 0 , CD ,
decimal sensor2 owconvert 500 ms
  sensor2 readtemp temp>pad pad count type
```

vorstellbar.

Referenzen

1. `http://www.4e4th.eu/`
2. `http://www.camelforth.com/`
3. `http://amforth.sourceforge.net/`
4. OneWire Bigforth, VD 2008-01

Bildnachweis

Abb. 1 aus: TUTORIAL 214, Using a UART to Implement a 1–Wire Bus Master, Sep 10, 2002, maxim integrated. `http://www.maximintegrated.com/app-notes/index.mvp/id/214`

Mecrisp - Forth für den MSP430

Matthias Koch

Mecrisp *ist eine native Implementation von Forth für die MSP430-Architektur und ist nun nach etwas mehr als anderthalb Jahren Entwicklungszeit stabil! Spezialität von Mecrisp ist die Verwaltung von Flash-Speicher, so dass die Pointer, die für das Dictionary nötig sind, nicht gesichert werden müssen und nach einem Reset an beliebiger Stelle stets wiederhergestellt werden können. Initialisierte Variablen stehen zur Verfügung und werden von weiteren Spezialitäten ergänzt: Forth-Definitionen können direkt als Interruptvektoren verwendet werden und beim Start kann automatisch eine selbstdefinierte Hardwareinitialisierung oder ein Anwenderprogramm aufgerufen werden.*

Die Features

Einfache Rechnungen, Logikoperationen und Kontrollstrukturen werden nach vorhergehender Konstantenfaltung direkt opcodiert - was die Konstantenfaltung ist habe ich im Anhang beschrieben. Die Routinen im Kern sind für den MSP430 maßgeschneidert, der von Hand optimierte Assembler-Quelltext von Mecrisp dürfte für deutschsprachige Entwickler eine Fundgrube sein. Glücklicherweise macht Forth es dem Entwickler leicht, Fehler schnell zu finden - so konnten im Laufe des Entstehens und Erprobens immer mehr Funktionen in den schließlich nur 9 kB großen Kern aufgenommen werden. Von Anfang an zählten separate Multiplikations- und Divisionsroutinen für einfach- und doppeltlange Zahlen dazu, neben den üblichen Routinen für doppeltlange Zahlen ist auch Unterstützung für Fixkommazahlen im s15.16 Format sowie deren komfortable Ein- und Ausgabe im Kern enthalten.

```
3,14159 2,0 f* f. 6,2831726074218750  ok.
```

Besonders praktisch ist dies bei Werten aus dem Analog-Digital-Wandler:

```
: vcc.
   ."  Vcc is " 0  11 analog  204,6 f/ f. ." V " ;
```

Routinen zur Nutzung der besonders eleganten Portzugriffe beim MSP430 sind selbstverständlich implementiert. In der letzten Zeit ist die `CASE`-Struktur hinzugekommen und der Codegenerator (siehe Beispiel weiter unten) ist noch einmal gründlich überarbeitet worden. Praktisch besonders für Einsteiger dürften auch die Wörter zur Verwendung des internen Analog-Digital-Wandlers sein.

Und das Beiwerk

Nachladbar und im Forth-Quelltext beigelegt sind ein Assembler, ein Disassembler, Routinen für Sinus und Cosinus, Konstantendefinitionen für häufig verwendete Ports sowie einige kleine Beispiele. Um die entwickelten Programme leicht weiterverwenden zu können, steht `hexdump` zur Verfügung. `Hexdump` ist ein Forth-Programm, das direkt im MSP430G2553 läuft - das Intel-Hex-Image wird über das Terminal ausgegeben und kann anschließend in eine Textdatei kopiert werden.

Zum Codegenerator des Mecrisp

Zum Hineinschnuppern sei hier einmal ein Disassembler-Listing der Funktion gezeigt, die Zufallszahlen aus dem Rauschen des internen Temperatursensors generiert:

```
: random ( -- u )
 ( Generiert Zufallszahlen mit dem Rauschen )
 ( vom Temperatursensor am ADC )
 ( Random numbers with noise of temperature )
 ( sensor on ADC )
   0
   16 0 do
     shl
    10 analog 1 and
    xor
  loop
;
```

Und mit `SEE` kann man sehen, was daraus wurde:

```
see random
89AC: 1206  push.w r6
89AE: 1205  push.w r5
89B0: 4305  mov.w #0h, r5
89B2: 4036  mov.w #10h, r6
89B4: 0010
89B6: 8324  sub.w #2h, r4
89B8: 4384  mov.w #0h, 0h(r4)
89BA: 0000
89BC: 54A4  add.w @r4, 0h(r4)
89BE: 0000
89C0: 8324  sub.w #2h, r4
89C2: 40B4  mov.w #Ah, 0h(r4)
89C4: 000A
89C6: 0000
89C8: 12B0  call.w #FE32h  --> analog
89CA: FE32
89CC: F394  and.w #1h, 0h(r4)
89CE: 0000
89D0: E4B4  xor.w @r4+, 0h(r4)
89D2: 0000
89D4: 5315  add.w #1h, r5
89D6: 9506  cmp.w r5, r6
89D8: 23F1  jnz 89BC
89DA: 4135  mov.w @r1+, r5
89DC: 4136  mov.w @r1+, r6
89DE: 4130  mov.w @r1+, r0
 ok.
```

Mecrisp hat die Forthquelle in ein Stück Maschinencode assembliert. Es macht das tatsächlich selbst!

Neugierig geworden?

Fertige Hex-Dateien liegen für den MSP430G2553 und den MSP430F2274 bereit bei:
`http://mecrisp.sourceforge.net/`

Zugabe! Mecrimemu und Ledcomm

Für die, die noch keinen MSP430 in der Bastelkiste haben, ist der Quelltext für Mecrimemu enthalten, ein Emulator, mit dem Mecrisp auch auf einem Linux-PC ausprobiert werden kann und der bei der Entwicklung von Mecrisp eine große Rolle gespielt hat.

Für die, die gern alternative Terminals ausprobieren oder in Forth selbst implementieren möchten, gibt es noch eine besondere, 10 kB große Variante für den MSP430G2553, bei der die Terminalroutinen vektorisiert sind und wo Ledcomm[1], eine serielle Schnittstelle, die trickreich über eine einzelne Leuchtdiode funktioniert, bereits im Kern enthalten ist. Ein Chat-Beispielprogramm für Ledcomm liegt bei! Nach erfolgreichen Experimenten lässt sich der Quelltext von Mecrisp leicht auf die gewählte Lösung, d. h. auch für andere Mitglieder der MSP430-Familie anpassen.

Rückmeldungen

Über Fragen und Ideen freue ich mich sehr !
Matthias Koch
m-atthias@users.sf.net

PS: Mecrisp ist ein Wortspiel - es kommt von MSP und *écris*, im Französischen „Du schreibst“, im Sinne von: „Du beschreibst den MSP430“.

Links

`http://mecrisp.sourceforge.net/`

Anhang: Konstantenfaltung

Die Konstantenfaltung kümmert sich darum, Operationen mit Werten, die schon beim Kompilieren feststehen, gleich durchzuführen und anschließend nur das Ergebnis zu kompilieren. Ein Beispiel:

```
$20 constant P1IN
$21 constant P1OUT
$22 constant P1DIR
$27 constant P1REN

%1 constant led_red
%10 constant led_green

: led-init led_green led_red or P1DIR c! ;
```

Konstanten sind in Mecrisp automatisch faltbar - und die Oder-Verknüpfung wirkt auf bereits bekannte Werte. Die Konstantenfaltung macht daraus intern

```
: led-init 3 $22 c! ;
```

Weil `c!` opcodierbar ist, wird schließlich der Befehl `mov.b #3, &022h` ins Dictionary geschrieben.

Die Konstantenfaltung funktioniert so, dass ein „Konstantenfüllstandszeiger“ auf den Stack gerichtet wird, wenn beim Kompilieren Zahlen bearbeitet werden. Wörter können als faltbar markiert werden, beispielsweise sind `+`, `swap`, `*` oder `xor` faltbar, wenn zwei Konstanten bereitliegen, wobei `0=`, `drop` und `dup` mit einer Konstanten faltbar sind. Zur Faltung selbst wird der Füllstand geprüft und das entsprechende Wort bei genügend vielen vorhandenen Konstanten auf dem Stack ausgeführt. Die Ergebnisse liegen danach wieder auf dem Stack und können weiterverarbeitet werden. Constant-Definitionen sind null-faltbar, d. h., sie benötigen keine Zahlen auf dem Stack, legen aber faltbare Ergebnisse bereit. Die opcodierbaren Wörter wie `@` und `c!` sind nicht überfaltbar, können jedoch Konstanten als Operanden benutzen. Es gibt auch Mischformen, z.B. kann `xor` mit zwei Konstanten gefaltet und mit einer Konstanten opcodiert werden. Sollen Wörter ohne Faltbarkeitsflag kompiliert werden, werden zuvor alle Zahlen bis zum Konstantenfüllstandszeiger einkompiliert. All dies funktioniert transparent für den Forth-Programmierer, so dass dieser sich um die Konstantenfaltung nicht zu kümmern braucht. Es ist jedoch möglich, selbstdefinierte Worte als mit entsprechender Konstantenzahl faltbar zu markieren.

[1] Zu Ledcomm wird es hoffentlich schon bald einen gesonderten Beitrag geben - das würde den Rahmen hier sprengen.

Die PC-Tastatur–LEDs als Lichtorgel

Fred Behringer

Ich gehe von einem PC–AT mit seinen in die Tastatur eingebauten drei LEDs aus und lasse diese per Programm unabhängig voneinander in einem Blinkkonzert als Lichtorgel spielen. (Man frage mich nicht, wozu. Ganz einfach, weil es ging. Mein Spaß betraf den Weg, das Ziel war mir schlicht egal.)

Auch für ZF - und anderswo

Jede Tastatur, die mit der IBM-AT-Tastatur kompatibel ist, sollte mit meinem Programm ebenfalls funktionieren. Es werden keine Code-Definitionen benötigt, die nicht im Turbo-Forth-Basis-System auch schon vorhanden wären. Also nicht unbedingt ANS-Forth, aber alles reines High-Level-(Turbo-)Forth. Damit das für Turbo-Forth geschriebene Programm auch in ZF verwendet werden kann, ergänze ich allerdings die drei hier benötigten und in ZF nicht vorhandenen Turbo-Forth-Worte schnell noch. In Forth geht sowas ja spielend leicht: Einfach aus dem Turbo-Forth-Quelltext übernehmen - und sich darüber freuen, dass Mehrfach-Definitionen (unter verschiedenen Namen) nicht stören!

Schnell noch Anleitung fürs Aufrufen in ZF

Ich gehe davon aus, dass mein Programm (im selben Verzeichnis wie Turbo-Forth und ZF) in eine Datei mit Namen listing.txt gelegt wurde.

In Turbo-Forth:	`Forth include listing.txt`
in ZF:	`ZF [taste] fload listing.txt`

und dann aufrufen über `licht-spiel` .

Wo steht „alles“ über den Keyboard-Controller?

Bei Google kaum. Ich bin sehr froh, dass ich mich noch an den prallgefüllten Leitz-Ordner mit Kopien der vollständigen Beschreibung des IBM-ATs in meinem Bücherschrank erinnern konnte. Zwar nur das ursprüngliche System mit dem 80286 als CPU, aber was den Keyboard-Controller (den 8042 von Intel) betrifft, konnte ich als Arbeitshypothese davon ausgehen, dass sich das, was mich für den vorliegenden Artikel interessiert, auch beim 80386, beim 80486, beim Pentium und bei noch höheren Systemen nicht geändert hat (8042, oder Emulation davon, gehorcht den Spezifikationen). Von Laptops, die gar keine LEDs eingebaut haben, spreche ich hier nicht - und außerdem scheint es eine LED zur Anzeige der eventuellen Sperrung der Insert-Funktion (InsLock) bei IBM-Kompatiblen nie gegeben zu haben.

Was interessiert mich am Kbd-Controller?

Der Controller hat ein Status-Register (Port 64h), einen Output-Puffer und einen Input-Puffer. Die letzteren beiden verteilen ihre Aufgaben zwischen Port 60h und Port 64h. Die Dinge, die vom Kbd-Controller in Bewegung gesetzt werden, sind zu komplex, als dass ich mich mit ihnen hier auseinandersetzen möchte. Mich interessiert nur das Byte aus Port 60h als Ganzes. Es kann gelesen und auch beschrieben werden. Sowohl in Turbo-Forth als auch in ZF geschieht das über pc@ und pc! (was für ein *Zufall*, wenn man an die störende Unterscheidung zwischen `lc@` und `lc!` in Turbo-Forth und `c@l` und `c!l` in ZF denkt!).

Einen Tastendruck simulieren?

Interessant, was man in den unter Google erreichbaren Foren darüber lesen kann! In Heft 4/2012 der Vierten Dimension (im Nachtrag zum dortigen Artikel) war ich schon fast so weit, dass ich Tastatur-LEDs an- und wieder ausschalten konnte - aber nur fast. Denn wenn ich eine LED (per Programm) anschaltete, auf sie (immer noch per Programm) eine genügend lange Verzögerung einwirken ließ und sie schließlich (abermals per Programm) wieder ausschaltete, blieb sie, sehr zu meiner Überraschung, angeschaltet und verlöschte erst dann, wenn ich danach (irgendwann danach!) eine Taste (irgendeine Taste, auch Shift oder Alt) drückte. Zwischendurch konnte ich (über eine Colon-Definition oder direkt über die Tastatur) so ziemlich alle Forth-Befehle aufrufen, ohne dass sich an dieser Situation etwas änderte. Ein Auslesen der Stelle 0000:0417h im BIOS-Datenbereich ergab, dass deren Inhalt (durch das im Listing beschriebene `scroll-off` oder `num-off` oder `caps-off`) zwar wieder auf denjenigen Wert zurückgesetzt worden war, der einem Löschen der betreffenden LED entsprach, dass es dem System aber (bis zum Drücken einer Taste) nicht gelungen war, die LED selbst wieder zum Verlöschen zu bringen. Damit konnte dann natürlich keine automatisch ablaufende Lichtorgel programmiert werden. Die Vermutung lag nahe, dass der Keyboard-Controller ein inhibierendes Fehlerbit gesetzt hatte, das erst *regesettet* werden musste.

Ohne Bit-Pfriemelei zur Tastendruck-Simulation

Ohne große Überlegungen über das tatsächliche Geschehen am Tastatur-Kontroller anstrengen zu wollen, habe ich dann (immer wieder) zwei Lese-Messungen am Port 60h vorgenommen: Einmal vor dem zusätzlichen Tastendruck und einmal danach. *Danach* ergab sich immer der Binärwert 11100. Es lag nahe, diesen Wert als *Normalwert* (Tastatur-Kontroller liefert kein Inhibitionsbit) zu interpretieren. Es lag ebenso nahe anzunehmen, dass (in der Folge von Forth-Worten) ein Beschreiben

des Ports 60h mit eben diesem Wert (11100) ein sofortiges *Zurücksetzen* des Tastatur-Kontrollers (auf den Normalzustand) zur Folge haben würde. Studieren geht über Probieren: Es hatte! Konkret gesagt, war ich von `scroll-off` ausgegangen und habe folgende Folge von Forth-Worten eingegeben:

```
hex scroll-on delay scroll-off 1c 60 pc!
```

Und es wirkte! Die Scroll-LED wurde eingeschaltet, brannte eine Delay-Zeit lang und wurde dann wieder ausgeschaltet. Dabei ist `delay` eine Verzögerungsschleife (siehe Listing) und `1c` ist die Hex-Entsprechung des Binärwertes 11100. Lässt man `1c 60 pc!` weg, dann steht man vor der Misere mit dem zusätzlich benötigten Tastendruck (um die Scroll-LED wieder zum Verlöschen zu bringen). Lässt man dagegen `delay` weg, dann kann man ein kurzes Flackern (LED aus - LED an - LED aus) beobachten. (Das eigentliche An- und Ausschalten der LEDs spielt sich wohl in der Größenordnung von zwei Millisekunden ab.) Das Ganze funktioniert, ausgehend von `scroll-on`, (natürlich) auch z. B. mit

```
hex scroll-off delay scroll-on 1c 60 pc!
```

Damit war die hier benötigte programmgesteuerte Tastendruck-Emulation geboren! Dass die (drei) LEDs unabhängig voneinander gesteuert werden können, war rein logisch selbstverständlich.

Und hier ein Lichtorgel-Beispiel

Die Parameter (im Listing 7000, 700 und 1000) können (weitgehend) beliebig variiert werden. Die Zeitverzögerungen habe ich diesmal (Turbo-Forth und ZF sind nur 16 Bit breit) mit einer doppelten DO-LOOP aufgebaut. Zunächst ein paar Forth-Worte, die das Programm auch für ZF verwendbar machen. (Es war müßig zu fragen, was um Himmels willen in ZF dem stop? aus Turbo-Forth entspricht! Ich habe das Wort für eine eventuelle Verwendung in ZF einfach ein zweites Mal ins Listing gepackt.) Dann eine Rekapitulation der Worte für scroll-on etc. aus meinem Artikel im Heft 4/2012, dann die Verzögerung (delay) und schließlich das Beispiel einer Lichtorgel (die über licht-spiel beliebig oft aufgerufen werden kann). Verlassen wird die Blink-Schleife durch Drücken der Return-Taste, wiederaufgenommen durch Drücken irgendeiner anderen Taste (z.B. [Space]), endgültig verlassen durch hintereinander ausgeführtes zweimaliges Drücken der Return-Taste.

Registry und Co?

Ich beziehe mich stark auf den BIOS-Datenbereich. Es ist sicher sehr interessant, darüber zu diskutieren, wie man XP und andere Windows-Systeme dazu bringen kann, die LEDs über Eintragungen in die Registry nach eigenem Gutdünken zu steuern (siehe Google). Wenn (im Protected-Mode) der BIOS-Datenbereich gar nicht berücksichtigt wird (?), kann man mit den hier geschilderten einfachsten Mitteln auch keine Blink-Konzert-Veranstaltung erwarten.

Listings

LED-Lichtorgel: listings.txt

```
1   \ LED-Lichtorgel (Erklärungen siehe Textteil)
2   \ ----------------------------------------
3
4
5
6   hex
7
8
9
10  \ ============================================================================
11  \ Fuer ZF hinzugefuegt: Drei in ZF so nicht vorhandene Turbo-Forth-Worte
12  \ ============================================================================
13
14
15
16  code c@l ( seg adr -- c )    \ Holt Byte c von der Langadresse seg:adr
17   ds cx mov  bx pop  ds pop  ax ax xor
18   0 [bx] al mov   cx ds mov  1push  end-code
19
20
21
22  code c!l ( c seg adr -- )    \ Legt Byte c  an die Langadresse seg:adr
23   ds cx mov  bx pop  ds pop     ax pop
24   al 0 [bx] mov   cx ds mov   next  end-code
```

```
25
26
27
28  : stop?  ( -- fl )           \ fl=true, wenn Return-Taste gedrueckt wurde
29    false key? if drop key drop key 0d = then ;
30
31
32
33  \ ============================================================
34  \ Verzoegerungsschleifen (willkuerlich) - Bitte experimentieren!
35  \ ============================================================
36
37
38
39  : delay-scroll ( -- ) 7000 2 do j drop 10 2 do i drop loop loop ;
40  : delay-num    ( -- )  700 2 do j drop 20 2 do i drop loop loop ;
41  : delay-caps   ( -- ) 1000 2 do j drop 40 2 do i drop loop loop ;
42
43
44
45  \ ==========================================================
46  \ Hilfsprogramme fuer die Lichtorgel (an- und ausschalten)
47  \ ==========================================================
48
49
50
51  : switch-on  ( led -- ) 0 417 c@l or  0 417 c!l ;      \ led =
52  : switch-off ( led -- ) 0 417 c@l and 0 417 c!l ;      \ 10/20/40
53
54
55
56  : scroll-on  10 switch-on  ;                           \ 10 =
57  : scroll-off ef switch-off ;                           \ scroll
58  : num-on     20 switch-on  ;                           \ 20 =
59  : num-off    df switch-off ;                           \ num
60  : caps-on    40 switch-on  ;                           \ 40 =
61  : caps-off   bf switch-off ;                           \ caps
62
63
64
65  \ ========================
66  \ Ein Lichtorgel-Beispiel
67  \ ========================
68
69
70
71  : licht-spiel ( -- )
72    cr     ."             Stop:         Returntaste druecken!"
73    cr     ."        Dann Exit: Nochmal Returntaste druecken!"
74    cr     ." Ansonsten Weiter:         Space-Taste druecken!"
75    begin
76      scroll-on  delay-scroll scroll-off delay-num  1c 60 pc!
77         num-on  delay-num       num-off delay-caps 1c 60 pc!
78        caps-on  delay-caps     caps-off            1c 60 pc!
79      stop?
80    until ;
81
```

DIY assembly approach
Instruktionen selbst assemblieren

M.Kalus

Enoch führte im Februar 2013 im Amforth–eMail-Rundbrief eine Diskussion über die Notation bit-manipulierender Worte. Und zeigte eine elegante Methode auf, mit der er einzelnen Maschinen–Instruktionen einen Namen geben kann, damit sie vom High–Level–Forth ausgeführt werden können. Anlass war eine Anwendung, für die Setzen oder Löschen von Portbits un-unterbrechbar (atomic) sein mussten. Er veröffentlichte dazu ein fiktives Beispiel, um zu illustrieren, um was es ging, und wie man das macht.

Angenommen, wir hätten ein Relay am Pin `PORTA.0` angeschlossen.

```
PORTA 0 port:hi! relay_on
PORTA 0 port:lo! relay_off
```

Natürlich könnte man das auch über die *bitnames* (der amforth library) bewerkstelligen. Doch mit dem folgenden Code wird das durch eine einzige Instruktion erreicht.

```
: _bitio
    dup $1F U> if &-9 throw then
    over $7 U> if &-9 throw then
;

: port:hi ( portadr bitno -- ) \ SBI
    swap $20 - _bitio
    3 lshift or $9A00 or
    code , end-code
;

: port:lo ( portadr bitno -- ) \ CBI
    swap $20 - _bitio
    3 lshift or $9800 or
    code , end-code
;
```

Nun, was geschieht dort in `port:hi` und `port:lo` eigentlich? Die Phrase `code , end-code` compiliert einen Namen ins Forth Wörterbuch, mit dem man zur Laufzeit die Instruktionen zwischen `CODE` und `END-CODE` ausführt. Im Fall des `port:hi` ist das nur eine einzige Instruktion, nämlich `SBI`. Der Instruktionssatz des AVR atmega kennt diese Instruktion als *Set Bit in I/O Register*, hat sie also speziell für diese I/O–Register zugeschnitten, so dass ihr kein Interrupt in die Quere kommen kann. Der 16-Bit–Opcode ist einfach. Binär geschrieben:

```
1001 1010 AAAA Abbb
```

Darin ist `AAAAA` die Register-Adresse (port), und `bbb` die Pin-Nummer. Die Syntax im Assembler dafür ist:

```
SBI A,b

z.B.
sbi $12,7  \ set bit 7 in port D
```

Die Forth–Passage bis zum `code` assembliert also die Instruktion. Und mit dem `,` (Komma) wird sie in die Definition geschrieben. Im `CBI` *Clear bit in I/O Register* ist das analog.

Auf diese Weise kommt man, ohne einen Assembler geladen zu haben, an ein assembliertes Wort. Zugegeben, für solche Ein-Instruktions-Worte geht das, der Overhead ist vertretbar, insbesondere, wenn viele bis alle Pins der Ports auch benutzt werden. Für längere Code–Sequenzen wird dieser DIY–Ansatz dann aber doch zu mühselig. In so einem Fall greift man lieber zu einem Forth–Hilfs–Assembler für die betreffende MCU. Wie das geht, hat Willem Ouwerkerk kürzlich wieder gezeigt [aux430ass.f]. Wer sich dafür interessiert, kann auch meinen eigenen Hilfsassembler für den MSP430 verwenden [fa430]. Wer lieber einen Assembler in der MCU selbst hat, schaue sich an, was Lubos Pekny für das amforth beigetragen hat. Lubos schuf einen erstaunlich kompakten Assembler [assembler.frt] für diese CPU [Size 3554B (opcode: 2980B, labels: 158B, constants Rx: 416B)] und hat eine verblüffende Lösung vorgeführt, wie vor– und rückwärts gerichtete Sprünge in nur einem Durchgang compiliert werden können.

Im 4e4th auf MSP430 LaunchPad gibt es im Kernel die Worte `CSET CCLR CTOGGLE` und `CGET`, weil diese MCU ebensolche bit-manipulierenden Instruktionen kennt. Sie heißen dort `BIS.B BIC.B XOR.B` und `BIT.B`. Anders als beim atmega gelten diese Instruktionen für den gesamten Adressraum, sind also universeller verwendbar. Sie nehmen die Adresse und die Bitmaske aus Registern, daher werden von der 4e4th–Ebene ebenfalls die Adresse und eine Bitmaske über den Datenstack an diese Instruktionen übergeben.

Ich denke es ist gut, sich richtig vertraut zu machen mit der Maschine, mit der man in Produktion gehen will. Forth erleichtert diesen Einstieg, weil man an alle Teile der MCU zur Laufzeit flexibel und interaktiv herankommt. Und damit einfach so herumzuspielen, ist bei so komplexen MCUs, wie wir sie jetzt haben, äußerst wichtig, damit man sie gut kennen lernt. Daher freue ich mich über jedes Forth auf jeder MCU, die schon grad genug Platz dafür hat. Und dabei ist es mir wichtig zu wissen, dass die jeweilige MCU gut unterstützt wird von meinem Forthsystem.

Links

```
http://amforth.sourceforge.net/
http://www.forthfreak.net/index.cgi?WillemOuwerkerk
http://www.forth-ev.de/repos/fa430/
```

Danksagung

Mein Dank gilt Enoch H. Wexler, der die Diskussion über die Notation solcher bit–manipulierenden Worte auf der Amforth–Mailingliste geführt hat. Sein Anliegen: „Ich hoffe, dass dein Beitrag (in der VD) Forth–Entwickler ermutigt, die jeweilige μC–Architektur zu beachten, und nicht nur die virtuelle Forth–Maschine“ [I hope that your article would encourage Forth developers to pay respect to the μC architecture and not just to the Forth VM.] (Quelle: Email–Korrespondenz zu diesem Beitrag.) Von ihm stammen auch die Überschrift zum Beitrag und die Forth–Beispiele.

SWAP und seine Freunde

Nicht nur wird SWAPs Name immer populärer (siehe Editorial), auch hat SWAP Freunde in aller Welt, z. B. diesen hier:

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

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

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

Wave Engine (5)

Hannes Teich

Wie in der letzten Folge angekündigt, soll diesmal mit einer neuen Version der *Wave Engine* begonnen werden. Drei Gründe: Es gibt Vereinfachungen durch den Wegfall der Schieberegister; es kommt mehr Ordnung in das Konzept; und der Code kann jeweils zugänglich gemacht werden.

Version 1 dieses „non-live“ PC-Synthesizers lief vor Jahren mit *Win32Forth*. Die bisher hier beschriebene Version 2 läuft mit *Gforth*, und das soll für Version 3 so beibehalten werden.

Es sollen vor allem die aufwändigen Schieberegister entfallen, welche die Aufgabe hatten, Daten vom Partitur-Interpreter zeitlich geordnet zum Sinus-Generator zu transportieren; sie sollen durch eine generator-freundliche Zwischendatei ersetzt werden. Zunächst aber geht es noch nicht ums Musizieren, sondern erst mal um die Tonerzeugung als solcher. Und auch die wird diesmal noch nicht erklingen können, denn dazu ist viererlei nötig: eine Datenquelle, ein Tonvorrat, ein Sinus-Generator, und schließlich der Wave-Generator, der eine extern abspielbare Datei erzeugt. Der Tonvorrat, mit dem ich heute beginnen möchte, muss – wie auch der Wave-Generator – von Anfang an komplett sein, alles Übrige wird aus kleinen Anfängen hochgezogen.

Tonumfang und Tonvorrat

Die Wave Engine ist – anders als ein Streichinstrument – mit einem endlichen Tonvorrat ausgestattet. Verglichen mit den sieben Oktaven des Klaviers ist der Tonumfang nach oben und unten um je eine Oktave erweitert und reicht vom Subkontra-C („C mit ca. 16,3 Hz) bis zum sechsgestrichenen C (c”””” mit ca. 8350 Hz). Eigentlich wären das 9 * 12 + 1 = 109 Töne, aber der Tonumfang ist hier feiner unterteilt, nämlich in 9 * 53 + 1 = 478 Töne, die sich aus der eigenwilligen Syntax der Partitur ergeben. Ich werde unten mein Motiv zu dieser Besonderheit erläutern. Die übliche 12-tönig-gleichstufig-temperierte Stimmung wird ebenfalls bedient, und es kann frei zwischen den beiden Temperaturen gewählt werden.

Die beiden Frequenztabellen

Die Frequenzen der Töne werden berechnet und in zwei separaten Tabellen abgespeichert. Genauer gesagt: Die Töne, die sich aus der Syntax ergeben, werden von 1 bis 478 durchnummeriert; aus diesen Nummern werden die Frequenzen berechnet und für den Generator in Phasenschritte gewandelt, ehe sie in den Tabellen landen. Um die rechte Frequenz zu erzeugen, muss der Sinuswinkel um angemessen kleine Schrittchen erhöht werden. Eine Sinusperiode entspricht einem Kreisbogen von 2pi, und die Auflösung der Wave-Datei beträgt 44100 Frames pro Sekunde. Somit ist die gewünschte Frequenz mit 2pi/44100 zu multiplizieren, um die Größe des Phasenschritts im Bogenmaß zu erhalten, wie er vom Forth-Wort `fsin` gebraucht wird.

```
jgt@qs:~$ gforth WE/Tontabelle.fs

base53 = 16.29822 Hz  (tiefster Ton)
base12 = 16.35168 Hz  (tiefster Ton)

Okt Sym Ton#  T53  Frequenz  Phasenschr  T12  Frequenz  Phasenschr
------------------------------------------------------------------
  Dauer  -1
  Pause   0
  0 C     1     0 1.629822E1 0.00232210    0 1.635160E1 0.00232971
  0 B#    2     1 1.651277E1 0.00235267    0 1.635160E1 0.00232971
  0       3     2 1.673015E1 0.00238364    1 1.732391E1 0.00246824
  0       4     3 1.695039E1 0.00241502    1 1.732391E1 0.00246824
  0 Db    5     4 1.717353E1 0.00244681    1 1.732391E1 0.00246824
  0 C#    6     5 1.739960E1 0.00247902    1 1.732391E1 0.00246824
  0       7     6 1.762865E1 0.00251166    2 1.835405E1 0.00261501
  0       8     7 1.786072E1 0.00254472    2 1.835405E1 0.00261501
  0       9     8 1.809584E1 0.00257822    2 1.835405E1 0.00261501
  0 D    10     9 1.833406E1 0.00261216    2 1.835405E1 0.00261501
  0      11    10 1.857541E1 0.00264655    2 1.835405E1 0.00261501
  0      12    11 1.881994E1 0.00268139    3 1.944544E1 0.00277051
  0      13    12 1.906769E1 0.00271668    3 1.944544E1 0.00277051
  0 Eb   14    13 1.931870E1 0.00275245    3 1.944544E1 0.00277051
  0 D#   15    14 1.957301E1 0.00278868    3 1.944544E1 0.00277051
  0      16    15 1.983067E1 0.00282539    4 2.060172E1 0.00293525
  0      17    16 2.009172E1 0.00286259    4 2.060172E1 0.00293525
  0 Fb   18    17 2.035622E1 0.00290027    4 2.060172E1 0.00293525
  0 E    19    18 2.062419E1 0.00293845    4 2.060172E1 0.00293525
  0      20    19 2.089569E1 0.00297713    4 2.060172E1 0.00293525
  0      21    20 2.117076E1 0.00301632    5 2.182676E1 0.00310979
  0      22    21 2.144946E1 0.00305603    5 2.182676E1 0.00310979
  0 F    23    22 2.173182E1 0.00309626    5 2.182676E1 0.00310979
  0 E#   24    23 2.201790E1 0.00313702    5 2.182676E1 0.00310979
  0      25    24 2.230775E1 0.00317832    6 2.312465E1 0.00329470
```

Abbildung 1: Hilfsprogramm zur Darstellung aller Frequenzen

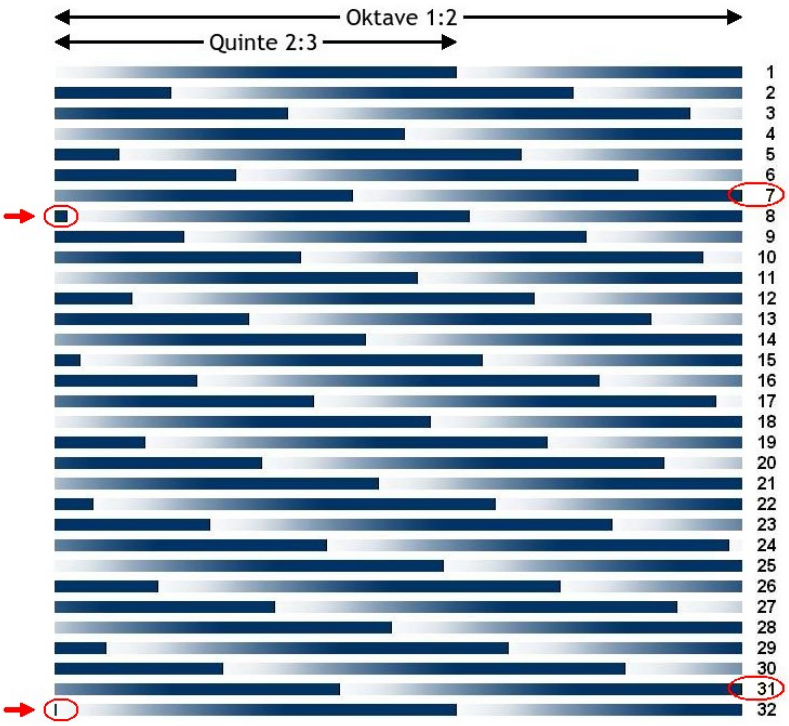


Abbildung 2: Wie Quinten in Oktaven passen: 12 Quinten in 7 Oktaven gut, 53 Quinten in 31 Oktaven exzellent.

Das Programm `freqtab.fs` im Listing auf der nächsten Seite kann von `tinyurl.com/WEpage` heruntergeladen werden. (Das ist ein Alias für die komplizierte Adresse, die am Artikelende steht.) Auch ein hier nicht abgedrucktes Programmchen (`Tontabelle.fs`) steht zum Download bereit; es gibt eine komplette Liste aller Frequenzen und Phasenschritte aus. *Abb. 1* zeigt den Anfang.

Ich pflege mit Ubuntu-Linux zu arbeiten, habe das Programmchen aber auch mit Gforth unter Windows 7 getestet. Windows ist mir etwas fremd geworden, deshalb hatte ich Mühe herauszufinden, wie dem Gforth Parameter mitgegeben werden können. Es geht vermutlich eleganter, aber mit folgender Kommandozeile im Terminal hatte ich schließlich Erfolg:

```
C:\Users\jgt>C:\"Program Files"\gforth
                \gforth Desktop\WE\Tontabelle.fs
```

Vom Sinn der 53 Töne

Es sind ein paar Bemerkungen fällig zur Besonderheit der Feinteilung des Tonumfangs und der damit korrespondierenden Partitur-Syntax (siehe auch Heft 3/2011). Ursprünglich ging es mir vor allem um diese Einteilung, daher kommt ein Verzicht nicht in Frage. Ich werde das also erläutern müssen und will mich so kurz wie möglich fassen – aber nicht kürzer.

Das musikalische Ohr empfindet ein Musikintervall dann als rein, wenn die Frequenzen ein einfaches Verhältnis bilden: *Oktave 1:2, Quinte 2:3, Durterz 4:5* etc. Kleine Abweichungen werden vom Ohr toleriert. *Abb. 2* zeigt, dass 12 Quinten etwas mehr sind als 7 Oktaven. (Der Unterschied ist als *Pythagoreisches Komma* bekannt.) Die übliche temperierte Stimmung nimmt die Quinten etwas kleiner, so dass sie genau in die Oktavenreihe passen. Obgleich der Unterschied für spitze Ohren hörbar ist, hat sich der akustische Kompromiss in der Praxis gut bewährt.

Mir geht es allerdings weniger um die Akustik als um die Semantik, also die musikalische Bedeutung der Töne. Es ist nicht gleichgültig, ob von Ton *C* eine Oktave nach oben gesprungen wird, um wieder ein *C* zu erreichen, oder ob man in sechs Ganztonschritten dorthin gelangt: *C – D – E – Fis – Gis – Ais – His*. Das *His* wird am Klavier durch die *C*-Taste bedient, aber der Namensunterschied deutet schon auf die unterschiedliche Semantik hin. Ähnlich ist es mit den Terzen, die für Dur und Moll verantwortlich sind. Die Quintenreihe *C – G – D – A – E* führt zu einem anderen *E* als das *E* im Dreiklang *C – E – G* (4:5:6). (Der Unterschied ist als *Syntonisches Komma* bekannt.) Leider werden die Terzen in der Musikpraxis skandalös diskriminiert, da solche semantische Verschiedenheit weder in Tonbezeichnungen noch in der Notenschrift repräsentiert ist. *Pythagoreisches* und *Syntonisches Komma* sind zufällig fast gleich groß, so dass sie akustisch durch ein *Mittleres Komma* (1/53-Oktave) repräsentiert werden können.

Meine These ist: Das Ohr ist tolerant, die Semantik ist es nicht. Definitionen sind wesensmäßig toleranzfrei (wie die Dreieck-Winkelsumme). Weil sich die Musikpraxis aber immer mehr darüber hinwegsetzt und den akustischen Kompromiss mit dem Tonsystem verwechselt, besteht für mich der Wunsch nach einem Instrument, das geeignet ist, semantische Unterschiede durch *Kommas*

(53stel-Oktaven) hörbar zu machen, damit sie nicht untergehen. Alles klar?

Die Noten-Schreibweise

Die Syntax ist schnell erläutert: *4A* ist das *A* in der vierten Oktave *(a')*. *4A#* erklingt einen Halbton höher *(ais')*, *4Ab* einen Halbton tiefer *(as')*. Das erwähnte *E* im Dur-Dreiklang sitzt ein Komma tiefer als das *E* der Quintenfolge und wird *4E/* geschrieben; im Moll-Dreiklang wird es durch *4Eb* ersetzt, ein Komma höher als *4Eb*. Die Zeichen können auch mehrfach auftreten; z. B. ist *0C##* formal in Ordnung. Internationaler Gepflogenheit gemäß entsprechen *4Bb*, *4B*, und *5B#* der deutschen Schreibweise *b'*, *h'* und *his"*. Das ist, was die Syntax der Tonhöhe betrifft, schon alles. (Die Tondauer wird separat formuliert.)

In der nächsten Folge sollen (via Wave-Datei) einfache Melodien zu hören sein, zunächst einstimmig und als starre Sinustöne. Der Code wird wie versprochen mitgeliefert.

Referenzen

In folgenden VD-Heften wurde bisher über die Wave Engine berichtet:
2/2011 („Top-One-Partitur"), 3/2011 , 1/2012 , 2/2012 , 4/2012.
Beachten Sie bitte den Dateibereich der Website der Forth-Gesellschaft unter
`http://www.forth-ev.de/filemgmt/viewcat.php?cid=54`
sowie die Website des Autors unter `http://www.stocket.de/WE`

Listing

freqtab.fs

```
\ -------1---------2---------3---------4---------5-----|
  cr cr ." Last edit: 22feb2013 22:40" cr
\ ######################################################
\ Aus Tonnummer (1...478, entspr. ,,C ... c""") Frequenz
\ und Phasenschritt berechnen. Die Schrittwerte werden
\ in Tabellen geladen und dort vom Generator ausgelesen.
\ ######################################################
\
\ Ausgehend vom "Kammerton" a' mit 440 Hz ergeben sich
\ für den tiefsten (Subkontra-)Ton ,,C je nach Stimmung
\ unterschiedliche Frequenzen:
\
\ Reine Stimmung (Oktaven 1:2 und Quinten 2:3):
\ Von a' aus 3 Quinten (3/2) und 3 Oktaven (2/1) tiefer:
\      freq = 440/((3/2)^3*(2/1)^3) = 16,29630 [Hz]
\ (Nur zum Vergleich, nicht verwendet)
\
\ Fast reine temperierte Stimmung mit 53stel Oktaven:
\ Von a' aus 40+4*53 = 252 Kommas (2^(1/53)) tiefer:
\      freq = 440/(2^(1/53))^252 = 16,29822 [Hz]
\
\ Die übliche temperierte Stimmung mit Zwölftel-Oktaven:
\ Von a' aus 9+4*12 = 57 Halbtöne (2^(1/12)) tiefer:
\      freq = 440/(2^(1/12))^57 = 16,35160 [Hz]
\
\ Der vom Sinus-Generator benötigte Phasenschritt wird
\ aus der Frequenz nach folgender Formel berechnet:
\      step = freq * 2pi / frames/sec [rad (Bogenmaß)]
\
\ Bei einer Auflösung von 44100 frames/sec beträgt der
\ Phasenschritt für den tiefsten Ton:
\      step = 16,29822 * 2pi / 44100 = 0,002322103 [rad]
\ bzw. step = 16,35160 * 2pi / 44100 = 0,002329708 [rad]
\
\ Dasselbe für den höchsten Ton c""", 9 Oktaven höher:
\      step = 8344,690 * 2pi / 44100 = 1,188917 [rad]
\ bzw. step = 8372,018 * 2pi / 44100 = 1,192810 [rad]
\
```

```
\ ====================================================
  2e 1e 53e f/ f**  fconstant 53root2 \ 2^(1/53) Komma
  2e 1e 12e f/ f**  fconstant 12root2 \ 2^(1/12) Halbton
          pi 2e f*  fconstant 2pi
                    fvariable 440Hz
                         440e 440Hz f!  \ Stimmgabel
                    fvariable frames    \ Auflösung
                       44100e frames f! \ Draft: 11025e
\ ----------------------------------------------------
\ Frequenz des tiefsten Tons der "53er"-Tabelle finden
: base53  ( r-440Hz -- r-freq)  53root2 252e f** f/ ;

\ Frequenz eines Tons der "53er"-Tabelle berechnen
: freq53  ( r-440Hz offs53 -- r-freq)
      base53 53root2 s>f f** f* ;
\ ----------------------------------------------------
\ Frequenz des tiefsten Tons der "12er"-Tabelle finden
: base12  ( r-440Hz -- r-freq)  12root2  57e f** f/ ;

\ Frequenz eines Tons der "12er"-Tabelle berechnen
: freq12  ( r-440Hz offs12 -- r-freq)
      base12 12root2 s>f f** f* ;
\ ----------------------------------------------------
\ Aus der Frequenz den Phasenschritt berechnen
: freq>step  ( r-freq -- r-step)  2pi f* frames f@ f/ ;

\ Wandlung des "53er"-Offset in den "12er"-Offset
: shrink  ( offs53 -- offs12)  12 * 22 + 53 / ;

\ ====================================================
\ Tabelle für die "53er"-Temperatur anlegen
  478   dup constant ftab53-items  \ 9*53 (9 Oktaven) +1
    8 * dup constant ftab53-size   \ 8-byte-Zellen
            create   ftab53 allot  \ 3824 bytes

\ Aus dem Offset Adresse in "53er"-Tabelle berechnen.
: fadr53  ( offs53 -- addr)  8 * ftab53 + ;

\ Füllen der Tabelle für die "53er"-Temperatur
: fill-ftab53  ( r-440Hz --)
      base53 freq>step \ Anfangston 16,3 Hz
      ftab53-items                \ Anzahl Float-Werte
      0 DO    fdup i fadr53 f!    \ Float-Wert laden
              53root2 f*          \ 1/53 Oktave höher
        LOOP  fdrop ;

\ Per Tonnummer Phasenschritt aus Tabelle ftab53 holen
: f53@  ( Ton# -- r-step)  1- fadr53 f@ ;
\ ====================================================
\ Tabelle für die "12er"-Temperatur anlegen
  109   dup constant ftab12-items  \ 9*12 (9 Oktaven) +1
    8 * dup constant ftab12-size   \ 8-byte-Zellen
            create   ftab12 allot  \ 872 bytes

\ Aus dem Offset Adresse in "12er"-Tabelle berechnen.
: fadr12  ( offs12 -- addr)  8 * ftab12 + ;

\ Füllen der Tabelle für die "12er"-Temperatur
: fill-ftab12  ( r-440Hz --)
      base12 freq>step \ Anfangston 16,3 Hz
      ftab12-items                \ Anzahl Float-Werte
      0 DO    fdup i fadr12 f!    \ Float-Wert laden
              12root2 f*          \ 1/12 Oktave höher
        LOOP  fdrop ;

\ Per Tonnummer Phasenschritt aus Tabelle ftab12 holen
```

```
105  : f12@  ( Ton# -- r-step)  1- shrink fadr12 f@ ;
106  \ =====================================================
107  \ shrink reduziert die (durch die spezielle Syntax sich
108  \ ergebenden) 53 Töne pro Oktave auf 12 Töne pro Oktave,
109  \ zum Auslesen der kleineren "12er"-Tabelle.
110  \
111  \ Syntax  C  B#  C\\ Db/ Db  C#  B## D// Ebb  D C## Eb\\
112  \ Ton#    1   2   3   4   5   6   7   8   9  10  11  12
113  \ offs53  0   1   2   3   4   5   6   7   8   9  10  11
114  \ offs12  0   0   1   1   1   1   2   2   2   2   2   3
115  \ Piano   C ---  Cis ----------   D ---------------  Dis
116  \
117  \             Oktave 0       Oktave 1        Oktave 9
118  \           ("Subkontra")   ("Kontra")      ("6-gestr.")
119  \            Ton#    Offs    Ton#   Offs     Ton#    Offs
120  \ C          1-2  ->  0     52-55 -> 12    476-478 -> 108
121  \ Cis/Des    3-6  ->  1     56-59 -> 13    --------------
122  \ D          7-11 ->  2     60-64 -> 14
123  \ Dis/Es    12-15 ->  3     65-68 -> 15
124  \ E         16-20 ->  4     69-73 -> 16
125  \ F         21-24 ->  5    --------------
126  \ Fis/Ges   25-29 ->  6    449-453 -> 102
127  \ G         30-33 ->  7    454-457 -> 103
128  \ Gis/As    34-37 ->  8    458-461 -> 104
129  \ A         38-42 ->  9    462-466 -> 105
130  \ Ais/B     43-46 -> 10    467-470 -> 106
131  \ H         47-51 -> 11    471-475 -> 107
132  \ =====================================================
133      440Hz f@ fill-ftab53
134      440Hz f@ fill-ftab12
135  \ =====================================================
136  0 [if] \ Zum Testen mit "1" einschalten
137
138  cr ."   +---------------------------+"
139  cr ."   | Die Tabellen sind geladen |"
140  cr ."   +---------------------------+"
141  cr
142  cr ." Stacks: " .s f.s         \ must be zero
143  cr
144  cr ." Beispiele für die Konsole:"
145  cr
146  cr ." 7 set-precision"
147  cr
148  cr ." 440e     base53    f. => 16.29822  Basisfrequenz"
149  cr ." 440e   0 freq53    f. => 16.29822  Freq Ton #1"
150  cr ." 440e  53 freq53    f. => 32.59645  Freq Ton #54"
151  cr ." 440e 477 freq53    f. => 8344.690  Freq Ton #478"
152  cr ."          ftab53 f@ f. => 0.002322103  Step Ton #1"
153  cr ."        0 fadr53 f@ f. => 0.002322103  Step Ton #1"
154  cr ."        1 f53@      f. => 0.002322103  Step Ton #1"
155  cr ."      477 fadr53 f@ f. => 1.188917  Step Ton #478"
156  cr ."      478 f53@      f. => 1.188917  Step Ton #478"
157  cr
158  cr ." 440e     base12    f. => 16.35160  Basisfrequenz"
159  cr ." 440e   0 freq12    f. => 16.35160  Freq Ton #1"
160  cr ." 440e  12 freq12    f. => 32.70320  Freq Ton #13"
161  cr ." 440e 108 freq12    f. => 8372.018  Freq Ton #109"
162  cr ."          ftab12 f@ f. => 0.002329708  Step Ton #1"
163  cr ."        0 fadr53 f@ f. => 0.002329708  Step Ton #1"
164  cr ."        1 f53@      f. => 0.002329708  Step Ton #1"
165  cr ."      108 fadr12 f@ f. => 1.192810  Step Ton #478"
166  cr ."      478 f12@      f. => 1.192810  Step Ton #478"
167  cr cr
168
169  [endif]
170  \ =====================================================
```

Forth–Gruppen regional

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

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

Hamburg Küstenforth
Klaus Schleisiek
Tel.: (0 40) – 37 50 08 03 (g)
`kschleisiek@send.de`
Treffen 1 Mal im Quartal
Ort und Zeit nach Vereinbarung (bitte erfragen)

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

Gruppengründungen, Kontakte

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

μP–Controller Verleih

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

Spezielle Fachgebiete

FORTHchips (FRP 1600, RTX, Novix) — **Klaus Schleisiek–Kern**
Tel.: **(0 40) – 37 50 08 03** (g)

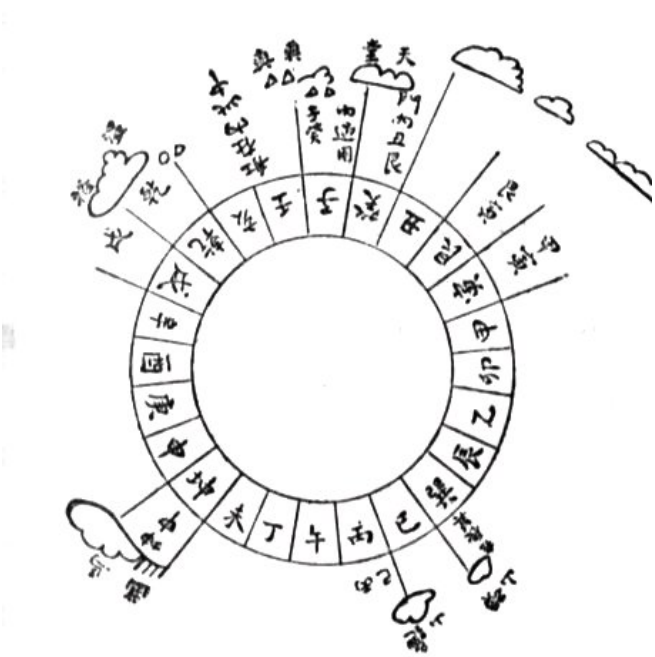
KI, Object Oriented Forth, Sicherheitskritische Systeme — **Ulrich Hoffmann**
Tel.: **(0 43 51) – 71 22 17** (p)
Fax: **– 71 22 16**

Forth–Vertrieb
volksFORTH
ultraFORTH
RTX / FG / Super8
KK–FORTH
Ingenieurbüro
Klaus Kohl–Schöpe
Tel.: **(0 70 44)–90 87 89** (p)

Termine

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

02.–03. Februar 2013 FOSDEM, Brüssel
16.–17. März 2013, Chemnitzer Linuxtage
23. März 2013, Augsburger Linux–Infotag
19.–21. April 2013 Forth–Tagung
22.–25. Mai 2013 Linuxtag, Berlin
09.–10. November 2013 OpenRheinRuhr, Oberhausen



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.

Forth–Tagung 2013

Forsthaus Graseck, Graseck 4, 82467 Garmisch–Partenkirchen
vom 19. bis 21. April 2013

Organisator:

Heinz und Ulrike Schnitter, Bernd Paysan, Email: `tagung@forth-ev.de`

Programm:

Donnerstag, 18.4.2013 Treffen der Frühankommer

Freitag, 19.4.2013 15 Uhr Beginn der Tagung
Vorträge & Workshops „Forth und Netzwerk:“
net2o, JavaScript, Web–Server.

Samstag, 20.4.2013 9 Uhr Vorträge & Workshops:
„Controller:“ msp430: noForth und 4€4th, Parallax Propeller, STRIP–Forth, GA144.
Und um Forth als Sprache selbst.
Exkursion Partnachklamm (mit Fackeln)

Sonntag, 21.4.2013 9 Uhr Jahresversammlung
13 Uhr Ende der Tagung