Vierte Dimension

4/2010

Das Forth–Magazin

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

In dieser Ausgabe:

GO-COLON

Forth auf dem Ben NanoNote

Forth–VM und realer Forth–Prozessor

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.

Leserbriefe und Meldungen 5

GO-COLON 7
Fred Behringer

Forth auf dem Ben NanoNote 10
David Kühling

Forth–VM und realer Forth–Prozessor 19
Willi Stricker

Quelle: Florida Center for Instructional Technology (FCIT) at USF

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

Lieber Leser,

ich wünsche Dir ein frohes Neues Jahr 2011.

Du hältst das vierte Heft des 26. Jahrgangs unseres Forth–Magazins in den Händen. Eigentlich ist dieses Heft ja für das vierte Quartal des vergangenen Jahres geplant gewesen, aber um das Repertoire ab Artikeln ist es, wie Du ja auch am Umfang dieses Hefts sehen kannst, nicht gut bestellt: Wir leben von der Hand in den Mund und einen Fundus von Artikeln können wir nicht vorweisen. Daran wird sich auch erst etwas ändern, wenn Du Dir einen Ruck gibst und über Deine Forth-Aktivitäten hier in der Vierten Dimension berichtest. David Kühling hat den Hilferuf im Editorial der Doppelausgabe 2+3/2010 bereits vernommen und uns den hervorragenden Artikel über den Winzcomputer *Ben NanoNote* geschrieben. Danke!

Zu den gut bekannten und stetig schreibenden Autoren gehören Fred Behringer, der uns über `go-colon` berichtet, ein Wort, um im Threaded–Code an beliebige Stellen zu springen, und auch Willi Stricker, der über seinen Forth–Prozessor *STRIP* schreibt. Mehr Artikel wären sehr zu begrüßen.

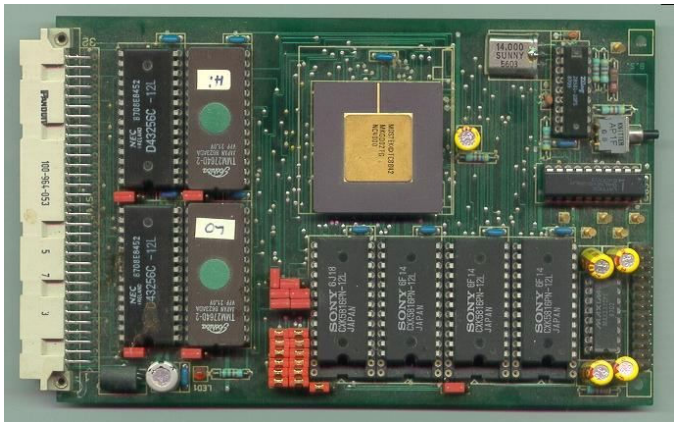
Was bringt uns das Jahr 2011? Nun — im April (vom 15.4. bis 17.4. siehe Rückseite dieser Ausgabe) wird die Jahrestagung 2011 der Forth-Gesellschaft in Goslar stattfinden. Melde Dich bitte alsbald an.

Vom 11. bis zum 14. Mai findet der Linuxtag 2011 in Berlin statt und die Forth–Gesellschaft wird dort wohl wieder mit einem Stand vertreten sein. Einen Besuch auf dem Linuxtag und auf unserem Stand kann ich sehr empfehlen.

Auch 2011 wird im September wieder eine EuroForth-Konferenz stattfinden, wobei derzeit der Veranstaltungsort noch offen ist. Sowohl Estland, Südafrika als auch Österreich sind in Diskussion.

Also — einen guten Start ins Jahr 2011 und hilf bitte mit, damit wir das Heft 1/2011 so bald wie möglich herausgeben können.

Ulrich Hoffmann



Novix EB1–Board aus dem Jahr 1987 mit dem Forth-Prozessor NC4000
Quelle: `http://www.vaxcluster.de`

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
`http://www.forth-ev.de/filemgmt/index.php`

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

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

Zu Martin Bitters Artikel aus Heft 2+3/2010

Hallo Martin,

du möchtest wissen, ob jemand an einer Fortführung deines Rittersitz-Besetz-Spiels über König Artus und dessen Tafelrunde interessiert sei. Ja, ich! Henry Vinerts, unser langjähriger Korrespondent aus Amerika, hat deine 16 Seiten als den interessantesten Artikel aus Heft 2+3/2010 eingestuft. Ich schließe mich diesem Urteil an und bin stark an einer Fortsetzung interessiert. Als Allererstes würde ich gern wissen, wie die Regel mit den drei zusammenhängenden Leersitzen zu verstehen sein soll: Mindestens drei oder genau drei? Wahrscheinlich genau drei? Die müssten dann aber wohl durch besetzte Sitze an den Enden begrenzt sein? Bei beispielsweise fünf zusammenhängenden leeren Stühlen wäre die Situation nicht eindeutig: 1+2+3 oder 2+3+4 oder 3+4+5 könnten (gleichzeitig) besetzt werden (?). Die Begriffe *Ecke* und *nebeneinander* müssten wohl in einem verallgemeinerten Sinne verstanden werden? Ist *nebeneinander ums Eck herum* zugelassen? Was ist, wenn der *rechteckige* Tisch sieben Sitze gleicher Bauart hat? Sollen auch Ecken als Begrenzung zählen? Hast du dir alle Regeln ausgedacht oder gibt es auch Vorläufer in der Literatur über *Recreational Mathematics*? Danke, Martin, für die immense Arbeit, die du in diese zum Nachdenken anregende Geschichte da wieder reingesteckt hast! Auf Heft 1/2011 bin ich gespannt.

Herzlichen Gruß, Fred

Lieber Fred und liebe Herren des Direktoriums:

ich darf meinen Brief an euch genau so beginnen wie vor einem Jahr und einem Tag, kurz nach unserem Forth–Treffen 2009, und mich deucht, ich hätte meine heutige Nachricht an euch auch einfach vom letzten Jahr her übernehmen können. Zunächst einmal meinen aufrichtigen Dank dafür, dass ihr mich wieder in eure Verteiler–Liste für die Vierte Dimension aufgenommen habt. Das Heft kam am 8. November an und ich nahm es zum SVFIG–Treffen in Stanford mit, um es einem rekordverdächtigen Teilnehmer–Kreis herumzureichen, der im Laufe des Tages auf eine Zahl von etwa 50 anwuchs.

Ich hatte ein ganzes Jahr lang kein SVFIG–Treffen mehr besucht und kam mir umso stärker total abgehängt vor und hatte noch mehr als je zuvor das Gefühl, einer der *ältesten Anfänger* zu sein. Aber ich durfte mich darüber freuen, mindestens ein Dutzend alter Freunde wiederzutreffen, die ich seit 20 Jahren kenne. Und ich habe mit Freude neue Leute kennengelernt, die gerade ihre ersten Schritte in die Forth–Welt lenken.

Einen besseren Bericht aber als den, den ich euch geben kann, könnt ihr unter `http://www.forth.org/svfig/kk/11-2010.html` finden. Soweit ich gehört habe, wird gerade ein Gruppenfoto zusammengezimmert, das demnächst auf unserer Website zu sehen sein wird.

Ich habe meine Sammlung von *Forth Dimensions*–Heften (die seinerzeit von der FIG, der Forth Interest Group, herausgegeben worden waren) der SVFIG, der Silicon–Valley–FIG, gestiftet. Die Hefte der *Vierten Dimension* dagegen werde ich in meinem Besitz behalten, in der Hoffnung, dass sie sich dermaleinst in entsprechenden Archiven unseres Landes wiederfinden werden.

Sollte man mich fragen, welchen Artikel ich für den interessantesten des Doppelheftes 2+3/2010 halte, so würde ich meine Stimme, die eines *ewigen Anfängers*, Martin Bitter mit seinen 16 Seiten über die Könige Arthur und Minos geben.

Ich wünsche euch allen alles Gute für die kommenden Feiertage und für das Neue Jahr!

Euer Freund Henry Übersetzt von Fred Behringer

Zu Willi Strickers STRIP-Artikel im vorliegenden Heft

Bei vielen Artikeln macht es wirklich Spaß mitzuhelfen, sie durch Korrektur-Arbeiten auf Hochglanz zu bringen. Willis STRIP gehört dazu. Willi schreibt u.a.: „Ein STRIP-Forth-System auf einem größeren Board mit RAM und Flash-ROM ist in Arbeit. Eine detaillierte Beschreibung für einzelne Baugruppen, wie etwa Hardware-Komponenten, die Interrupt-Organisation oder das Bootprogramm für die Programmier- und Debug-Schnittstelle, soll in weiteren Artikeln folgen.“ Danke, Willi.

Ich für meinen Teil gehöre zu denjenigen, die auf die baldige Veröffentlichung sehr gespannt sind. (Interessant ist Willis Return-Bit: Ausschöpfung der Möglichkeiten des Komprimierens bis zum letzten Bit.)

Fred Behringer

FSL und Forth200x

Charles G. Montgomery schrieb neulich [1] auf comp.lang.forth:

Seit seinem Bestehen basiert das Projekt der wissenschaftlichen Sammlung zu Forth (Forth Scientific Library Project FSL) auf Code im ANS-94-Forth-Standard, der Portabilität und Unzweideutigkeit halber. Dabei kann der FSL-Code natürlich Erweiterungen des Standardforth enthalten, wenn eine adäquate Dokumentation mitgeliefert wird. Und dafür wiederum kann es erforderlich sein, dann eine Standard-Implementierung einer solchen Erweiterung ebenfalls anzugeben.

Nun, die Anstrengungen des Forth 200x haben zur Spezifikation mehrerer wohlüberlegter Erweiterungen des ANS-94 geführt, die auch für unsere FSL-Beitragenden und -Nutzer recht nützlich sein könnten. Daher ist es angebracht zu erwägen, ob Forth-200x-Funktionen inzwischen in FSL akzeptiert werden könnten.

Ich hatte zunächst daran gedacht, einige Möglichkeiten aufzuzeigen und um deren Diskussion zu bitten. Meine Vorschläge wären vom Erhalt des status quo bis hin zum automatischen Akzeptieren aller anerkannten Forth200x-Erweiterungen gegangen. Aber nach einigem Nachdenken bin ich zu dem Schluss gekommen, einen spezifischen und begrenzten Vorschlag für die weitere Erörterung zu machen.

Meiner Ansicht nach sollte die Entscheidung über solche Angelegenheiten von den Anwendern und potentiellen Anwendern des FSL kommen. Stimmte diese wenn auch vage definierte Gruppe darin überein, könnte es als FSL-*Richtlinie* und in Dokumente wie den Leitfaden für

Beitragende und Reviewer von Beiträgen aufgenommen werden, und weiter auf die Nutzer bauen. Daher bitte ich um Kommentare zu folgender Anregung:

Die 200x-Worte `[DEFINED]` und `[UNDEFINED]` aus XXXXX sind für FSL akzeptabel, wenn sie der 200x-Spezifikation folgen. Die Funktionalität der 200x-Worte `DEFER` und `IS` ist schon durch die FSL-Worte `v:` und `defines` gegeben, aber die Verwendung von `DEFER` und `IS` wird auch für FSL-Code akzeptiert.

Diese Worte scheinen mir die offensichtlichsten Fälle von klarem Vorteil einerseits und geringem Konfliktpotential oder Verwirrung andererseits zu sein. Weitere Kandidaten wären `PARSE-NAME INCLUDE REQUIRE REQUIRED TO VALUE FVALUE FTRUNC` und vielleicht noch einige. Spezifikationen für akzeptierte 200x-Worte und deren ANS-94-Implementation können dann auf der FSL-Seite `http://www.taygeta.com/fsl/sciforth.html` bereitgestellt werden und Dateien oder Links zu den 200x-Quellen versehen werden.

Bitte kommentiert diese Fragen: Ist jemand dagegen, dass `[DEFINED] [UNDEFINED] DEFER IS` in den FSL-Code aufgenommen werden?
Gibt es noch weitere 200x-Worte, die aufgenommen werden sollten?
Oder sollte man besser beim puren ANS-94 bleiben?

Danke, cgm
(current FSL maintainer, Charles G. Montgomery)

Marcel Hendrix antwortete:
[..]
Einverstanden, nur zu.
Ob noch andere Worte aufgenommen werden sollten, dürfte eine Geschmacksfrage sein.
> Oder sollte man besser beim puren ANS-94 bleiben?

Einige der numerischen Algorithmen, die hier kürzlich diskutiert worden sind, sind sehr gut geworden, z.B. das IEEE-Wordset. Im Laufe dieser Diskussion hat sich gezeigt, dass gegenwärtige und geplante Erweiterungen des Standards sehr nützlich sind, um die Systemenabhängigkeit sauber zu isolieren. Mir würde es besser gefallen, alle möglichen Erweiterungen dieser Art zuzulassen und damit dann auch gut ausgetestete und akkurate Algorithmen zu bekommen, als Erweiterungen auszuschließen und bei Algorithmen zu landen, die auf halbem Wege stehen geblieben sind und weniger Leistung und Genauigkeit bieten. marcel

David N. Williams meinte daraufhin:
... dem stimme ich zu. Ich favorisiere es ebenfalls alle Forth-200x-Erweiterungen zuzulassen. David

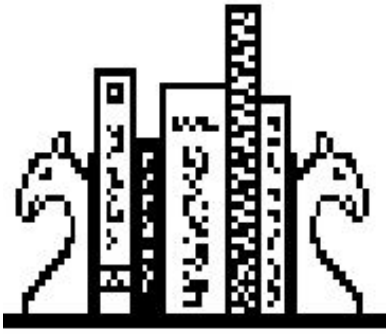
Und Brad schrieb:
Ich stimme dafür, 200x anzunehmen. Falls Code in ANS94 laufen muss, kann ein Kompatibilitäts-Layer im FSL bereitgestellt werden. Wir sollten uns nicht behindern lassen, um so etwas wie eine C-Portabilität zu erreichen.

`REQUIRE` wäre auch eine gute Zugabe, denn es ist mitunter sehr mühsam, die Abhängigkeiten von Dateien herauszufinden. Brad

Und noch weitere Stellungnahmen gab es, meistens Befürworter des Vorhabens, Forth 200x aufzunehmen. Weitere Einzelheiten können in der Quelle nachgelesen werden. mk

Quelle:
[1] comp.lang.forth, Charles G Montgomery, Dec 17, 5:02 pm

Link:
The Forth Scientific Library Project
`http://www.taygeta.com/fsl/sciforth.html`



Logo der Forth Scientific Library

Quelle: http://www.taygeta.com/fsl/sciforth.html

GO-COLON als Highlevel–Pendant zu GO in Turbo–Forth und ZF

Fred Behringer

Angenommen, `' dup >body` ergibt im konkreten Fall `3ff`. Dann kann man in Turbo–Forth und in ZF in das Parameterfeld der Code–Definition dup springen und die Eingabe `4711 3ff go` beispielsweise liefert `4711 4711` auf dem Stack. Man kann mit `go` so auch mitten in das Parameterfeld (nicht nur an den Anfang) einer beliebigen Code–Definition springen und den Rest der Code–Sequenz ab da abarbeiten lassen. Mit `go-colon` aus dem vorliegenden Artikel gelingt das auch bei Colon–Definitionen — und das sogar auf überraschend einfache Weise.

Mit `go` kann man in Turbo–Forth und ZF mitten ins Parameterfeld (PF) einer Code–Definition hineinspringen und den Rest der Maschinenbefehle (bis zum Abschluss per `next end-code`) abarbeiten lassen. `go` erwartet eine *vernünftige* Adresse auf dem Stack, d. h., es muss sich um das Anfangsbyte eines PC–Assemblerbefehls handeln. Zum Aufspüren der gesuchten Adresse steht einem `dump` zur Verfügung. Das System kann nichts davon wissen, dass es sich um eine (wenn überhaupt existente) schon vorhandene Code–Definition handelt. Also kann man per `go` auch an den Anfang einer beliebigen (dazu eigens, z. B. per Hexcode und `c,` konstruierten) Folge von Maschinenbefehlen im System springen — soweit diese Folge mit dem bei Code–Definitionen üblichen Abschluss `next end-code` endet.

Als Sonderfall kann man die Wirkungsweise von `execute` betrachten. `execute` erwartet die CFA eines Forth–Wortes `<name>` auf dem Stack (die man sich mit Hilfe des Ticks per `' <name>` beschaffen kann) und führt nach Aufruf der Sequenz `' <name> execute` das Wort `<name>` aus.

Bei `execute` ist es egal, ob `<name>` eine Code– oder eine Colon–Definition ist. Ist es eine Code–Definition, dann wirkt `' <name> execute` genau so, als wenn man `go` auf die PFA (die Anfangsadresse des PFs) von `<name>` wirken lässt: `' <name> >body go` = `' <name> execute` ('=' im Sinne der Funktionsgleichheit).

Was fehlt, ist die Möglichkeit, mitten in das PF einer Colon–Definition hineinzuspringen und die restlichen mit ihren CFAs im PF abgelegten Worte abarbeiten zu lassen. Das geht in Forth (zumindest in Turbo–Forth und ZF) so einfach, dass man es kaum glauben kann, nämlich mit der folgenden Colon–Definition, die ich `go-colon` nennen möchte. (Ich verwende `BSP:` zur Kennzeichnung von Beispielen.)

```
BSP:    : go-colon ( ad -- ) >r ;
```

Dieses Wort `go-colon` erwartet eine *vernünftige* Adresse *ad* aus dem PF derjenigen Colon–Definition, in die hineingesprungen werden soll, und arbeitet ab dieser Adresse *ad* den Rest der dort liegenden CFAs (bis zum abschließenden ;) ab. *Vernünftig* soll heißen, dass *ad* die CFA eines schon definierten Forth–Wortes zum Inhalt hat. Ein Sprung zwischen (`lit`) und dem von (`lit`) bestimmten numerischen Wert oder ein Sprung in das Innere eines Strings hinein wären *unvernünftig*.

Auch hier, d. h., falls unvernünftige Fälle vermieden werden, weiß das System nicht, dass es sich um die restlichen CFAs in einer Colon–Definition handelt. Also kann man auch mit `go-colon`, ähnlich wie bei `go` für Maschinencode–Folgen, an den Anfang einer beliebigen Folge von (durch ihre CFAs repräsentierten) Forth–Worten springen — soweit diese Folge einen Abschluss mit `;` aufweist. Die Frage, wie man eine solche *kopflose* Colon–Definition (eine Folge von CFAs mit Abschluss per `;`) erzeugen soll, lässt sich mit den üblichen Mitteln sofort klären. Ich gebe ein Beispiel.

```
BSP:    here
        ] 5 . 6 . 7 . [ !csp ;
        go-colon
        [ret] 5 6 7 ok
```

Hierbei wird über `: !csp sp@ csp ! ;` der Absicherungs–Mechanismus gegen ungewollte Stackveränderungen — und dessen Fehlermeldungen — ausgeschaltet. Dieses Beispiel funktioniert auch bestens in ZF! Mit `here` wird die momentane Dictionary–Adresse festgehalten (die man sich natürlich auch in einer Konstanten oder in einer Variablen aufbewahren kann). Alles, was zwischen `]` und `[` steht, wird ab `here` ins Dictionary compiliert. Und das Wort `;` schließt die *kopflose* CFA–Folge `5 . 6 . 7 .` wie bei `go-colon` verlangt ab. Die Folge steht ab dann im Dictionary und ist jederzeit per `go-colon` aufrufbar — soweit man sich die Anfangs–Adresse der Folge gemerkt hat. Im Beispiel steht sie von `here` her noch auf dem Stack und `go-colon` erweckt die kopflose Folge zum Leben: `5 6 7 ok` .

An sich ist nicht einzusehen, warum man nicht auch die Compilation solcher kopflosen CFA–Folgen über die Stackveränderungs–Sicherung von Turbo–Forth und ZF absichern soll. Im folgenden Beispiel wird das gemacht.

```
BSP:    here
        !csp ] 5 . 6 . 7 . [ ;
        go-colon
        [ret] 5 6 7 ok
```

Die Auflösung der Absicherung gegen ungewollte Stackveränderungen, nämlich `?csp` (`: ?csp sp@ csp @ ;`) steckt (nach wie vor) im Semikolon (`;`). Bei Colon–Definitionen versteckt sich (in Turbo–Forth und ZF) die Einleitung der Absicherung gegen Stackveränderungen, nämlich `!csp` (`: !csp sp@ csp ! ;`), schon im Colon (`:`).

Natürlich kann man die Erzeugung kopfloser CFA–Folgen der Bequemlichkeit halber noch über ein definierendes Wort automatisieren (Aufgabe!).

Zurück zu `go`! `go` ähnelt dem BASIC–Befehl `goto`. Mit `goto` kann man (in BASIC) glänzend Spaghetti–Code erstellen. Mit `go` und `go-colon` in Forth also auch? Nicht hundertprozentig! Spaghetti–Code ist allgemein verpönt. Verpönt heißt nicht verboten. Forthler verhalten sich gegenüber Programmier–Systemen, die ihnen Vorschriften machen, reserviert. Das Streben nach Freiheit beim Programmieren ist sicher einer der Gründe, weshalb man sich mit Forth beschäftigt. (ANS–Forth macht kaum Vorschriften. ANS–Forth liefert lediglich *Empfehlungen*.)

Zur Veranschaulichung der Wirkungsweise von `go-colon` möchte ich ein Beispiel bringen, Als Demonstrationsobjekt suche ich dazu eine Colon–Definition, die in ihrem PF nur CFAs enthält — keine Strings oder dergleichen. Zum problemlosen Hineinspringen (in die betrachtete Colon–Definition) kann ich mir dann die Anspring–Adressen durch einfaches Abzählen aus der PFA (der Anfangsadresse des PFs der Colon–Definition) heraus beschaffen. Ob die dann vom System aufzurufenden CFAs selbst zu Code– oder zu Colon–Definitionen gehören, ist egal.

Ich rekonstruiere mir zu diesem Zweck (umständlich, aber natürlich nur zur Demonstration) das Forth–Wort `8*` als Colon–Definition. Um die Erklärungen durchsichtiger zu gestalten, konstruiere ich zunächst ein Forth–Wort, das den obersten Stackwert um genau eine Bitstelle nach links (zu höheren Bitstellen hin) verschiebt. Es sei `1-lshift` genannt (in ANS–Forth entspricht das der Kombination 1 lshift):

```
BSP:    code 1-lshift
            ax pop
            ax shl
            ax push
        next end-code
```

Damit kann ich `8*` als Colon–Definition fassen zu:

```
BSP:    : 8* 1-lshift 1-lshift 1-lshift ;
```

Das ist wirklich umständlich, verdeutlicht aber (hoffentlich) das, was gezeigt werden soll. Mit Hilfe von `dump` (oder aber über `' 8* >body` und Abzählen) suche ich mir die benötigten Adressen im PF von `8*` heraus: PFA, PFA+2, PFA+4 und PFA+6. (Das von mir verwendete Turbo–Forth und auch ZF sind 16–Bit–Systeme.) Ruft man diese Fassung von `8*` auf (die Warnmeldung darüber, dass `8*` schon existiert, kann ignoriert werden), dann wird der oberste Stackwert, sagen wir n, dreimal um je eine Bitstelle nach links verschoben. Die Zahl n, die ja als Binärzahl auf dem Stack liegt (im folgenden BSP ist es die Zahl 1), wird also insgesamt mit 8 malgenommen. Es ergibt sich (Adresswerte aus einem von mir durchgeführten Experiment in Hexdarstellung):

```
BSP:    hex                         [ret] ok
        ' 8*            u.          [ret] 6771 ok
          6771 >body    u.          [ret] 6773 ok
        1 6773 go-colon .           [ret] 8 ok
        1 6775 go-colon .           [ret] 4 ok
        1 6777 go-colon .           [ret] 2 ok
        1 6779 go-colon .           [ret] 1 ok
```

Ich habe hier `u.` statt einfach nur `.` geschrieben, weil in ZF (in Turbo–Forth nicht) in der Bildschirmanzeige auch negative Adressen auftreten können und weil ich Adressen–Angaben (verständlicherweise) immer nur nicht–negativ lesen möchte.

Um `go-colon` ein weiteres Mal (diesmal in anderer Weise) mit `execute` zu vergleichen, möge (mit den Adressen des eben gebrachten Beispiels) das folgende Beispiel betrachtet werden:

```
BSP:    hex                         [ret] ok
        1 6773 @ execute .          [ret] 2 ok
        1 6775 @ execute .          [ret] 2 ok
        1 6777 @ execute .          [ret] 2 ok
```

Dass hierbei immer nur 2 herauskommt, hat seine Richtigkeit, da ja `execute` in allen drei Fällen immer nur das eine und einzige Wort ausführt, dessen CFA an der angesprungenen Adresse im PF liegt, nämlich `1-lshift`, nicht etwa den ganzen Rest der im PF noch folgenden CFAs. (Sonst hätte man ja gleich mit `execute` arbeiten können und die Einführung des Wortes `go-colon` wäre unnötig gewesen.)

Ich weiß nicht mehr genau, wie ich auf `go-colon` gekommen bin. Es steht bei mir auf der Festplatte im Forth–Verzeichnis `tools` als Datei–Eintragung aus dem Jahre 1990.

`go-colon` jongliert mit dem Returnstack. Das ist nicht verboten, `Do`–`Loop`s machen das auch. Interessant hierzu ist vielleicht noch der Artikel *Lusstructuren maken zonder IMMEDIATE* von Coos Haak [CH] aus dem Vijgeblad 43 (1993), S.10. Dort geht es darum, Unendlich–Schleifen ohne die `Immediate`–Eigenschaft von `begin` und `again`, allein durch Operieren mit den Returnstack–Worten `r> r@ >r` , zu erzeugen.

`again` entspricht in gewisser Weise dem obigen `go-colon`.

Für den Aussprung lässt sich dabei `exit` ebenfalls durch direkten Zugriff auf den Returnstack fassen.

Coos Haaks Vorschläge finden sich in Listing 1 auf der gegenüberliegenden Seite.

Und zur Funktions–Überprüfung hier noch (von mir) ein Wort zur Abfrage auf [ret]. Es entspricht in etwa dem beispielsweise in `dump` sehr nützlichen Wort `stop?` von Turbo–Forth (in ZF nicht vorhanden), hier aber mit den Returnstack–Fassungen von `begin`, `again` und `exit` von Coos Haak (mit dessen `begin` und `again` und dabei aber mit dem üblichen `exit` von Turbo-Forth und ZF geht es nur nach doppelter Betätigung der Eingabe-Taste):

```
BSP:    hex
        : weiter? ( -- )
            begin
              key 0d = if exit then
            again ;
```

Nach dem Aufruf von `weiter?` passiert beim Drücken einer beliebigen anderen Taste als der Eingabetaste nichts, aber auch rein gar nichts (die Eingabe–Schleife wird leer

```
: begin ( -- ) r@ >r ;   \ Kopiert die Adresse nach begin auf den Returnstack
: again ( -- ) r> drop   \ Entfernt die Adresse nach again vom Returnstack
               r@ >r ;   \ Springt an die Adresse nach begin
: exit  ( -- ) r> drop   \ Entfernt die Adresse nach begin vom Returnstack
               r> drop ; \ Das wirkt genau so wie das ursprüngliche exit
```

Listing 1: Kontrollstrukturen nach Coos Haak 1993

durchlaufen). Genau dann, wenn die Eingabetaste gedrückt wird, wird diese `begin`–`again`–Schleife, wie man es von einer solchen Schleife erwartet, verlassen. (`0d` ist der von `key` gelieferte Hexcode für die Eingabetaste.) Das System pausiert also so lange, bis die Eingabetaste gedrückt wird. Baut man in diese `begin`–`again`–Schleife dann noch ein (sinnvolles) Programm ein (beispielsweise für die Ausgabe einer jeweils weiteren Zeile im Wort `dump`), dann hat man ein schrittweise arbeitendes `dump`, das mit der Eingabetaste verlassen werden kann. (Das Ganze nur mal so, zur Übung.)

Und noch was: Als Forth–Amateur hat man die Möglichkeit, sich in jedem Einzelfall interaktiv *ganz schnell* zu vergewissern, dass der angewendete Trick (hier `go-colon`) auch wirklich funktioniert. Andererseits gibt es, wie oben schon gesagt, Fälle, in denen `go-colon` nicht gehen kann (nicht ohne Vorsichtsmaßnahmen). Man betrachte das Wort:

```
BSP:     : xxx 4 . 4 . 4 . ;
```

und versuche, im PF von xxx mit dem Wort `go-colon` unmittelbar vor einen der drei Werte 4 (also zwischen `(lit)` und dem durch `(lit)` als *Literal* gekennzeichneten Wert 4) zu springen ;-) , also z. B. `' xxx >body 8 + go-colon` [ret] ... Das geht selbstverständlich nicht! Mit

```
BSP:     ' xxx >body 6 + go-colon [ret]
```

dagegen geht es prima!

Die hier gemachten Vorschläge beziehen sich auf Turbo–Forth und ZF. Sie sind natürlich sehr stark vom Aufbau des jeweiligen Forth–Systems abhängig. (Im vorliegenden Artikel geht es nicht um Forth als Sprache, sondern um Forth als System.) In MinForth 1.5 von Andreas Kochenburger [AK] und in 32/FORTH 3.07 von Rick VanNorman [RV] funktioniert das eine oder andere der im Vorliegenden besprochenen Vorschläge nicht. Es wäre sicher interessant zu untersuchen, inwieweit sich die Vorschläge an diese beiden Systeme (beide sind ANS–Forth) anpassen lassen. Diese Frage drängt sich auf, sobald man erkannt hat, dass die Konstruktion von Coos Haak (`begin again exit über >r r@ r>`) ohne Weiteres auch in 32/FORTH von Rick VanNorman funktionieren. (Ich habe das mit dem von mir oben besprochenen Wort `weiter?` überprüft.)

Quellen

[AK] Andreas Kochenburger: MinForth — Minimalistic Forth in C.
http://home.arcor.de/a.s.kochenburger/minforth.html — 34k .

[CH] Coos Haak: Lusstructuren maken zonder IMMEDIATE.
http://www.forth.hccnet.nl/vijgebladarchief/ — 118k .

[RV] Rick VanNorman: 32/Forth.
http://www.taygeta.com/forthcomp.html — 21k .

Dann hab ich `GOTO 500` getippt — und hier bin ich.

Quelle: Thinking Forth

Forth auf dem Ben NanoNote

David Kühling

In der 2+3-Ausgabe der Vierten Dimension berichtete Carsten Strotmann über den OpenMoko-Wikireader, ein portables Gerät mit Open-Source-lizenzierten Bauplänen und Software, das ein Forth enthält.

Nun möchte ich euch in diesem Artikel den großen Bruder des Wikireaders vorstellen: den Ben NanoNote. Gebaut wird er von Qi Hardware, einer Firma die von ehemaligen OpenMoko-Mitarbeitern gegründet wurde. Wie auch OpenMoko hat sich Qi Hardware der Entwicklung von Open-Source-lizenzierter Hardware verschrieben, und seit der Firmware-Version vom 17. 11. 2010 enthält der NanoNote nun auch von Hause aus mit gForth ein vollwertiges Forth-System.

Eine Kreuzung aus PDA und Netbook

Der NanoNote entspricht in der Bauweise einem stark geschrumpften Netbook. Die Daumen-Tastatur hat ein Qwerty-Layout, bei dem die oberste Tastenreihe mit Ziffern fehlt. Ersetzt wird diese durch Mehrfachbelegungen der anderen Tastenreihen. Es gibt tatsächlich alle Sonderzeichen, die sich ein Programmierer wünscht.

Entsprechend geschrumpft ist auch das 3-Zoll-Display. Es bietet 320x240 Pixel mit 24-Bit-Farbwerten. Der 6x10-Font in der aktuellen Firmware stellt 53x24 Zeichen dar, nicht ganz ein Forth-Screen, aber doch völlig ausreichend zum Bearbeiten von Programmen.

Ein Mono-Lautsprecher ist eingebaut, ebenfalls ein Mikrofon. Ein 3.5-mm-Klinkenstecker erlaubt auch Tonausgabe in Stereo.

Gespart wurde bei den Anschlussmöglichkeiten von Peripheriegeräten. Lediglich eine USB-Mini-Buchse zum Anschluss an einen PC ist vorhanden, die auch auch zum Aufladen des Akkus dient. Eine USB-Schnittstelle im “Host”-Modus zum Anschluss von USB-Geräten wird erst in einer Folgeversion des NanoNote vorhanden sein. Die 2GB an fest eingebautem Flash-Speicher im NanoNote können mittels preiswerter Micro-SD Karten um bis zu 16GB aufgestockt werden.

126g bringt das Ganze, samt Lithium-Ionen Batterie, auf die Waage. Leichter als so manches Mobiltelefon. Beziehen kann man den NanoNote in Deutschland über `pulster.de`, für 119 € plus Versand. Etwas günstiger erhält man ihn aus dem Ausland über `hackable-devices.org` oder über `tuxbrain.com`, ab einer gewissen Stückzahl amortisieren sich dann die höheren Versandkosten.

Technische Details

Im NanoNote arbeitet ein Ingenic Xburst System-On-Chip, mit einer MIPS32-kompatiblen CPU, getaktet auf 336 MHz, angebunden an 32 MB SD-RAM. Die CPU kommt ohne Fließkomma-Einheit daher, dafür ist aber eine spärlich dokumentierte Vektor-Einheit für Multimedia-Zwecke vorhanden, die man mit moderatem Aufwand im gForth-Assembler unterstützen könnte.

Bild 1: Der Ben NanoNote im Größenvergleich mit einem handelsüblichen Bleistift

Software

Die Firmware, mit der der NanoNote ausgeliefert wird, basiert auf OpenWRT-Linux. Aktuelle Firmware-Versionen präsentieren nach dem Booten einen einfachen grafischen Programmstarter. Aber wie bei Desktop-Linuxen, kann man mittels Strg+Alt+F2 auf eine Text-Konsole umschalten. Stört einen die kleine Schriftgröße der Linux-Konsole, startet man das Programm `fbterm`, ein Terminal im Terminal, das moderne Grafikbibliotheken für die Schrift-Darstellung benutzt. Eine Eigenheit der Linux-Text-Konsole beim NanoNote ist, dass sie Text nicht farbig darstellen kann. Auch dabei schafft aber die Verwendung von `fbterm` Abhilfe.

Inbetriebnahme

Zunächst bekommt der Benutzer "root" ein Passwort. Um dieses zu setzen, wechselt man per Strg+Alt+F2 auf die Konsole, und gibt ein:

```
$ passwd
```

gefolgt von dem zu setzenden Passwort.

Verbindet man den NanoNote per mitgeliefertem USB-Kabel mit einem PC, meldet sich dieser als Netzwerkgerät an. Jedenfalls, falls der PC unter Linux läuft, wovon ich im Folgenden ausgehe. Nun gibt man der PC-Seite der USB—Verbindung die IP Adresse 192.168.254.100:

```
$ ifconfig usb0 192.168.254.100
```

Und kann dann den NanoNote unter der Adresse 192.168.254.101 erreichen:

```
$ ping 192.168.254.101
```

Per SSH kann man sich jetzt über Netzwerk anmelden:

```
$ ssh root@192.168.254.101
```

Oder Dateien zum NanoNote kopieren

```
$ scp *.fs root@192.168.254.101:
```

Wir loggen uns noch einmal auf dem NanoNote ein und starten `gforth`. Schlägt das Kommando fehl, läuft der NanoNote wohl unter einer zu alten Firmwareversion. Der Kasten "Aktualisierung der Firmware" erläutert kurz, wie man das behebt. Wer gForth-Quelltexte direkt auf dem NanoNote editieren möchte, hat eine reichhaltige Auswahl an Editoren zur Verfügung: joe, jstar, jpico, jmacs und vi, um nur einige zu nennen.

Programmieren in Forth

Einfache Programme kann man nun nativ auf dem NanoNote verfassen und mittels gForth ausführen. Forth-Skripte erhält man, in dem man die Programme mit

```
#!/usr/bin/gforth
```

beginnt, und diese dann mittels `chmod +x` ausführbar macht. Man ist aber zunächst begrenzt auf den Funktionsumfang der in gForth enthaltenen Wörter. Insbesondere hat man keine einfache Möglichkeit, Grafik auszugeben, das macht das ganze etwas langweilig.

Für interessantere Funktionalität benötigt man Zugriff auf die installierten Systembibliotheken. gForth enthält zwar eine Schnittstelle zu in C geschriebenen Bibliotheken, jedoch benötigt diese zur Laufzeit einen C-Compiler. Der NanoNote enthält jedoch keinen C-Compiler in der Standard-Firmware.

Das Ganze ist nicht so schlimm, wie es scheint. gForth bietet nämlich dennoch die Wörter `OPEN-LIB` und `LIB-SYM`, mittels derer man Bibliotheken laden kann und die Adressen der enthaltenen Funktionen ermittelt. Ein Beispiel:

```
s" libc.so.0" open-lib CONSTANT libc
s" sleep" libc lib-sym CONSTANT 'sleep
'sleep HEX . DECIMAL
-> 2AB8EA60  ok
```

Es fehlt dann lediglich ein Weg, diese Funktionen auch auszuführen. Diesen Mangel beheben wir, indem wir mit dem in gForth enthaltenen MIPS-Assembler ein Wort zum Aufruf von C-Funktionen ergänzen. Zum Verständnis folgt aber zunächst ein Abstecher zum gForth-Assembler für MIPS.

CODE für MIPS

Die MIPS-CPU hat einen extrem einfachen Befehlssatz. Die meisten CPU-Befehle haben jeweils 3 Register als Operanden. Der erste Operand ist das Ziel, gefolgt von zwei Quelloperanden. Einige Befehle benutzen statt einem Register eine konstante 16-Bit Zahl als zweiten Quelloperanden. In gForth werden sowohl Register als auch konstante Operanden ohne weiteren syntaktischen Zuckerguss als Zahlen notiert. Ob es sich um eine Konstante oder aber um ein Register handelt, entscheidet sich einzig und allein nach dem Namen des CPU-Befehls. Register 0 ist ein Dummy und liefert immer den Wert 0 bzw. verwirft alle Werte die hineingeschrieben werden.

Der Befehl `1 2 3 addu,` addiert Register 2 und 3 und speichert das Resultat in Register 1. Der Befehl `1 2 123 addiu,` addiert die Konstante 123 zum Register 2 und legt das Resultat in Register 1 ab.

Laden und Speichern erfolgt über die Befehle `lw,` bzw `sw,`, die nur einen Adressmodus kennen: der letzte Operand ist ein Register, das die Adresse enthält, der vorletzte Operand ist eine 16-Bit-Konstante, die zur Adresse hinzuaddiert wird. Der erste Operand ist für `lw,` das Ziel, für `sw,` die Quelle der Operation.

Sprungbefehle wie `jr,` haben die Eigenart, dass der nachfolgende Befehl erst noch ausgeführt wird, bevor tatsächlich gesprungen wird.

Grob vereinfacht gesagt, benutzt gForth einen Fadencodeinterpreter für die Ausführung von Forth-Code, und so sehen Assemblerdefinitionen so aus, wie man das auch aus anderen klassischen Forth-Systemen kennt: Argumente vom Stack holen, Rückgabewerte auf den Stack legen, danach der Programmcode für NEXT.

Hier z. B. eine `CODE`-Definition von `+` (funktioniert in `gforth-fast`)

```
CODE +  ( n1 n2 -- n3 )
 2   4 17  lw,      \ lade [SP+4] nach R2
 17  17  4  addiu,  \ Stackpointer inkrement
 21  2  21  addu,   \ addiere [SP+4] zum TOS
 22  0 16  lw,      \ NEXT: Lade IP
 22  jr,            \ .. springe zu [IP]
```

Aktualisierung der Firmware

Zunächst benötigen wir auf dem Host-PC die `xburst-tools` zum Hochladen der Firmware. Fertige Ubuntu/Debian Pakete findet man hier [1], ebenso Tar-Files für andere Linux-Varianten.

Danach startet man den Nanonote im USB-Modus durch Drücken der Taste "u" beim Startvorgang (am sichersten ist es, Power+"u" gleichzeitig einige Sekunden gedrückt zu halten). Der Bildschirm sollte daraufhin schwarz bleiben, der normale Boot-Vorgang findet nicht statt.

Zum Herunterladen der neusten Firmware aus dem Netz und dem darauf folgenden Hochladen auf den Nanonote verwendet man am besten das Shell-Skript `reflash_ben.sh`:

```
$ wget http://downloads.qi-hardware.com/\
  software/images/NanoNote/Ben/reflash_ben.sh
$ chmod +x reflash_ben.sh
$ ./reflash_ben.sh latest
```

Um genau dieselbe Firmware-Version zu verwenden, die diesem Artikel zugrunde liegt, ersetze man `latest` durch `2010-11-17`. Nun benötigt man noch einige Geduld, bis der Schreibvorgang abgeschlossen ist.

Warnhinweise: Der USB—Modus des NanoNote ist nicht sehr fehlertolerant, weswegen man für eine sichere USB-Verbindung sorgen muss. Es sollte kein USB—Hub verwendet werden und man sollte das hochwertige USB-Kabel benutzen, mit dem der NanoNote ausgeliefert wurde. Standardmäßig schreibt das `reflash_ben.sh` Skript auch den Boot-Loader neu. Falls dabei etwas schiefgeht, ist der NanoNote danach zunächst nicht mehr boot-fähig, nichteinmal den USB—Modus kann man mehr aktivieren. Ich empfehle daher, `reflash_ben.sh` so zu modifizieren, dass der Boot-Loader nicht geschrieben wird. Das erreicht man durch folgendes Kommando:

```
$ sed -ie 's/B="TRUE"/B="FALSE"/g' \
   reflash_ben.sh
```

Zu weiteren Details zur Firmware-Aktualisierung sei auf [2] verwiesen. Bei Problemen lohnt es sich auch immer, auf der NanoNote-Mailingliste [3] nachzufragen.

```
 16  16  4  addiu,  \ .. inkrementiere IP um 4
END-CODE
```

Die schlechte Nachricht ist jedoch, dass gForth in C geschrieben ist, damit die Zuordnung der Forth—Register zu Maschinenregistern automatisch vom C-Compiler vorgenommen wurde und je nach Version variiert. Das eben gegebene Beispiel für + funktioniert deswegen nur mit `gforth-fast`, und nicht mit `gforth`. Auch wird es eventuell nicht in neueren gForth Versionen funktionieren.

Seit Neustem gibt es in gForth deswegen eine alternative Möglichkeit, Maschinencodedefinitionen zu schreiben, die mit einer standardisierten Registerbelegung aufgerufen werden. Benutzt wird dabei die Konvention für Funktionsaufrufe des C-Compilers. Das heißt, der Maschinencode, den man schreibt, entspricht einer C-Funktion mit dem Prototypen

```
void *funktion(void *sp);
```

Der Forth-Stackpointer geht als Argument herein; zurückgeliefert wird der neue Wert des Stackpointers[1].

Definitionen dieser Art werden mit `ABI-CODE` statt `CODE` eingeleitet. Dazu muss man jetzt nur noch wissen, dass bei C-Funktionsaufrufen auf MIPS-CPUs, die ersten 4 Argumente in Registern 4–7 übergeben werden und für Rückgabewerte die Register 2–3 dienen. Als temporäre Register, die nicht gesondert gesichert werden müssen, sind Register 8–15 ausgewiesen. Die Rücksprungadresse nach Funktionsaufrufen liegt in Register 31, Register 29 enthält den C-Stackpointer.

Eine portable Definition für + sieht somit wie folgt aus:

```
ABI-CODE +  ( n1 n2 -- n3 )
 8  4 4  lw,     \ lade [SP+4] nach R8
 9  0 4  lw,     \ lade [SP] nach R9
 8  8  9  addu,  \ R8 = R8+R9
 8  4 4  sw,     \ Resultat in [SP+4] speichern
 31  jr,         \ Funktions-Return
 2  4  4  addiu, \ SP+4 als neuer SP zurück
END-CODE
```

Aufruf von Bibliotheksfunktionen

Nun können wir versuchen, mit einem einfachen Maschinencode-Wort eine Brücke zu Bibliotheksfunktionen zu bauen. Als ersten Versuch rufen wir die Funktion `sleep` auf. Ein Argument wird übergeben: die Zeit, die `sleep` schlafen soll. Auf der Forth-Seite übergeben wir dieses Argument und dazu noch die Adresse der Funktion über den Forth-Stack. Das Argument laden wir in Register 4, die Adresse gehört in Register 25, bevor wir die Funktion anspringen. Register 25 zu verwenden, ist wichtig, da Bibliotheksfunktionen aus positionsunabhängigen Maschinencode bestehen, der unter der Annahme kompiliert wurde, dass die Adresse der aktuellen Funktion immer in Register 25 vorzufinden ist.

Unser Brückenwort sieht also so aus:

```
ABI-CODE call1   ( x fn-addr -- )
 29  29  -8  addiu,
 31   0 29  sw,  \ Rücksprungadresse sichern
 4    4 29  sw,  \ Forth-SP sichern
 25  0 4  lw,   \ lade [SP] nach R25
 4   4 4  lw,   \ lade [SP+4] nach R4
 31  25  jalr,  \ Funktionsaufruf
 nop,
 31   0 29  lw,  \ widerherstellen R31, R4
 4    4 29  lw,
 29  29 8  addiu,
```

[1] Das ist nur die halbe Wahrheit. In Wirklichkeit ist der Funktions-Prototyp `void *funktion(void *sp, double **fp);` und ermöglicht damit auch Zugriff auf den Fließkommazahlenstack. Da uns Fließkomma aber hier nicht interessiert, wollen wir uns von derartigen Details nicht ablenken lassen.

```
 31  jr,          \ Funktions-Return
 2    4  8 addiu,  \ SP+8 zurück
END-CODE
```

Wir testen, ob es funktioniert:

```
10 'sleep call1
```

tatsächlich: die Ausführung benötigt genau 10 Sekunden.

Das Ganze lässt sich auf Funktionen mit Rückgabewerten und beliebig vielen Argumenten verallgemeinern. Listing `funcall.fs` enthält den entsprechenden Code für mehrere Brückenwörter, die verschiedene Argumentanzahlen unterstützen. Nun können wir z.B. vereinfacht schreiben:

```
s" printf" libc lib-sym CONSTANT 'printf
s\" Hello World %i\0" DROP 123
'printf void(2xint)
-> Hello World 123 ok
```

Wir benutzen hierbei `S\"..\0" DROP` statt `S""`, da C Null-terminierte Strings erwartet.

Ein bisschen Zuckerguss

Um nicht für jede C-Funktion die Adresse erst mit `LIB-SYM` zu ermitteln, dann für jeden Aufruf das richtige Brückenwort `void(int)` etc. benutzen zu müssen, schaffen wir uns eine Reihe von import-Wörtern.

Diese ermitteln die Adresse für eine C-Funktion, und definieren ein Forth-Wort gleichen Namens, das den Aufruf per Brückenwort übernimmt. Der entsprechende Code findet sich im Listing `import.fs`. Wir können jetzt ganz kompakt schreiben:

```
s" libc.so.0" open-lib current-lib !
void(int): sleep
void(2xint): printf
1 sleep
-> ok
s\" hello world %i\0" DROP 123 printf
-> hello world 123 ok
```

Handwerkszeug

Jetzt importieren wir die grundlegenden Linux-Systemfunktionen, die notwendig sind, um auf das Grafik-Gerät zuzugreifen. Wir benötigen `fileno`, das zu einem mit `OPEN-FILE` generierten Dateihandle den zugehörigen Linux-Dateideskriptor ermittelt. Außerdem benutzen wir `mmap` zum Einblenden von Gerätespeicher in den gForth-Prozess, und `ioctl` um mit Linux-Gerätetreibern zu reden.

```
s" libc.so.0" open-lib current-lib !
int(int): fileno  ( fid -- fd )
int(3xint): ioctl  ( fd n1 x1 -- n2 )
int(6xint): mmap ( n1 fd a1 n2 x1 x2 -- a2 )
```

Dies und ein paar weitere Konstanten für die Nutzung von `mmap` finden sich im Listing von `linux.fs`

Es wird grafisch

Das Fundament ist gelegt. Jetzt können wir auf die Linux-Grafik-Gerätedatei `/dev/fb0` zugreifen. Das ist sogar noch einfacher als es klingt. Blenden wir mit `mmap` den zu `/dev/fb0` gehörenden Speicher in unseren Prozess ein, erhalten wir Zugriff auf den Videospeicher:

```
s" /dev/fb0" R/W OPEN-FILE THROW fileno
CONSTANT fb
0   fb   0   320 240 * 4 *
PROT_READ PROT_WRITE OR  MAP_SHARED
mmap  CONSTANT video-mem
```

`video-mem` zeigt jetzt auf den Anfang des Grafikspeichers. 4 Byte pro Pixel: ein ungenutztes Byte, dann rot, grün, blau. 320x240 Pixel, von links nach rechts und oben nach unten. Um den Bildschirm rot zu färben, geben wir ein:

```
: makered  ( -- )
   320 240 * 4 * 0 DO
      $FF0000 video-mem I + !
   4 +LOOP ;   makred
```

Es wird jedoch nicht alles Rot. Der Textcursor blinkt weiterhin und hinterlässt ein schwarzes Kästchen. Ein Aufruf von `ioctl` sagt dem Linux-Kernel, dass unsere Konsole im Grafikmodus laufen soll, und deaktiviert den störenden Cursor:

```
s" /dev/tty" R/W OPEN-FILE THROW fileno
CONSTANT console
console KDSETMODE KD_GRAPHICS ioctl  DROP
```

Die vollständigen Grafikroutinen finden sich im Listing `grafik.fs`. Falls wir per SSH-Login arbeiten, ersetzen wir `/dev/tty` mit `/dev/tty2` und drücken auf der NanoNote-Tastatur Ctrl+Alt+F2 um die zweite Konsole sichtbar zu machen.

Ein Apfelmännchen

Um die Grafik in Aktion zu erleben, wollen wir nun mit gForth ein Apfelmännchen (das Mandelbrot-Fraktal) auf den Bildschirm zaubern. Wie erwähnt, besitzt der NanoNote keine Fließkommaeinheit, und die Verwendung der Softwareemulation würde unser Programm extrem verlangsamen.

Stattdessen verwenden wir eine einfache Festkommaarithmetik. Wir benutzen dafür die 32-Bit Ganzzahlen von gForth, und rechnen so, als wäre in der Mitte das Komma gesetzt. Addition und Subtraktion funktionieren dann weiter wie gewohnt. Lediglich nach einer Multiplikation müssen wir das Komma um 16 Binärstellen korrigieren. Die Festkommamultiplikation sieht also so aus:

```
: fix*  ( fix1 fix2 -- fix3 )
   M* 16 lshift swap 16 rshift or ;
```

Das gesamte Apfelmännchen-Programm findet sich im Listing `mandelbr.fs`. Um das Ganze mit maximaler Geschwindigkeit darzustellen, nutzt man folgendes Kommando:

```
gforth-fast --dynamic ./mandelbr.fs
```

Das Ganze dauert nur 3 Sekunden, und hinterlässt ein Apfelmännchen samt blau-rot schimmernder Kontur auf dem Bildschirm. Bild 2 auf der nächsten Seite lässt erahnen, wie das in Natura aussieht.

Bild 2: Wir zaubern mit gForth ein Apfelmännchen auf den Nanonote-Bildschirm

Der Hardware an den Kragen

Ganz nebenbei haben wir uns mit den grafischen Spielereien auch die Grundlage für ernsthaftere Hardware-Basteleien mit dem NanoNote geschaffen. Unten im Batteriefach liegen ein paar Lötpunkte für I/O Leitungen, die direkt an der CPU hängen. Auch die Leitungen des Micro-SD Ports entspringen direkt der CPU und lassen sich zu generischen I/O Leitungen umfunktionieren.

Dazu benötigen wir Zugriff auf die entsprechenden Steuerregister der CPU, die alle auf *physikalischen* Speicheradressen liegen. Unter Linux sehen wir aus gForth heraus zunächst aber nur *logische* Adressen, mit `@` und `!` kommen wir also so einfach nicht weiter.

Der Trick ist, dass wir per `mmap` unter Verwendung der speziellen Gerätedatei `/dev/mem` dennoch beliebige physikalische Adressbereiche in den gForth Speicherbereich einblenden können. Ein Blick ins Datenblatt [4] offenbart, dass die für uns interessanten Register zur Steuerung der I/O Leitungen bei Adresse 10010000_h beginnen. An die Arbeit:

```
s" /dev/mem" R/W OPEN-FILE THROW fileno
CONSTANT mem
$10010000   mem  0  4096
PROT_READ PROT_WRITE OR   MAP_SHARED
mmap CONSTANT gpio-base
```

Die Adressen ab `gpio-base` referenzieren jetzt den gefragten physikalischen Speicher. Zum Test lesen wir den Status der I/O Pins, an der die Tastatur hängt. Laut Schaltplänen [5, 6] liegen diese an Port D, an Adresse `gpio-base` + 300_h:

```
: test  BEGIN
   gpio-base $300 + @ hex. 10 ms
  AGAIN ;   test  \ Abbruch mit Ctrl+C
```

Drücken wir ein paar Tasten, können wir beobachten, wie der Pin-Status sich entsprechend ändert.

Etwas komplizierter ist es, den Pegel von Pins aktiv zu setzen. Dafür müssen so einige Hardware-Register gesetzt werden. Fertige Routinen finden sich im Listing `gpio.fs`. Testweise erwecken wir damit den eingebauten Buzzer zum Leben, der mit Pin 27 an Port D angesteuert wird (übrigens nicht zu verwechseln mit dem ebenfalls vorhandenen Lautsprecher):

```
io-init
27 #output port-D io-direction
: click  27 port-D io-0pin! ;
: clack  27 port-D io-1pin! ;
: buzz  BEGIN click 1 ms clack 1 ms AGAIN ;
buzz   \ Abbruch mit Ctrl+C
```

Das Ganze erinnert an die Verwendung des PC-Parallelports zur Hardwaresteuerung. Allerdings verwenden die I/O-Pins des NanoNote 3-V-Pegel, und sind um mehrere Größenordnungen schneller als ein Parallelport. Auf der Mailingliste [3] gibt es Berichte, nach denen bitweise mehrere MBit/s zu externer Hardware übertragen wurden. Wer weiß, was man so mit gForth noch alles zusammengebaut bekommt.

Links

[1] *Xburst-Tools zum Brennen der Firmware*, `http://projects.qi-hardware.com/index.php/p/xburst-tools/downloads/`

[2] *Informationen zur Firmware und zum Firmware Upgrade*, http://en.qi-hardware.com/wiki/Official_Software_Image

[3] *Mailingliste zur Diskussion über den NanoNote*, http://lists.en.qi-hardware.com/mailman/listinfo/discussion

[4] *Vollständige Dokumentation des System-On-Chip im Nanonote*, http://www.gmun.unal.edu.co/cicamargoba/embebidos/Jz4725_pm.pdf

[5] *Schaltpläne des NanoNote (korrekterweise: Schaltpläne der noch nicht gebauten nächsten Hardwarerevision)* http://downloads.qi-hardware.com/hardware/qi_avt2/v1.0/orcad_sch/qi_avt2_v1.0.pdf

[6] *Belegung der I/O Leitungen im NanoNote* http://en.qi-hardware.com/wiki/Hardware/Ben#GPIO_pins

Bibliotheksfunktionen aufrufen — funcall.fs

```
1   \ Generischer Aufruf einer C-Funktion mit Adresse a-addr.
2   \
3   \ Übergabe von x1..x4 als Argumente der Funktion (in Registern).  Weitere
4   \ Argumente i*x kann die Funktion vom Forth-Stack beziehen, allerdings muss
5   \ man diese (und nur diese) in umgekehrter Reihenfolge auf den Stack legen,
6   \ und am Ende selbst wieder vom Stack entfernen.
7   \
8   \ Rückgabewerte der Funktion werden als x5 und x6 zurückgegeben.  Je nachdem,
9   \ ob und wieviel die Funktion zurückliefert, enthalten diese keine sinnreichen
10  \ Werte.  Z.B. liegen zurückgelieferte 'int' Werte nur in x5.
11  ABI-CODE funcall  ( i*x x1 x2 x3 x4 a-addr -- i*x x5 x6 )
12     29   29  -12  addiu,   \ Register 4, 31, 16 sichern
13     4   0 29  sw,
14     31   4 29  sw,
15     16   8 29  sw,
16
17     \ Tausche Stackpointer von C-Stack und Forth-Stack
18     \ Achtung Stackpointeroffset: Registerargumente in C reservieren auch
19     \ (ungenutzten) Platz auf dem Stack!
20     16   29   move,        \ C-Stackpointer in Register 16 sichern
21     29   4   4  addiu,     \ Forth-Stackpointer zu C-Stackpointer
22
23     25   0 4  lw,          \ a-addr laden
24     7   4 4  lw,           \ Argumentregister 5-7 laden
25     6   8 4  lw,
26     5   12 4  lw,
27
28     31   25  jalr,         \ Funktionsaufruf
29     4   16 4  lw,          \ und Argument 4 laden (Sprungverzögerung beachten!)
30
31     29   16  move,         \ C-Stackpointer wiederhestellen
32     4   0 29  lw,          \ Register 4, 31, 16 wiederherstellen
33     31   4 29  lw,
34     16   8 29  lw,
35     29   29  12  addiu,
36
37     2   16 4 sw,           \ Resultatregister 2, 3 auf Forth-Stack
38     3   12 4 sw,
39     31 jr,                 \ Funktions-Return
40     2 4 12 addiu,          \ neuer Forth-Stackpointer in Reg. 2 zurück
41  END-CODE
42
43  \ Funktionsaufrufe von C-Funktionen mit bestimmter Zahl an
44  \ Parametern/Rückgabewerten.  Achtung: auch hier müssen ab dem 5. Argument
45  \ alle Argumente in umgekehrter Reihenfolge übergeben werden!
46  : void(void)  >r 0 0 0 0 r>  funcall 2drop ;
47  : void(int)  >r 0 0 0 r>  funcall 2drop ;
48  : void(2xint)  >r 0 0 r>  funcall 2drop ;
49  : int(void)  >r 0 0 0 0 r>  funcall drop ;
50  : int(int)  >r 0 0 0 r>  funcall drop ;
51  : int(2xint)  >r 0 0 r>  funcall drop ;
52  : int(3xint)  >r 0 r>  funcall drop ;
53  : int(4xint)  funcall drop ;
```

```
54  : int(6xint)  funcall drop nip nip ;
```

Bibliotheksfunktionen nach Forth importieren — `import.fs`

```
1   require ./funcall.fs
2
3   VARIABLE current-lib  \ Vom Benutzer zu setzen: <string> open-lib current-lib !
4
5   \ Funktionsaddresse in Forth-Wort gleichen Namens speichern
6   : c-import:  ( "name" -- )
7      >in @ create >in !    \ Wort "name" erzeugen, aber "name" nicht konsumieren
8      parse-name            \ "name" nochmal als String auf den Stack legen
9      current-lib @ lib-sym  \ und Addresse der Funktion "name" ermitteln
10     dup 0= ABORT" symbol not found"  \ hat es geklappt?
11     , ;         \ Addresse der Funktion im Datenbereich von "name" ablegen
12
13  \ Definiere Wörter, die c-import aufrufen, und 'xt' benutzen um die
14  \ importierten Funktionen aufzurufen
15  : c-importer:  ( xt "name" -- )
16     CREATE ,   DOES>  c-import: @ ,  DOES> dup @ swap cell + PERFORM ;
17
18  \ Ein Import-Wort pro Anzahl Parameter/Rückgabewerte
19  ' void(void) c-importer: void(void):
20  ' void(int) c-importer: void(int):
21  ' void(2xint) c-importer: void(2xint):
22  ' int(void) c-importer: int(void):
23  ' int(int) c-importer: int(int):
24  ' int(2xint) c-importer: int(2xint):
25  ' int(3xint) c-importer: int(3xint):
26  ' int(6xint) c-importer: int(6xint):
```

Import von Linux Systemfunktionen — `linux.fs`

```
1   require ./import.fs
2
3   s" libc.so.0" open-lib current-lib !
4   int(int): fileno  ( fid -- deskriptor )
5   int(3xint): ioctl  ( deskriptor n1 x1 -- n2 )
6   int(6xint): mmap  ( n1 deskriptor a-addr1 n2 x1 x2 -- a-addr2 )
7
8   \ Konstanten zur Verwendung mit 'mmap'
9   0 Constant MAP_FILE   1 Constant MAP_SHARED   2 Constant MAP_PRIVATE
10  1 Constant PROT_READ  2 Constant PROT_WRITE   4 Constant PROT_EXEC
11
12  \ Konstanten zur Konfiguration der Konsole mit ioctl
13  $4B3A Constant KDSETMODE
14  0 Constant KD_TEXT    1 Constant KD_GRAPHICS  4 Constant KD_TRANSPARENT
```

Zugriff auf den Grafikspeicher — `grafik.fs`

```
1   require ./linux.fs
2
3   0 VALUE fb          \ Filedeskriptor für Grafikgerät
4   0 VALUE console     \ Filedeskriptor für Konsolengerät
5   0 VALUE video-mem   \ Zeiger zum Grafikspeicher (von +GRAPHICS gesetzt)
6
7   : +grafik ( -- )  \ Grafikmodus initialisieren;
8      \ Daraufhin ist der Grafikspeicher über video-mem adressierbar
9      s" /dev/fb0" R/W OPEN-FILE THROW fileno TO fb
10     0   fb   0   320 240 * 4 *
11     PROT_READ PROT_WRITE OR   MAP_SHARED   mmap TO video-mem
12     video-mem 0< ABORT" mmap fehlgeschlagen!"
13     s" /dev/tty" R/W OPEN-FILE THROW fileno TO console
14     console KDSETMODE KD_GRAPHICS ioctl
15     0< ABORT" ioctl KDSETMODE fehlgeschlagen!" ;
16  : -grafik  ( -- ) \ Zurück zum Textmodus
17     console KDSETMODE KD_TEXT ioctl
```

```
18      0< ABORT" ioctl KDSETMODE fehlgeschlagen!" ;
```

Zeichnen eines Apfelmännchens — mandelbr.fs

```
1   require ./grafik.fs
2
3   : fix*  ( fix1 fix2 -- fix3 )
4      \ Festkomma-Multiplikation; Komma liegt zwischen Bit 15 und 16.
5      M* 16 lshift swap 16 rshift or ;
6
7   : mandel-pixel  ( xc yc N -- iter )
8      \ Berechne ein Pixel des Mandelbrot-Fraktals am Punkt (xc,yc) mit bis zu N
9      \ Iterationen; Rückgabewert ist -1 für Punkte, die zur Mandelbrot-Menge
10     \ gehören, ansonsten liefern wir die Iteration zurück, ab der die Reihe
11     \ divergiert
12     { xc yc N }  \ Parameter in lokale Variablen übernehmen
13     0 0
14     N 0 do  ( s: x y )
15        over dup fix*  over dup fix*   ( s: x y x^2 y^2 )
16        2dup +  $40000 > if  2drop 2drop I unloop exit  then
17        - xc +   ( s: x y x_new )
18        -rot  fix* 2*  yc +  ( s: x_new y_new )
19     loop
20     2drop  -1 ;
21
22  \ Der Bereich des Mandelbrot-Fraktals, der dargestellt wird.
23  \ Alles Festkommazahlen in hexadezimal
24  -$18000 value x0     \ = -1.5
25   $08000 value x1     \ =  0.5
26  -$0C000 value y0     \ = -0.75
27   $0C000 value y1     \ =  0.75
28
29  50 value N  \ Anzahl der Iterationen
30
31  : mandelbr  ( -- )  \ Mandelbrot-Fraktal berechnen und zeichnen (320x240 Punkte)
32     240 0 do
33        320 0 do
34           I  x1 x0 -  320 */  x0 +  ( x )   J  y1 y0 -  240 */  y0 +  ( y )
35           N mandel-pixel dup 0< if
36              drop $FFFFFF  \ weiße Farbe für Punkte in der Mandelbrot-Menge
37           else  \ ansonsten Farbe aus einem rot-blau Farbverlauf
38              dup 16 * 255 and 16 lshift  swap 100 swap -  255 and or
39           then  \ schreibe 4-byte Farbwert direkt in den Grafikspeicher:
40           video-mem  J 320 *  I +  4 * + !
41        loop
42     loop ;
43
44  +grafik mandelbr  \ Grafikmodus aktivieren und Fraktal zeichnen
45  EKEY DROP         \ Auf Eingabe warten, danach Grafikmodus und gForth beenden
46  -grafik  BYE
```

Zugriff auf die I/O Ports — gpio.fs

```
1   require ./linux.fs
2
3   $10010000 Constant IO-HWBASE  \ physikalische Basis-Adresse
4   0 Value io-base  \ enstprechende logische Adresse (von IO-INIT gesetzt)
5   0 Value mem        \ Filedeskriptor der '/dev/mem' Gerätedatei
6
7   : io-init  ( -- ) \ Einblenden der HW-Register in den logischen Adressbereich
8      s" /dev/mem" R/W OPEN-FILE THROW fileno TO mem
9      IO-HWBASE mem 0 4096   \ 4096 bytes ab IO-HWBASE einblenden
10     PROT_READ PROT_WRITE OR MAP_SHARED mmap TO io-base ;
11
12  : io-port:  ( n "name" -- )  \ Definition v. Adresskonstanten relativ zu IO-BASE
13     CREATE ,  DOES>  ( a-addr1 -- a-addr2)  @ io-base + ;
```

```
14
15  $000 io-port: port-a    $100 io-port: port-b
16  $200 io-port: port-c    $300 io-port: port-d
17
18  struct   \ Datenstruktur der Register zu einem der IO-Ports A/B/C/D
19     cell% field PXPIN                ( PIN Level Register )
20     cell% field Reserved0
21     cell% field Reserved1
22     cell% field Reserved2
23     cell% field PXDAT                ( Port Data Register )
24     cell% field PXDATS               ( Port Data Set Register )
25     cell% field PXDATC               ( Port Data Clear Register )
26     cell% field Reserved3
27     cell% field PXIM                 ( Interrupt Mask Register )
28     cell% field PXIMS                ( Interrupt Mask Set Reg )
29     cell% field PXIMC                ( Interrupt Mask Clear Reg )
30     cell% field Reserved4
31     cell% field PXPE                 ( Pull Enable Register )
32     cell% field PXPES                ( Pull Enable Set Reg. )
33     cell% field PXPEC                ( Pull Enable Clear Reg. )
34     cell% field Reserved5
35     cell% field PXFUN                ( Function Register )
36     cell% field PXFUNS               ( Function Set Register )
37     cell% field PXFUNC               ( Function Clear Register )
38     cell% field Reserved6
39     cell% field PXSEL                ( Select Register )
40     cell% field PXSELS               ( Select Set Register )
41     cell% field PXSELC               ( Select Clear Register )
42     cell% field Reserved7
43     cell% field PXDIR                ( Direction Register )
44     cell% field PXDIRS               ( Direction Set Register )
45     cell% field PXDIRC               ( Direction Clear Register )
46     cell% field Reserved8
47     cell% field PXTRG                ( Trigger Register )
48     cell% field PXTRGS               ( Trigger Set Register )
49     cell% field PXTRGC               ( Trigger Set Register )
50     cell% field Reserved9
51     cell% field PXFLG                ( Port Flag Register )
52     cell% field PXFLGC               ( Port Flag clear Register )
53  end-struct jz_pio%
54
55  \ Konstanten zur Programmierung der Richtung eines I/O Ports mit IO-DIRECTION
56  0 CONSTANT #input    1 CONSTANT #output
57
58  : io-direction  { n direction pio-addr -- }  \ Richtung eines Ports setzen
59     1 n lshift  \ Bit-Maske für Pin n
60     DUP pio-addr PXFUNC !  DUP  pio-addr PXSELC !  \ FUN/SEL auf 0
61     pio-addr direction if PXDIRS else PXDIRC then ! ;  \ Richtung setzen
62  : io-1pin! ( n pio-addr -- )  \ pin 'n' auf 1 setzen
63     1 rot lshift  swap  PXDATS ! ;
64  : io-0pin! ( n pio-addr -- )  \ pin 'n' auf 0 setzen
65     1 rot lshift  swap  PXDATC ! ;
66  : io-pin@  ( n pio-addr -- flag )  \ pin 'n' auslesen
67     PXPIN @  1 rot lshift AND 0<> ;
68
69  \ Vor Verwendung der io-* Routinen, erst IO-INIT aufrufen !
```

Von der virtuellen Forth–Maschine zum realen Forth–Prozessor

Willi Stricker

Zunächst sollen einige grundsätzliche Überlegungen zum Aufbau des Forth–Systems und zur virtuellen Forth–Maschine angestellt werden:

Der Aufbau des Forth–Systems

Das Forth–System besteht bekanntlich im Gegensatz zu anderen Programmiersprachen aus mehreren Komponenten:

- Virtuelle Forth–Maschine (Forth Virtual Machine)
- Programmiersprache
- Betriebssystem

Die virtuelle Forth–Maschine

In der Praxis wird die virtuelle Forth–Maschine aufbauend auf einem beliebigen Mikroprozessor per Software erzeugt. Sie ist, zumindest theoretisch, für alle Mikroprozessoren identisch und bildet eine einheitliche Schnittstelle zwischen der Hardware (unterlagertem Mikroprozessor) und der darauf aufbauenden Programmiersprache Forth.

Die Kombination der individuellen Mikroprozessor–Hardware mit der Software–erzeugten virtuellen Forth–Maschine soll im folgenden als *virtueller Forth–Prozessor* bezeichnet werden.

Der reale Forth–Prozessor

Es ist naheliegend, die Kombination aus (Mikro–) Prozessor und Software und damit den virtuellen Forth–Prozessor ausschließlich in Hardware ohne zusätzliche Software direkt zu erstellen, etwa mit Hilfe der heute verfügbaren programmierbaren Logik–Bauelemente (z. B. FPGAs). Das wäre dann ein realer Forth–Prozessor. Um dessen Eigenschaften zu ermitteln, müssen zunächst die Unterschiede zwischen der virtuellen Forth–Maschine und einem *normalen* Mikroprozessor üblicher Bauart und Struktur betrachtet werden. Es sind im Wesentlichen 2 Merkmale:

1. Anstelle der üblichen Register gibt es einen Parameter–Stack.
2. Jeder Befehl ist eine Adresse (und umgekehrt).

Der virtuelle Forth–Prozessor und damit das Forth–Software–System kennt 2 Arten von Befehlen:

1. **Primitives:**
   Befehle, die in Assembler–Code geschrieben sind. Sie entsprechen den Maschinen–Befehlen (Assembler–Befehlen) bei einem üblichen Prozessor
2. **High–Level–Befehle:**
   Befehle, die in Forth geschrieben sind (aus Primitives zusammengesetzte Befehle). Sie entsprechen den Unterprogrammen bei üblichen Prozessoren.

Wie oben erwähnt, sind beide Befehlsarten im Forth–System durch Adressen im Programm gekennzeichnet und werden durch ihre Adressen aufgerufen.

Die virtuelle Forth–Maschine enthält einen *inneren Interpreter*, meist als *NEXT-Routine* bezeichnet. Er dient als *Programm–Weiterschalter* für beide Befehlsarten. Zur Unterscheidung benötigen die Befehle deswegen einen Befehls–Vorsatz (Instruction Prefix) in Form einer (Unter–) Programm–Adresse. Bei Primitives ist der Befehls–Vorsatz die Adresse des eigenen (Assembler–) Programms, bei High–Level–Befehlen ist er die Adresse eines kleinen Unterprogramms (meist als *DOCOL-Routine* bezeichnet), das anstelle eines Call–Befehls das Speichern der Return–Adresse auf dem (Return–) Stack übernimmt. Die High–Level–Befehle werden durch einen Return–Befehl (`;S` = *SEMIS–Routine*) abgeschlossen, der den Rücksprung ins aufrufende Programm bewirkt, d. h. das Rückladen der Return–Adresse vom (Return–) Stack und den Rücksprung auf diese Adresse.

Eine Besonderheit folgt daraus: **Es gibt keinen expliziten Call–Befehl!**

Die virtuelle Forth–Maschine macht keinerlei Aussagen über Hardware–Eigenschaften. Das betrifft insbesondere das Interrupt–System. In der Praxis wird hierbei auf die Hardware des benutzten (Mikro)–Prozessors zurückgegriffen. Das gilt letztlich aber auch für alle gängigen Programmiersprachen.

Forderungen an einen realen Forth–Prozessor

Der reale Forth–Prozessor soll so weit wie möglich die gleichen Eigenschaften besitzen, wie der virtuelle Forth–Prozessor. Außerdem muss er über die notwendigen Hardware–Eigenschaften eines Mikro–Prozessors verfügen.

Praktische Realisierung eines realen Forth–Prozessors

Im Folgenden soll eine Realisierungsmöglichkeit gezeigt werden, die sich möglichst nah an die vorgenannten Forderungen hält.

Er erhält die Bezeichnung: **STRIP** (**ST**ack **R**elated **I**nstructions **P**rocessor)

Er besitzt die folgende Hardware–Ausstattung:

- Je einen separaten Parameter– und Return–Stack,
- ein Interrupt Interface,
- die Möglichkei interne und/oder externe Speicher anzuschließen
- die Möglichkeit interne und/oder externe I/Os anzuschließen

Er benötigt lediglich 3 Pointer–Register:

1. Parameter–Stack–Pointer: **SP**
2. Return–Stack–Pointer: **RP**
3. Instruction–Pointer: **IP**

Darüber hinaus besitzt er keine weiteren Pointer oder Register (insbesondere auch kein Flag–Register)!

Befehlssatz:

Es wird ein **Minimal–Befehlssatz** vorgesehen, der nach folgenden Kriterien ausgewählt wird:

- Er soll alle diejenigen Befehle enthalten, die für den Aufbau eines Forth–Systems zwingend erforderlich sind.
- Er soll zusätzliche Befehle enthalten, die für die Hardware–Steuerung erforderlich sind.

System–Befehle (Befehle, die nur vom Compiler benutzt werden):

`;S`	(->)	Return = pop address from return stack and store to IP
`LIT`	(-> data)	Load immediate data on stack
`BRANCH`	(->)	Branch to the following address
`?BRANCH`	(data ->)	Branch to the following address if data equals zero, else continue

Indirekter Befehls– und Unterprogramm–Aufruf

`EXECUTE`	(address ->)	Execute address (on top of stack)

Zugriff auf den Parameter– und den Return–Stackpointer:

`RP@`	(-> RP)	get RP
`RP!`	(RP ->)	store RP
`SP@`	(-> SP)	get SP
`SP!`	(SP ->)	store SP

Return–Stack manipulation:

`R>`	(-> data)	pop data from return stack
`>R`	(data ->)	push data to return stack

Parameter–Stack manipulation (wahlfreier Zugriff auf den Parameter–Stack):

`DROP`	(data ->)	drop data from stack
`PICK`	(position -> data)	load data from relative stack position
`-PICK`	(data position ->)	store data on relative stack position

Speicher–Zugriff:

`@`	(address -> data)	fetch data from memory address
`!`	(data address ->)	store data on memory address

Logische Funktionen:

`INVERT`	(data -> result)	bitwise NOT
`AND`	(data1 data2 -> result)	bitwise AND
`OR`	(data1 data2 -> result)	bitwise OR

Arithmetische Funktionen:

`+C`	(data1 data2 -> result carry) add data1 to data2 with sum and carry (lsb)
`U2/C`	(data -> carry result) shift right one bit, with result and carry (msb)

Spezielle Byte–Befehle:

`CSWAP`	(byte1|byte2 -> byte2|byte1) swap bytes in data
`C@`	(address -> 0|byte) fetch byte from address (upper byte = 0)
`C!`	(byte address ->) store byte to address (only lower byte, upper byte is discarded)

Prozessor Steuerungs–Befehle (Interrupt–Befehle)

`DISINT`	(->)	Disable Interrupts
`ENINT`	(->)	Enable Interrupts

In der Summe sind das lediglich 26 Befehle! Mit ihnen lässt sich gemäß obiger Vorgabe ein vollständiges Forth–System aufbauen (siehe dazu den Artikel in der VD Nr.3/2009). Hinweis: Einige der Befehle sind in Forth nicht definiert oder nicht üblich.

Für den realen Forth–Prozessor ergeben sich aus der Hardware–Realisierung zwei Besonderheiten gegenüber dem virtuellen Forth–Prozessor.

1. **Kein Befehls–Vorsatz**
   Wie vorher beschrieben, benötigt die virtuelle Forth–Maschine den Befehls–Vorsatz (Instruction Prefix, erste Adresse im Code Field), um durch die NEXT–Routine die Unterscheidung zwischen Primitives und High–Level–Befehlen vorzunehmen.

   Dieser Weg ist im realen Forth–Prozessor wenig sinnvoll, denn es wird zusätzlicher Speicherplatz benötigt (mehr Speicherbedarf), es ist ein zusätzlicher Speicherzugriff nötig (längere Laufzeit).

   Der reale Forth–Prozessor verzichtet deswegen auf den Befehls–Vorsatz, er muss dann aber die Befehlsart an der Programmadresse erkennen. Da die Primitives ohnehin keine realen Adressen besitzen, werden dafür Pseudo–Adressen festgelegt, für die ein reservierter Adressbereich vorgesehen wird. Dort können dann zwar keine High–Level–Programme liegen, wohl aber Speicher– oder IO–Adressen.

2. **Return–Bit**
   Jede Code–Adresse ist eine gerade Adresse (bei 16– oder 32–Bit–Systemen). Damit ist das ganz rechts stehende Bit (LSB = least significant bit) immer null! Folglich kann dieses Bit für eine zusätzliche Information benutzt werden, in diesem Fall als *Return–Bit*!

   Das Return–Bit bewirkt, dass zusätzlich zum gerade ausgeführten Befehl ein Return durchgeführt wird. Das gilt unabhängig davon, ob der Befehl ein Primitive– oder ein High–Level–Aufruf ist. In einem Unterprogramm (High–Level–Befehl) bekommt dann der letzte Befehl ein Return–Bit. Damit entfällt der explizite Return–Befehl.

Da es beim Programmieren jedoch Situationen gibt, die einen expliziten Return–Befehl erfordern, wird in dem vorher definierten Befehlssatz an dessen Stelle ein neuer Befehl eingefügt (NOP = no operation) und der Return wird durch einen NOP–Befehl mit Return–Bit ersetzt.

`NOP` (->) No Operation

Folgende Sonderfälle erfordern einen Return–Befehl (mit Adresse):

1. **Trivialfall:**
   Ein Unterprogramm muss mindestens einen Befehl enthalten, auch dann, wenn es nichts bewirkt. Es muss also mindestens der Return–Befehl vorhanden sein.
2. **Strukturen (Sprung–Befehle):**
   Ein Sprung–Befehl erfordert immer eine gültige Sprung–Adresse, an der zwangsläufig ein Befehl stehen muss. Wenn aber der Sprung an das Ende eines Unterprogramm erfolgen soll, ist dort kein Befehl mehr vorhanden, so dass dort ein NOP (mit Return–Bit) stehen muss.

Praktische Ausführung des STRIP–Forth–Prozessors

Oberstes Ziel bei der Realisierung des Prozessors ist selbstverständlich eine möglichst hohe Ablauf–Geschwindigkeit. Um dieses Ziel zu erreichen, wird soviel wie möglich parallel verarbeitet. Eingeschränkt wird der Programmablauf lediglich durch die Zugriffe auf externe Speicher. Diese bestimmen damit den Mindest–Zeitablauf. Im Idealfall sollten deswegen möglichst wenige Buszugriffe erfolgen.

Stack–Zugriff:

Die Stacks sind physikalisch getrennt vom Arbeitsspeicher untergebracht. Sie sind deswegen nur über die Stack–Befehle zugreifbar. Dadurch hat der Prozessor aber zeitgleich Zugriff auf beide Stacks und auf den Programm– oder Daten–Speicher.

Der STRIP–Kernel

Er enthält 3 Pointer, deren Steuerung, sowie die die gesamte Befehlsverarbeitung. Er benötigt ein Clock–Signal und besitzt Schnittstellen für den Parameter– und den Return–Stack sowie ein Interrupt–Interface, außerdem Daten– und Adressbusse für Speicher und Peripherie.

Der Kernel wird erweitert zu einem lauffähigen Prozessor durch Anfügen eines Clock–Elements, Parameter– und Return–Stack sowie eines Reset/Interrupt–Controllers.

Ein vollständiges lauffähiges STRIP–Forth–System benötigt dann noch RAM (bzw ROM) für Daten und Programme sowie eine Programmier– und Debug–Schnittstelle. Zusätzlich können nach Bedarf Ein– und Ausgabe–Elemente angefügt werden.

Zeitablauf (Timing)

Ein Clock–Zyklus ist identisch mit einem Buszugriff. Er besteht aus 2 Zuständen (States) S0 und S1:

1. S0 = Fetch State (Befehls–Zugriff)
2. S1 = Execute State (Befehls–Ausführung)

Wie eingangs erwähnt, ist der Kernel so konstruiert, dass die Befehle nur so viele Clock–Zyklen brauchen, wie sie Buszugriffe benötigen (Ausnahmen siehe später). Daraus folgt, dass der Bus (fast) immer im Betrieb ist. Es gibt deswegen keine Pipelines oder Ähnliches.

Der Unterprogramm–Aufruf und die meisten Primitives benötigen nur **einen Buszugriff**. Zwei Buszugriffe benötigen lediglich:

- Befehle, die einen Operanden nachladen müssen: `LIT`, `BRANCH`, `?BRANCH`
- Befehle, die einen Speicherzugriff vornehmen: `@`, `!`, `C@`, `C!`
- Befehle, die mit Return–Bit einen zweiten Zyklus benötigen: `R>` mit Return, `RP!` mit Return

Anmerkungen:

Befehle mit Operanden benötigen zwei Programm–Speicherplätze, Befehle mit Speicherzugriff dagegen nur einen. Enthält der Befehl ein Return–Bit, dann ist **kein zusätzlicher Buszugriff** nötig, d. h., der Return–Befehl benötigt keine zusätzliche Zeit! Ausnahmen sind lediglich die Befehle `RP!` und `R>`, sie benötigen dann wegen des doppelten Zugriffs auf den Return–Stack einen zweiten Zyklus für den Reset (ohne Buszugriff).

Befehlsablauf:

Standard–Befehle mit nur einem Buszugriff:

S0: Laden des Befehlscodes
S1: Ausführen des Befehls

Befehle mit einem Operanden (2 Buszugriffe):

1. Zyklus:
   S0: Laden des Befehlscodes
   S1: Interne Bearbeitung

2. Zyklus:
   S0: Laden des Operanden
   S1: Ausführen des Befehls

Befehle mit Speicherzugriff (2 Buszugriffe):

1. Zyklus:
   S0: Laden des Befehlscodes
   S1: Vorbereiten des Datentransfers

2. Zyklus:
   S0: Ausgabe der Speicheradresse
   S1: Ausführen des Datentransfers

Befehle `R>` und `RP!` mit Return–Bit (ein Buszugriff)

1. Zyklus:
   S0: Laden des Befehlscodes
   S1: Ausführung des Befehls

2. Zyklus: (Bus inaktiv)
   S0: keine Aktion
   S1: Ausführen des Return–Befehls

Pseudo–Primitive–Adressen

Bei der virtuellen Forth–Maschine liegen die Primitive–Befehle als Assembler–Code auf normalen Speicheradressen. Beim STRIP–Forth–Prozessor werden, wie vorher beschrieben, dafür *Pseudo–Adressen* definiert. Das sind Adressen, unter denen die Primitives aufgerufen werden. Diese Adressen stehen dann für High–Level–Befehle (Unterprogramme) nicht zur Verfügung, können aber für Speicher und I/O–Adressen benutzt werden. Die Pseudo–Primitive–Adressen werden deshalb in einen Bereich gelegt, der außerhalb des Programmspeichers liegt. Es ist sinnvoll, sie an den Anfang der Memory Map (bei Adresse 0 beginnend) zu legen.

Bei den 26 benötigten Basis–Befehlen werden entsprechend 26 Adressen belegt, bei einem 16–Bit–System demgemäß 52 Bytes. In der Praxis werden dafür 64 (2^6) Bytes reserviert.

Restart und Interrupts

Für den Restart wird im Programmspeicher ein Speicherplatz definiert. Er enthält die Restart–Adresse. Für die Interrupts wird ebenfalls je ein Speicherplatz reserviert. Er enthält jeweils die Startadresse des zugehörigen Interrupt–Service–Programms.

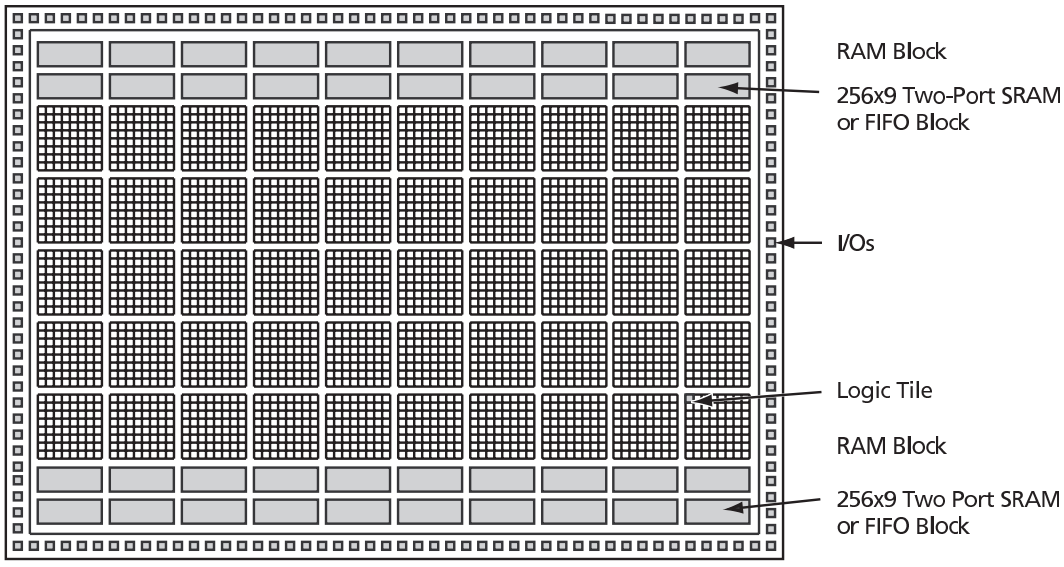
Der Restart ist ein durch Hardware ausgelöster Primitive–Befehl. Er benötigt einen Buszugriff (Laden der Restart–Adresse) und damit einen Clock–Zyklus.

Ein Interrupt wird hardwareseitig asynchron ausgelöst. Er wird vom Kernel in S0 abgefragt (falls es nicht der 2. Zyklus eines 2–Zyklen–Befehls ist) und muss dann den gleichfalls eingelesenen Befehl abwarten (ein oder 2 Clock–Zyklen), danach wird das zugehörige Interrupt–Programm ausgeführt.

Schlussbemerkung zur praktischen Realisierung

Der STRIP–Forth–Prozessor wurde zunächst in einem FPGA der Firma Actel auf einem Eval–Kit programmiert (Typ APA 075). Der Baustein enthält 3075 Tiles (kleine Logikeinheiten mit 3 Eingängen und einem Ausgang), dazu RAM–Blöcke mit insgesamt 3 kByte RAM. Das reicht gerade für die experimentelle Realisierung eines STRIP–Systems als 16–Bit–Version zur erfolgreichen Funktionsprüfung. Ein STRIP–Forth–System auf einem größeren Board mit RAM und Flash–ROM ist in Arbeit.

Eine detaillierte Beschreibung für einzelne Baugruppen, wie etwa Hardware–Komponenten, die Interrupt–Organisation oder das Bootprogramm für die Programmier– und Debug–Schnittstelle, soll in weiteren Artikeln folgen.



Architektur der Aktel APA-FPGAs

Quelle: Actel

Forth–Gruppen regional

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

München
Bernd Paysan
Tel.: (0 89) – 46 22 14 91 (p)
`bernd.paysan@gmx.de`
Treffen: Jeden 4. Donnerstag im Monat um 19:00, im Sommer (Mai–September) im Chilli Asia Dachauer Str. 151, im Winter im Sloveija Grill, Dachauer Str. 147, 80335 München.

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

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

Gruppengründungen, Kontakte

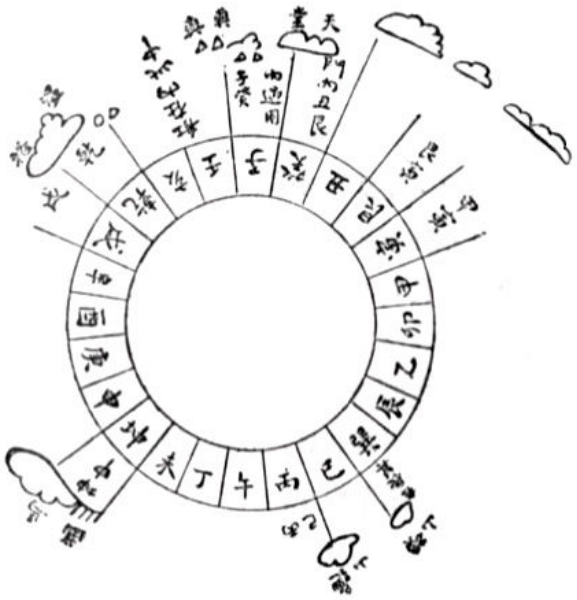
Hier könnte Ihre Adresse oder Ihre Rufnummer stehen — wenn Sie eine Forthgruppe gründen wollen.

µP–Controller Verleih

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

Spezielle Fachgebiete

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



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

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

Einladung zur

Forth–Tagung 2011
vom 15. bis 17. April 2011

im Bildungshaus Zeppelin
Zeppelinstraße 7, 38640 Goslar

http://www.bildungshaus-zeppelin.de

Geplantes Programm

Donnerstag, 14.04.2011
nachmittags Treffen der Frühankommer
Forth–200x–Standard–Treffen

Freitag, 15.04.2011
nachmittags Beginn der Forth–Tagung
Vorträge und Workshops

Samstag, 16.04.2011
vormittags Vorträge und Workshops
nachmittags Exkursion

Sonntag, 17.04.2011
09:00 Uhr Mitgliederversammlung
nachmittags Ende der Tagung

Anreise siehe: http://www.bildungshaus-zeppelin.de/Anreise.156.0.html

Die Innenstadt von Goslar

Quellen: www.bildungshaus-zeppelin.de, wikimedia.org