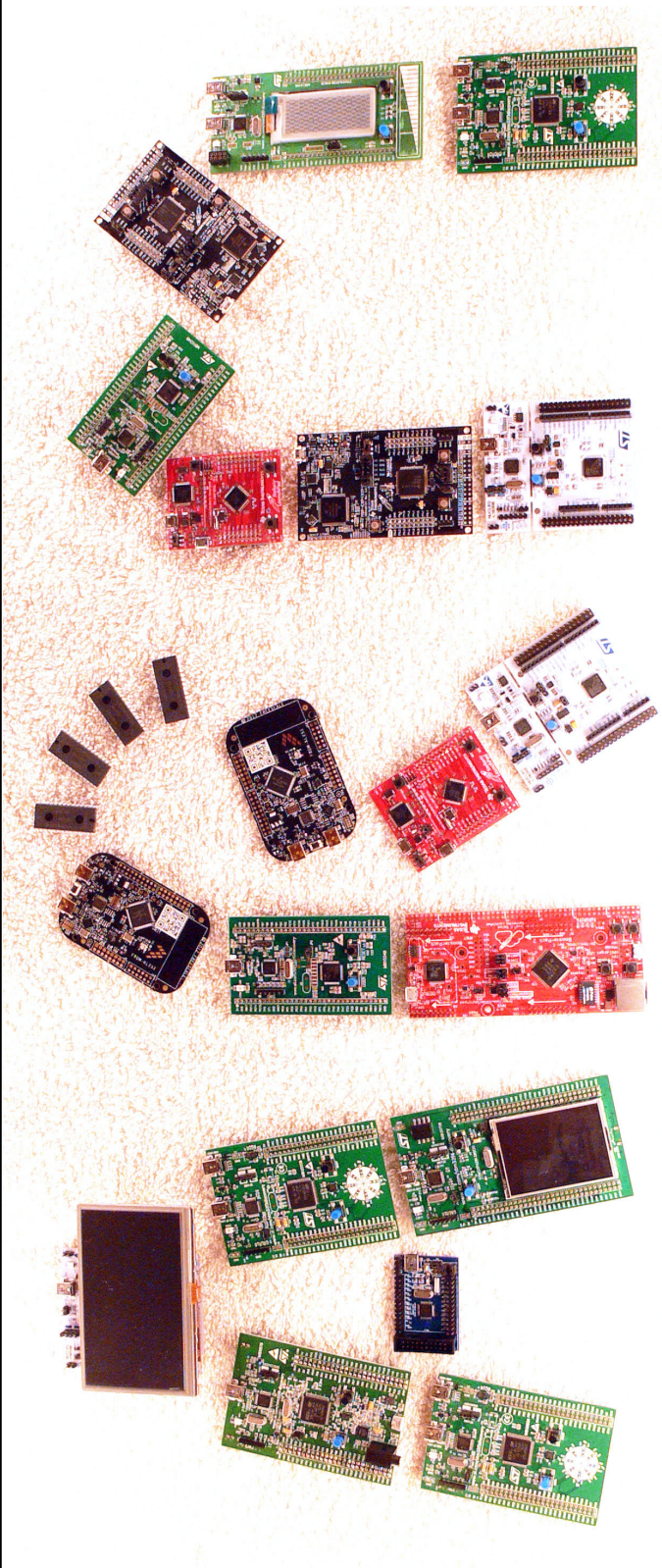
Vierte Dimension

2015–arm

Das Forth–Magazin

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



In dieser Ausgabe:

Dienstleistungen und Produkte fördernder Mitglieder des Vereins

tematik GmbH
Technische
Informatik

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

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

LEGO RCX–Verleih

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

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

Inhalt:

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

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

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

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

KIMA Echtzeitsysteme GmbH

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

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

FORTecH Software GmbH
Entwicklungsbüro Dr.–Ing. Egmont Woitzel

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

Ingenieurbüro
Klaus Kohl–Schöpe

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

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

Impressum

Name der Zeitschrift
Vierte Dimension

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

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

Anzeigenverwaltung
Büro der Herausgeberin

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

Erscheinungsweise
1 Ausgabe / Quartal

Einzelpreis
4,00€ + Porto u. Verpackung

Ein Vorwort

Im Sommer 2014 trafen wir zusammen, Matthias Koch und ich. Bei einer Tasse Tee und Gebäck im Antiquitäten-Cafe, das liegt etwas nördlich von Hannover, drehte sich alles um Forth und Matthias' *Mecrisp*, das auf dem MSP430G2553 lief. Und dabei kam die Idee auf, doch auch darüber mal ein Sonderheft des Forth–Magazins zu machen. 2007 gab es schon mal eins, da drehte sich alles um Atmels AVR–ATmega–MCUs. Das kleine *Butterfly* Evaluation Board war gerade in Mode bei den Makern und Vorbild für eingene kleine Platinen. Das Sonderheft haben damals fünf Autoren bestritten. Auch damals wurde Forth von der Pike auf beschrieben, und das gleich in zwei Varianten. Das ByteForth (Willem Ouwerkerk und Ron Minke) war für den ATmega8515, und das AmForth (Matthias Trute) war gleich für mehrere MCUs der ATmega-Familie zu haben. AmForth wurde populär und ist es immer noch, weil auch die ATmegas noch gerne genommen werden. Die ganzen Arduinos waren damit gemacht worden. Damals dachte ich, man könne ja auch ein Heft rund um den MSP430 machen, die verschiedenen Forths dafür vorstellen, es gab ja rasch gleich mehrere dafür: CamelForth, 4e4th, noForth und eben auch Mecrisp. Und ich wollte Matthias als einen der Autoren gewinnen. Doch es kam anders. Matthias Kochs Entwicklung ging bald zum Mecrisp-Stellaris weiter und es wurde immer deutlicher, dass es ein Sonderheft für ARM–MCUs werden würde. So hat er nun den Sommer 2015 über daran gearbeitet, Mecrisp zu verbessern und zu portieren auf alle möglichen ARM-basierte Plattformen und schließlich *alle* Beiträge des Heftes selbst verfasst. Statt Beiträge verschiedener Autoren, sind es nun eigentlich Kapitel einer Monografie geworden, und darum findet ihr unter den Kapiteln auch keine gesonderte Autoren erwähnt. Matthias Koch ist schnell, wenn es darum geht zu schreiben, und noch schneller, wenn es darum geht in *Assembler* zu schreiben, denn er *denkt* so flüssig in Assembler, wie unser einer in seiner Muttersprache. Und weil er um ein Vorwort bat, er könne sich ja schließlich nicht selbst dem Leser vorstellen, gibt es nun ein solches. Doch nun viel Vergnügen bei der Lektüre. Euer Michael.

Liebe Leser,

bestimmt haben die meisten von denen, die dieses Heft lesen, schon einmal einen kleineren Microcontroller wie einen Atmel AVR oder einen MSP430 von TI programmiert. Und wer es noch nicht getan hat, dem möchte ich es zuvor dringend empfehlen. Viele Grundzüge werden bekannt vorkommen oder tauchen leicht abgewandelt wieder auf. Wer allerdings schon weiß, worauf es ankommt, der weiß auch, wonach im Fall der Fälle zu suchen ist, und gerade dies ist bei den Microcontrollern aus der riesigen ARM–Cortex–Familie von großem Wert, denn die kleinen Käfer mit vielen Pins kommen mit einer zu Beginn fast unüberschaubaren Vielfalt von Möglichkeiten, Peripheriemodulen und Registern.

Doch auch wenn der Weg zuerst ein bisschen steinig wirken mag, soll niemand sich nach der Lektüre dieses Heftes entmutigt fühlen oder bei offenen Fragen verzagen, denn nachdem die ersten Brocken gemeistert sind, tut sich eine spannende Landschaft auf, die zu Entdecken viele interessante Bastelwochenenden füllen wird.

Matthias Koch

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

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

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

Wie alles anfing

Meine ersten Begegnungen mit Computern hatte ich schon früh, als ich am Commodore C64 meiner Eltern spielte — am liebsten Hover Bovver. Sobald ich in der Grundschule halbwegs Lesen gelernt hatte, brachte mir meine Mutter das Programmieren in Basic bei, was ich sehr viel lieber tat, als für Diktate zu üben. Einige Jahre später schenkte mir mein Onkel einen ausrangierten PC XT mit bernsteinfarbenem Monitor und Turbo Pascal 4, wovon ich total begeistert gewesen bin. Ein guter Freund zeigte mir, wie Leuchtdioden am Druckeranschluss angesteuert werden können, und schon war mein geliebter analoger Kosmos–Elektronik–Experimentierbaukasten mit der digitalen Welt verbunden.

Vom Z80 . . .

Mein erster tiefer Einstieg in die digitale Elektronik begann später, als ich als Schüler zum Dank für meine Dienste in einem Praktikum einen Z80–Einplatinencomputer geschenkt bekam, den ich mit einem EEPROM ausstattete (Kein UV–Löschen, sehr komfortabel!) und dessen Programme ich auf Papier entworfen und mit einem Hex–Editor getippt habe. Kurz darauf habe ich mir einen eigenen Z80–Computer ausgedacht, mit viel Mühe die vielen, vielen Leitungen zusammengelötet, mit Leuchtdioden an den Busleitungen und mit einem Taster als Taktquelle die Fehler gesucht und mein Werk recht bald mit einem Druckerport–Bootloader ausgestattet. Nachdem ich mir sicher war, den Prozessor verstanden zu haben, bin ich zum Komfort eines Assemblers übergewechselt, der mir das mühselige Berechnen von Sprungzieladressen im Kopf abnahm. Mein Ziel war es, ein räumliches Magnetometer zu bauen, um ein kleines Modellflugzeug bei Wind anhand der Richtung des Erdmagnetfeldes stabil zu halten. Sechs Hall–Sensoren, viele Operationsverstärker und AD–Wandler kamen zum Einsatz und die erste geglückte Messung der Richtung des Erdmagnetfeldes ließ mich jubeln. Leider wurde die Schaltung inklusive ihrer Stromversorgung so groß und schwer, dass ein Modellflugzeug, welches die Last hätte tragen können, schon von sich aus keine Probleme mit ein bisschen Wind gehabt hätte.

. . . zum MSP430

Eines Tages, mittlerweile als Physik–Student, wollte ich mich gern einmal mit Mikrocontrollern beschäftigen, stöberte in Datenblättern und wünschte mir vor allem einen schönen Befehlssatz, denn Assembler mochte und mag ich sehr. AVR und PIC, beide Familien unter Bastlern wohlbekannt, schieden der Prozessorarchitektur wegen aus. Mein Traum waren der legendäre 68000 und die großen DSP's von Analog Devices. . . Doch als ich dann zum ersten Mal so einen Chip vor mir liegen hatte, betrachtete ich erst die unzähligen winzigen Pins, dann meinen Lötkolben und wusste, dass ich nicht weit kommen würde. Ich nahm also ein bastelfreundliches Gehäuse mit in die Kriterienliste auf und entschied mich für den MSP430, der mir von der Architektur her sehr gefiel, wenngleich in der Open-Source-Welt noch kaum Unterstützung zu finden war.

Nachdem ich das JTAG–Protokoll verstanden und mein Flash–Werkzeug für Linux entwickelt hatte, näherte ich mich dem MSP430 unfreiwillig auf ähnliche Weise mit Hexeditor an, bis ich herausfand, dass es einen Fehler in dem von mir auf Linux verwendeten Open–Source–Crossassembler gab. Viele kleine Spielereien später war meine Ledcomm–Implementation in Assembler fertig und ich erinnerte mich an Forth, welches ich schon viel früher einmal über die Mailliste der Hamburger Minix–Gruppe kennengelernt habe, holte Starting Forth von Leo Brodie aus der Uni–Bibliothek, druckte mir ein FIG–Forth–Listing aus und begann zu lesen. . .

. . . und weiter über das Ur-Mecrisp

Das (bislang) nie veröffentlichte Ur–Mecrisp[1] lief übrigens mit Subroutine–Threading auf einem MSP430F1232 mit 8 kb Flash und 256 Bytes RAM, kompilierte nur ins RAM und hatte ein Ledcomm–Terminal. Es diente vor allem meiner Neugier und der gründlichen Untersuchung von Ledcomm. . . Mir machte Forth viel Spaß, und es folgte die Entwicklung für das Mecrisp, wie es nun bekannt ist. Zunächst musste ich die direkte Kompilation in Flash gefühlt neu erfinden, auch die Konstantenfaltung zog früh in Mecrisp ein. Vor der ersten Veröffentlichung hatte Mecrisp so bereits fast zwei Jahre Entwicklungszeit genossen.

Die Mühe, den Compiler so zu gestalten, dass dieser den Flash strikt als nur–einmal–beschreibbar betrachtet, sollte sich später bei der Entwicklung von Mecrisp–Stellaris auszeichnen. Es begann damit, dass Michael Kalus mich nach einer ARM–Portierung fragte — ich lehnte ab, mit der Begründung des in meinen Augen hässlichen Befehlssatzes und der für Forth denkbar unpassenden Load–Store–Architektur. Trotzdem hat Michael mir ein Stellaris–Launchpad mit der Post geschickt[2], welches bei mir anschließend in einer dunkelen Schublade verstaubte.

Mein MSP430–Abschieds–Projekt, wie es sich später herausstellen sollte, womit ich auch mein erstes großes Feuerwerk zündete, war die Bit–Bang–Implementation von USB auf dem MSP430G2452 und später sogar auf dem MSP430F2012. Dort kamen mir meine Kenntnisse sehr zu Gute: Opcodes, Taktzyklen und Fehlersuche mit dem Oszilloskop waren mir bestens vertraut. Für mich war es

[1] Mecrisp kommt von MSP und dem französischen „écris“ — im Sinne von „Du beschreibst den MSP430“.

[2] ... unterstützt von der Forth-Gesellschaft. mk

ein schönes Puzzle für die Weihnachtsferien… Anschließend las ich zum ersten Mal auf Hackaday von meinem eigenen Projekt!

… zum Mecrisp-Stellaris

Eines Tages, als ich als Biophysik–Doktorand im Labor bei der Messung von Chlorophyllfluoreszenz bereits sanft mit den Grenzen des MSP430 in Berührung kam, las ich von Loran–C, bekam das Navigationssignal auf Langwelle auch recht bald mit dem Oszilloskop zu Gesicht und hatte plötzlich das Bedürfnis nach einem schnellen AD–Wandler mit viel RAM. Mein erster Gedanke war, den AD–Wandler des MSP430 zu übertakten, was auch überraschend gut gelang; reichen tat es dennoch nicht, außerdem war und blieb der verfügbare Speicher vergleichsweise winzig. Einen AD–Wandler und Speicher direkt aneinanderlöten, Takt und Logik dazu? Nein, die Erinnerung an die Z80–Entwicklung aus Schultagen wog zu schwer. Und da fiel mein Blick auf das Stellaris–Launchpad. Wenn doch der Befehlssatz nicht gewesen wäre!

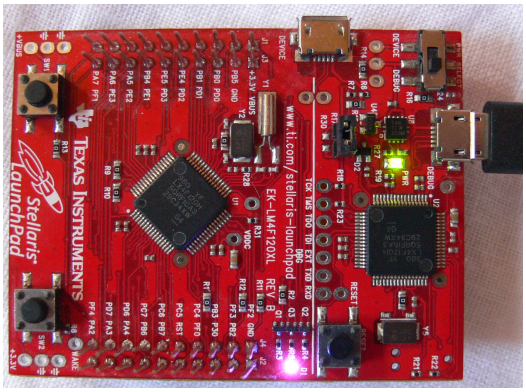


Abbildung 1: Damit ging es los — das TI Stellaris LaunchPad

Die grundlegenden Gedanken und die Anforderungen an Mecrisp–Stellaris waren somit von Anfang an klar: Harte Echtzeitanforderungen, interruptfest, digitale Signalverarbeitung on–the–fly. Und so entstand um den AD–Wandler und Speicher herum alles, was ich brauchte. Es dürfte den Leser nun gar nicht wundern, wieso das Discovery F4 die zweite Portierung geworden ist…

… und dem M0-Kern

Für mich war damit alles erreicht, was ich wollte, doch die Flut von E–Mails, die ich zu Mecrisp–Stellaris bekam, war überwältigend. Der wichtigste und häufigste Wunsch aus der weiten Welt war eine Portierung auf den M0–Kern. Ich selbst wäre nie darauf gekommen, jedoch schaffte vor allem Glen Worstell, der schon während der Entwicklung von Mecrisp mit mir schriftlich über Forth diskutierte, mich dazu zu überreden. Ein Wochenende hat es gedauert, und die M0–Portierung war einsatzbereit. Etwa ab diesem Zeitpunkt begann ich auch, den Quelltext so zu sortieren, dass weitere Portierungen leicht hinzugefügt werden können, was zu der Vielfalt an Portierungen geführt hat, die jetzt in diesem Heft zu bestaunen ist. Doch ganz besonders entscheidend sollte es sich erweisen, dass Glen mich seinem Freund John O. „Sandy“ Bumgarner vorstellte, einem Forth–Veteranen aus den Anfangstagen, der an der Entwicklung der Canon Cat mitgewirkt hat und mich mit so vielen kostbaren Ideen und Erfahrungen versorgt hat, die die Benutzerfreundlichkeit und Nützlichkeit von Mecrisp und Mecrisp–Stellaris ganz erheblich verbessert haben. Danke!

Über mich

Mit Gewächshaus und Riesenkaninchen lebe ich ein Stückchen nordwestlich von Hannover, habe Physik mit dem ungewöhnlichen Nebenfach Gartenbau studiert und bastele nun an meiner biophysikalischen Doktorarbeit zum Thema Laserspektroskopie an Algen. Besonders spannend finde ich Meeresbiologie, bin Apnoe–Taucher, übe Jonglage und bringe beim historischen Tanz alte, fast vergessene Tänze auf die Bühne, wirbele beim Salsa und Lindy hop über das Parkett, mag kochen sehr und sammele Obstbäume, Beerensträucher und andere Leckereien in meinem Garten.

Abbildung 2: … ich im historischen Studierzimmer

Links

http://forum.43oh.com/topic/2962-bit-bang-usb-on-msp430g2452/

http://www.minixhh.org

Eine kleine Einführung in die ARM Cortex M Architektur

Grundsätzlich beschreibt „ARM Cortex M“ eine gemeinsame Basis, zu der dann jeder Hersteller seine eigenen Erweiterungen hinzufügt und in einen Chip verwandelt — umgekehrt bedeutet dies, dass es nicht „den“ typischen ARM Cortex gibt, sondern viele, viele Varianten mit einem gemeinsamen Minimum. Zuallererst gehört dazu ein gemeinsamer Befehlssatz in verschiedenen Ausbaustufen.

M0 ist der kleinste, er besteht fast nur aus 16–Bit breiten Opcodes und kann von allen „M“s ausgeführt werden. Wer diese Familie in Assembler programmieren möchte, sollte unbedingt hier beginnen, denn er ist überschaubar und besteht aus nicht allzu vielen Befehlen. Die nächstgrößere Stufe heißt M3, hier werden viele 32 Bit breite Opcodes hinzugefügt, die die Arbeit von zwei oder mehreren M0–Opcodes in einem Takt vereinigen. Dazu kommen leistungsstarke Ganzzahl–Rechenoperationen. Wogegen der M0 nur eine $32 * 32 = 32$ Multiplikation besitzt, sind im M3 auch längere Multiplikationen und verschiedene Divisionen direkt in Hardware implementiert. Außerdem erlauben die längeren Opcodes auch das Einbringen von größeren Konstanten, was das Kompilieren von Forth in Maschinensprache sehr viel eleganter ermöglicht. Die maximale Ausbaustufe M4 schließlich fügt noch Gleitkomma–Operationen hinzu, die allerdings für Mecrisp–Stellaris nicht benötigt werden.

Auch für Forth–Programmierer wichtig zu wissen ist, dass es drei unterschiedliche Lese– und Schreibzugriffe in den Speicher mit unterschiedlicher Größe gibt — 8 Bits, 16 Bits und 32 Bits breit. Im M0 müssen diese stets auf „ausgerichtete“ Adressen treffen, während ein Bytezugriff also prinzipiell überall möglich ist, müssen 16–Bit–Zugriffe an geraden Adressen erfolgen und 32–Bit–Zugriffe erfordern „auf 4 gerade“ Adressen. Im M3 und M4 ist zusätzliche Logik vorhanden, die diese Einschränkung umschiffen kann, allerdings auf Kosten zusätzlicher Takte. Außerdem gibt es Peripherieregister, die eine bestimmte Zugriffsbreite erfordern oder sich je nachdem unterschiedlich verhalten.

Leider ist ARM Cortex eine „Load–Store–Architektur“ — das bedeutet, dass Rechen- oder Logikoperationen stets nur in Prozessorregistern stattfinden und alle Werte zuvor mit Lade- oder Speicherbefehlen umhergeschaufelt werden müssen. Dies bedeutet, dass es keine „atomaren“ Veränderungsbefehle gibt: Um ein Bit zu setzen, muss der alte Inhalt geholt werden, das Bit gesetzt und der Wert zurückgeschrieben werden, während Interrupts dazwischenkommen können. Es gibt einige Tricks wie Bit–Banding oder das BSRR–Register in Chips von STM, aber es gilt: Vorsicht ist geboten!

An dieser Stelle ist ein kleiner Einschub sicherlich hilfreich, denn das Suchen eines bestimmten Informations–Puzzleteils gestaltet sich mitunter schwierig. Grundsätzlich gilt: Für jeden Chip gibt es ein Datenblatt, in dem beispielsweise die Pinbelegung, eine Liste der enthaltenen Peripheriemodule und elektrische Daten gefunden werden können. Die Beschreibung der einzelnen Module ist meist in Form eines (manchmal chaotischen) „Familien–Referenzhandbuches“ gebündelt, worin zum Beispiel nach den Registern des AD–Wandlers gesucht werden kann. Und schließlich gibt es die übergeordnete Architektur–Spezifikation, die für alle Hersteller gleich ist und die außer des Befehlssatz (ARM Architecture Reference Manual Thumb–2 Supplement) noch generelle Richtlinien (ARM Architecture Reference Manual) und einige wenige, aber wichtige gemeinsame Peripheriebausteine wie den Interruptcontroller (NVIC) und einen Timer (Systick) beschreibt, die immer vorhanden sind[1]. Leider gehören weder ein Terminal noch eine Spezifikation für das Beschreiben des Flash–Speichers dazu...

An Adresse 0 beginnt immer die Vektortabelle, die den anfänglichen (Return–)Stackpointer, die Startadresse, die Adressen von Fehler–Handlern, die Adresse des Systick–Timer–Handlers und danach aller weiteren individuell vorhandenen Interruptvektoren enthält. Natürlich muss hier Flash vorhanden sein, damit der Chip nach einem Reset starten kann. An Adresse $20000000 beginnt das RAM. Hierhin kommt all das, was sich im Laufe der Zeit verändert — vor allem Stacks, Puffer, Variablen und die flüchtigen Teile des Dictionarys. Die individuelle Peripherie ist ab Adresse $40000000 zu finden, dort tauchen die Ports, AD–Wandler und vieles mehr auf. Von Adresse $E0000000 an sind allen Chips gemeinsame Register zu finden, dort wird beispielweise der Systick–Timer angesprochen.

Los gehts: Beim Start holt jeder ARM Cortex M Chip seinen (Return–)Stackpointer von Adresse 0, liest die Startadresse des Programmes aus Adresse 4 und beginnt, es auszuführen. Und schon sind wir alleine. Eins noch: Trotz des „Gerade–Gebotes" sind die Einsprungadressen und Rücksprungadressen auf dem Stack stets ungerade — dies signalisiert, dass es sich bei den Opcodes an der Adresse um den „Thumb“–Befehlssatz handelt, was bei ARM Cortex M Chips immer der Fall ist. Für die Neugierigen sei jedoch hinzugefügt, dass es einige noch größere Familien gibt, die noch einen anderen, ausschließlich

[1] Um den Einstieg zu erleichtern und die sonst langwierige Suche nach Handbüchern und Datenblättern abzukürzen, gibt es auf Sourceforge außer den Mecrisp–Stellaris–Paketen auch fertig gebündelte „Lesesammlungen“ für die meisten unterstützten Chips & Platinen! (Target literature package...)

32–Bit breiten Befehlssatz verwenden und anhand gerader/ungerader Adresse hin–und–her schalten.

[Anmerkung: Die Adressen der Instruktionen selbst im Speicher sind immer gerade. An dem letzten Bit der Adresse erkennt der Prozessor, ob es sich um Befehle im Thumb–Befehlssatz (dessen Befehle größtenteils 16 Bits breit sind, mit einigen 32 Bit breiten Erweiterungen) oder um den bei ganz großen ARM–Chips zusätzlich vorhandenen „klassischen“ ARM–Befehlssatz handelt, dessen Befehle alle stets 32 Bits breit sind. Um den Thumb–Befehlssatz auszuwählen, muss also im Programmzähler und bei den Rücksprungadressen auf dem Stack das letzte Bit gesetzt sein, was zu scheinbar „ungeraden“ Adressen führt. Darum kümmert sich Mecrisp–Stellaris jedoch von selbst, nur bei komplizierten Tricks mit dem Returnstack muss der Forth–Programmierer daran denken.]

Wer schon Erfahrung mit Microcontrollern hat, wird nun vermutlich im Datenblatt zum Kapitel „Digital–IO“ springen und versuchen, passende Bits zum Erstrahlen einer Leuchtdiode zu setzen. Aber halt, langsam! Wichtig, aber am Anfang meist nicht gleich ersichtlich sind die sogenannten „clock gating“ Register. Jedes Peripheriemodul, welches benutzt werden soll, muss zuvor in einem korrespondierenden „run mode clock gating register“ aktiviert und so mit Strom und Takt versorgt werden. Erst danach kann es losgehen. Dies ist im Hinterkopf zu behalten, denn Mecrisp–Stellaris aktiviert zwar standardmäßig beim Start die serielle Schnittstelle für das Terminal und der Bequemlichkeit halber gleich alle digitalen IO–Ports, wer jedoch den AD–Wandler oder etwas anderes benutzen möchte, muss sich selbst darum kümmern, möchte er keinen Absturz miterleben[2]. Denn im Gegensatz zu vielen anderen Microcontrollern werden beim ARM–Cortex Zugriffe auf nicht existente Speicherstellen oder Register nicht laufender Peripheriemodule abgefangen.

Erfrischend einfach ist der rückwärts zählende, 24 Bit breite Systick–Timer zu benutzen. Er ist immer da, braucht nicht aktiviert zu werden und ist oft sehr nützlich. Er besteht aus drei Registern:

```
$E000E010 constant Systick-Control
$E000E014 constant Systick-Reload
  \ Der Wert, der geladen wird,
  \ nachdem der Timer 0 erreicht hat
$E000E018 constant Systick-Current
  \ Der aktuelle Stand des Timers
```

Das Control–Register hat vier Bits, die interessant sind:

Bit16 ($10000): Dies Bit gibt beim Lesen eine 1, falls der Timer seit dem letzten Auslesen übergelaufen ist.

Bit02 ($00004): Genau den gleichen Takt nehmen wie der Prozessor. Falls es nicht gesetzt ist, wird eine andere, individuelle Taktquelle verwendet.

Bit01 ($00002): Erreichen von Null löst Interrupt aus.

Bit00 ($00001): Aktiv.

Wer Interrupts benutzen möchte, möge ansonsten daran denken, dass es nicht reicht, das entsprechende „Interrupt Enable“ Bit in Peripherieregister zu aktivieren, sondern auch ein übergeordnetes Bit im NVIC gesetzt werden muss… Wenn etwas partout nicht geht, obwohl alles stimmen sollte, empfiehlt sich ein prüfender Blick ins „Silicon Errata“, worin bekannte Hardware–Fehler versammelt sind.

Dictionarystruktur im Flash

Die klassische Dictionarystruktur, die vom neuesten Eintrag hin zu den älteren verlinkt und tief im Kern aufhört, hat den Nachteil, dass der Zeiger zur gerade aktuellen Definition an einem sicheren Platz zwischengespeichert werden muss, damit nach einem Reset der Anfang der Linkkette wiedergefunden werden kann. Genau dies ist jedoch im Flash, der nur begrenzt oft blockweise gelöscht werden kann, nicht sonderlich praktikabel.

In Mecrisp–Stellaris ist das Dictionary deshalb umgekehrt verlinkt. Im Flash beginnt die Suche bei den Kerndefinitionen und die Pointer zeigen jeweils auf die neueren Definitionen, bis in der aktuellen Definition ein Dictionary–Header mit einem (noch) leeren Link–Feld, welches auf das nächste Create wartet, das Ende signalisiert.

Am besten lässt sich dies anhand eines Beispieles veranschaulichen — wobei besonderes Augenmerk auf den Adress- und Linkfeldern liegt.

Zuerst kommt das Dictionary im RAM mit gewöhnlicher Linkreihenfolge, dessen Ende auf die erste Kerndefinition im Flash zeigt.

```
words
Address: 20002764 Link: 200026E8 Flags: 00000000 Code: 2000276E Name: see
Address: 200026E8 Link: 2000267C Flags: 00000000 Code: 200026F4 Name: seec
Address: 2000267C Link: 20002644 Flags: 00000000 Code: 2000268E Name: disasm-step
Address: 20002644 Link: 20000FC8 Flags: 00000000 Code: 20002654 Name: memstamp
[...]
```

[2] Ungültige Zugriffe lösen Fehler–Interrupts aus, die über irq–fault auch in Forth selbst behandelt werden können.

```
Address: 2000038C Link: 2000036C Flags: 00000000 Code: 200003A0 Name: disasm-fetch
Address: 2000036C Link: 20000330 Flags: 00000040 Code: 2000037C Name: disasm-$
Address: 20000330 Link: 000001EC Flags: 00000000 Code: 2000033C Name: list
```

Falls keine Definitionen im RAM vorhanden oder diese zum Kompilieren in Flash ausgeblendet sind, beginnt FIND einfach direkt im Kern.

Jetzt kommen die Kernroutinen, die nicht verändert oder gelöscht werden:

```
Address: 000001EC Link: 00000212 Flags: 0000FFFF Code: 00000212 Name: --- Mecrisp-Stellaris Core ---
Address: 00000212 Link: 0000022A Flags: 00000042 Code: 0000021E Name: 2dup
Address: 0000022A Link: 0000023C Flags: 00000062 Code: 00000236 Name: 2drop
Address: 0000023C Link: 0000025A Flags: 00000044 Code: 00000248 Name: 2swap
Address: 0000025A Link: 0000026E Flags: 00000062 Code: 00000266 Name: 2nip
Address: 0000026E Link: 00000288 Flags: 00000044 Code: 0000027A Name: 2over
[noch viele mehr...]
Address: 00003956 Link: 00003980 Flags: 00000081 Code: 00003968 Name: irq-timer0a
Address: 00003980 Link: 000039AA Flags: 00000081 Code: 00003992 Name: irq-timer0b
Address: 000039AA Link: 000039D4 Flags: 00000081 Code: 000039BC Name: irq-timer1a
Address: 000039D4 Link: 000039FE Flags: 00000081 Code: 000039E6 Name: irq-timer1b
Address: 000039FE Link: 00003A28 Flags: 00000081 Code: 00003A10 Name: irq-timer2a
Address: 00003A28 Link: 00003A52 Flags: 00000081 Code: 00003A3A Name: irq-timer2b
Address: 00003A52 Link: 00004000 Flags: 0000FFFF Code: 00003A72 Name: --- Flash Dictionary ---
```

Der Link der letzten Kerndefinition zeigt stets auf den Anfang des veränderlichen Dictionary–Bereiches im Flash, in den eigene Definitionen einfügt werden können. An dieser Stelle ist entweder eine frisch gelöschte Flash–Zelle oder der Kopf einer weiteren Definition zu finden.

```
Address: 00004000 Link: 00004038 Flags: 00000000 Code: 00004012 Name: numbertable
Address: 00004038 Link: 000040D0 Flags: 00000000 Code: 00004044 Name: e^ka
Address: 000040D0 Link: 000040FC Flags: 00000000 Code: 000040DE Name: 2rshift
Address: 000040FC Link: 00004224 Flags: 00000042 Code: 0000410A Name: cordic
Address: 00004224 Link: 00004244 Flags: 00000042 Code: 00004230 Name: sine
Address: 00004244 Link: 00004268 Flags: 00000042 Code: 00004252 Name: cosine
Address: 00004268 Link: 00004288 Flags: 00000040 Code: 00004272 Name: pi
Address: 00004288 Link: 000042A8 Flags: 00000040 Code: 00004294 Name: pi/2
Address: 000042A8 Link: 00004358 Flags: 00000042 Code: 000042BA Name: widecosine
Address: 00004358 Link: FFFFFFFF Flags: 00000042 Code: 00004368 Name: widesine
ok.
```

In der letzten Definition ist der Link noch nicht gesetzt und signalisiert damit das Ende des Dictionarys.

Im RAM beginnt FIND also mit der Suche bei der aktuellsten Definition, in dem Beispiel bei see, so wie in einem klassischen Forth auch. Der erste Fund mit korrektem Namen wird zurückgegeben — bis hierhin ist noch nichts Ungewöhnliches geschehen. Ist die Definition jedoch nicht im RAM, oder soll ins Flash kompiliert werden, wobei das RAM ausgeblendet wird, durchsucht FIND von der *ältesten* Definition ausgehend die gesamte Linkkette im Flash und stoppt *nicht* beim ersten Fund, sondern durchsucht das gesamte Flash–Dictionary, ob es nicht vielleicht noch etwas Neueres gibt. Auf diese Weise werden auch Redefinitionen korrekt behandelt.

Nach einem Reset kann das aktuelle „Fadenende“ somit durch das Entlanghangeln an der Linkkette wiedergefunden werden. Die Stelle, wo der freie Platz beginnt und auf die HERE zu zeigen hat, kann gefunden werden, indem vom Ende des Flash aus das erste Auftreten einer beschriebenen Speicherstelle gesucht wird. Damit dies funktioniert, muss sichergestellt werden, dass Definitionen nicht mit dem Wert einer gelöschten Flash–Zelle enden — dieser würde sonst beim nächsten Reset fälschlicherweise als „freier Platz“ erkannt und später überschrieben werden. Um dies zu vermeiden, prüft SMUDGE, welches auch für das Schreiben der gesammelten Flags zuständig ist, das Ende der aktuellen Definition und fügt bei Bedarf noch eine Enderkennungs–Füll–Null an.

Zum Einfügen einer neuen Definition benötigt CREATE ein bisschen zusätzliche Logik, um zu erkennen, ob und wohin ein Link geschrieben werden muss, doch das dürfte soweit einleuchten.

Eine Wahl ist nun zu treffen: Womit soll es beginnen?

Nach der grundlegenden Einführung in die Besonderheiten der großen Familie von ARM–Cortex–Mikrocontrollern stellt sich nun die Frage: Womit beginnen? Die Auswahl ist groß und auch die Liste der bereits unterstützten Platinen wächst beständig, unter denen sich auch manche Exoten tummeln. Es ist nicht leicht, eine Empfehlung zu geben, so groß sind die Unterschiede zwischen den einzelnen. Die beliebtesten und grundsätzlich empfehlenswerten sind in den Tabellen 1 und 2 zusammengestellt und werden nun kurz beschrieben.

Gängige Platinen

Wer zum ersten Mal einen ARM Cortex kennenlernen möchte, wem es auf unkomplizierte Handhabung, viele fertige Forth–Beispiele, eine ausgewogene Mischung aus Rechenpower und handhabbarer Komplexität ankommt und für den gute Dokumentation wünschenswert ist, dem sei das *TI–Stellaris–Launchpad* ans Herz gelegt. Auch denen, die sich nicht so leicht entscheiden können, möchte ich es empfehlen. Wer Erweiterungsplatinen für das MSP430–Launchpad hat, kann sie sogleich weiterverwenden. Der einzige Wermutstropfen ist ein fehlender analoger Ausgang. Es ist übrigens die namensgebende erste Plattform, auf der *Mecrisp–Stellaris* lief — und entwickelt worden ist. Mittlerweile wird es fast identisch auch als „Tiva Launchpad“ verkauft.

Wer es gerne sehr übersichtlich mag, ein kleines „Arbeitspony“ sucht und vielleicht auch selbst einmal SMD löten möchte, dem sei das *STM Discovery F0* empfohlen. Es enthält den einfachsten ARM–Chip in dieser Reihe und verfügt ebenfalls über eine sehr gute Dokumentation.

Wer sich „Analoges vom Feinsten“ wünscht, dem sei entweder das *STM Discovery F3* oder *F4* ans Herz gelegt. Auf dem *Discovery F3* ist alles enthalten, um Inertialnavigation auszuprobieren und vielleicht im Modellbau mit einem Autopiloten zu experimentieren. Die Analog–Digital–Wandler sind die schnellsten, sie können mit verringerter Genauigkeit bis zu 9 Msps erreichen. Das *Discovery F4* hat mit 2,4 Msps etwas langsamere AD–Wandler, aber noch mehr Speicher und Rechenpower und ist vor allem für Software Defined Radio beliebt. Kopfhöreranschluss und Mikrofon sind enthalten — eine Einladung für Experimente mit Klang und Musik.

Wer 3,3V–Arduino–Erweiterungsplatinen hat und diese in der „neuen Welt“ weiterbenutzen möchte, hat momentan die Wahl zwischen drei Platinen. Kurz zusammengefasst: Das *Nucleo 401RE* ist am stärksten, hat aber keinen DAC, das *Nucleo L152RE* hat gleich zwei analoge Ausgänge, während das mit nur wenig Speicher ausgestattete *Freescale Freedom KL25Z* eher langsame analoge Peripherie bietet, aber dafür einen AD–Wandler mit 16 Bit Auflösung.

Wer etwas von den Maßen her sehr Kleines benötigt, für den wird das *Teensy 3.1* wie geschaffen sein.

Eine besondere Kategorie bleibt noch übrig

Wer mit Drachen kämpfen möchte, mag es mit dem *Discovery–F429* oder mit dem *Tiva–Connected–Launchpad* aufnehmen. Ersteres hat 8 MB externen Speicher und ein berührungsempfindliches Grafikdisplay an Bord, letzteres einen Netzwerkanschluss und ist mit so manchen Silicon–Bugs gewürzt.

Und was habe ich selbst benutzt?

In diesem Heft soll der Fokus vor allem auf dem ***Stellaris–Launchpad*** liegen, welches im Laufe der Zeit zum „Flagschiff“ von Mecrisp–Stellaris geworden ist und im Mikrocontrollerverleih der Forth–Gesellschaft zur Probe ausgeliehen werden kann. Aber auch die Discovery–Reihe soll beleuchtet werden.

Und es wächst mit...

Im Laufe des Schreibens[1] sind noch einige neue Portierungen dazugekommen, von denen drei besonders hervorzuheben sind:

- Das STM–Discovery–L053, welches mit einer E–Paper–Anzeige ausgerüstet ist!
- Mit dem MSP432 ist TI die Kreuzung eines ARM–Cortex–M4–Prozessors mit der übersichtlichen Peripherie eines MSP430 gelungen.
- Und schließlich der LPC1114FN28 — welcher als einziger im bastelfreundlichen DIP–28 Gehäuse daherkommt, was ihn trotz seiner eher knappen Ausstattung für Experimente auf der Steckplatine interessant macht.

[1] Die Arbeit am Heft begann im Winter 2014/15

Tabelle 1: Cortex M3 und M4 Typen

Platine / Ausstattung	*Flash*	*RAM*	*ADC*	*DAC*	*Standard—-Terminal*
TI Stellaris Launchpad	256 kb	32 kb	2×12 Bits @ 1 Msps	0	USB–Brücke
	RGB–Led, besonders komfortabel, klein und praktisch				
TI MSP432P401R Launchpad	256 kb	64 kb	1×14 Bits @ 1 Msps	0	USB–Brücke
	RGB–Led, Peripherie so ähnlich wie beim MSP430				
TI Tiva Connected Launchpad	1 MB	256 kb	2×12 Bits @ 2 Msps	0	USB–Brücke
	Netzwerk, sehr kompliziert, viele Silicon–Bugs				
STM Discovery F3	256 kb	40 kb	4×12 Bits @ 5,1 Msps	2×12 Bits	TTL
	Viele Leds, Beschleunigungssensor, Gyroskop und Kompass				
STM Discovery F4	1 MB	128 kb + 64kb	3×12 Bits @ 2,4 Msps	2×12 Bits	TTL
	Beschleunigungssensor, Mikrofon und Kopfhörerbuchse				
STM Discovery F429	2 MB	128 kb + 64kb	3×12 Bits @ 2,4 Msps	2×12 Bits	TTL
	Grafikdisplay mit Touch, 8 MB externes RAM, sehr kompliziert, nur wenige Pins frei				
STM Nucleo 401RE	512 kb	96 kb	1×12 Bits @ 2,4 Msps	0	USB–Brücke
	Arduino–Aufsteckmöglichkeit				
STM Nucleo L152RE	512 kb	80 kb	1×12 Bits @ 1 Msps	2×12 Bits	USB–Brücke
	Arduino–Aufsteckmöglichkeit				
Teensy 3.1	256 kb	64 kb	2×16 Bits @ 460 ksps	1×12 Bits	TTL
	Sehr klein und steckplatinenfreundlich				

Tabelle 2: Cortex M0 Typen

Platine / Ausstattung	*Flash*	*RAM*	*ADC*	*DAC*	*Standard—-Terminal*
STM Discovery F0	64 kb	8 kb	1×12 Bits @ 1 Msps	1×12 Bits	TTL
	Einfachster Chip in dieser Reihe, gut überschaubar und steckplatinenfreundlich				
Freescale Freedom KL25Z	128 kb	16 kb	1×16 Bits @ 460 ksps	1×12 Bits	USB–Brücke
	Arduino–Pinleisten einlötbar, RGB–LED und Beschleunigungssensor, chaotische Dokumentation				
STM Discovery L053	64 kb	8 kb	1×12 Bits @ 1,14 Msps	1×12 Bits	USB–Brücke
	E–Paper–Anzeige mit 172×72 Pixeln, 4 Graustufen				
LPC1114FN28	32 kb	4 kb	1×10 Bits @ 400 ksps	0	TTL
	Chip kommt im bastelfreundlichen DIP–28 Gehäuse				

Das Stellaris–Launchpad

Jetzt geht es los![1] *Neugierige werden vielleicht gleich bis hierhin vorgeblättert haben, jetzt sind wir auf jeden Fall wieder beisammen. Bevor es mit dem Lesen im Datenblatt losgeht, seht euch die Übersicht über die Pinbelegung des Stellaris–Launchpads mal an [Abb.1+2].*

Am einfachsten und am wichtigsten sind doch stets die digitalen Ein- und Ausgänge. Da die RGB–Leuchtdiode an Port F angeschlossen ist, soll dieser hier einmal exemplarisch angesprochen werden. Wer von anderen, kleineren Mikrocontrollern kommt, wird spätestens jetzt die Flut von Registern entdecken, von denen hier nur die für das „digitale Leitungswackeln“ benötigten gezeigt werden sollen:

```
\ Reset-Werte im Kommentar.
$400253FC constant PORTF_DATA   \ 0 Ein- und Ausgaberegister
$40025400 constant PORTF_DIR    \ 0 Soll der Pin Eingang oder Ausgang sein ?
$40025500 constant PORTF_DR2R   \ $FF 2 mA Treiber
$40025504 constant PORTF_DR4R   \ 0 4 mA
$40025508 constant PORTF_DR8R   \ 0 8 mA
$4002550C constant PORTF_ODR    \ 0 Open Drain
$40025510 constant PORTF_PUR    \ 0 Pullup Resistor
$40025514 constant PORTF_PDR    \ 0 Pulldown Resistor
$40025518 constant PORTF_SLR    \ 0 Slew Rate
$4002551C constant PORTF_DEN    \ 0 Digital Enable
```

Um die RGB–Leuchtdiode zum Blinken zu bringen, müssen die Leitungen PF1, PF2 und PF3 zunächst als digitale Pins in PORTF_DEN angemeldet werden. Anschließend können sie in PORTF_DIR als Ausgänge gesetzt werden. Da die RGB–LED mit eigenen Treibertransistoren ausgestattet ist, reicht die Standardeinstellung von 2mA Ausgangsstrom aus. Die Taster sind an PF0 und PF4 angeschlossen, müssen also ebenfalls in PORTF_DEN als digitale Pins konfiguriert werden, können aber als Eingang belassen werden. Stattdessen sollen sie Hochziehwiderstände bekommen, die in PORTF_PUR ausgewählt werden können.

Es gibt noch einen Sonderfall zu beachten: PF0 ist standardmäßig nach einem Reset der NMI–Eingang. Dies soll hier angesprochen werden, weil oft einige Pins nach dem Reset mit einer Sonderfunktion belegt ist, die im Zweifelsfalle mit einer Sequenz, die im Datenblatt irgendwie zu finden sein sollte, deaktiviert werden muss, falls ein Pin sich partout nicht so verhält, wie eigentlich gewünscht.

Damit ist die Initialisierung[2] für unsere ersten Experimente mit dem Stellaris Launchpad komplett:

```
: init
  \ PF0 ist auch der NMI-Eingang.
  \ Benötige also eine besondere Sequenz,
  \ um ihn für den Taster freizuschalten.
    $4C4F434B $40025520 ! ( PORTF_LOCK )
    1 $40025524 bis!      ( PORTF_CR )
    0 $40025520 !         ( PORTF_LOCK )
  \ Alle Leitungen an Port F seien digitale Pins
    %11111 portf_den !
  \ Die Leuchtdiodenanschlüsse seien Ausgänge
    %01110 portf_dir !
  \ Hochziehwiderstände für die Taster aktivieren
    %10001 portf_pur ! ;
```

Jetzt ist alles soweit, ein 2 PORTF_DATA ! lässt rotes Licht leuchten. Blau leuchtet mit 4 und Grün mit 8. Ebenso können die Taster aus PORTF_DATA eingelesen werden.

[1] Mecrisp–Stellaris ist unter `mecrisp.sourceforge.net` zu finden, außerdem wird zum Flashen noch lm4flash aus `https://github.com/utzig/lm4tools` benötigt.

[2] Die neueste Definition im Flash mit dem Namen „init“ wird beim Start von Mecrisp–Stellaris übrigens automatisch ausgeführt!

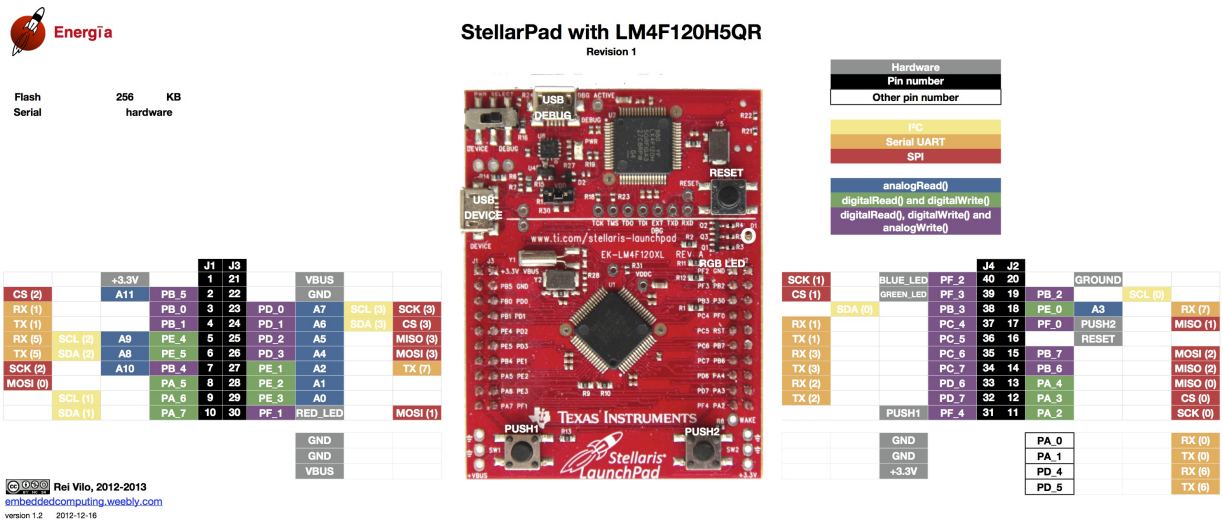


Abbildung 1: Pinout des Stellaris–Launchpads

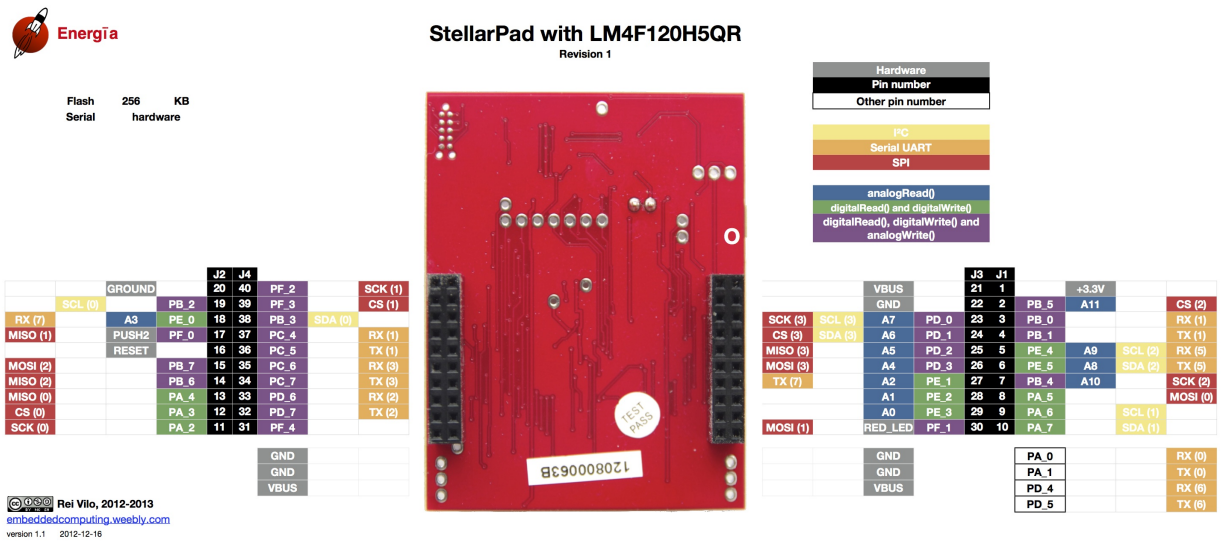


Abbildung 2: Pinout Stellaris–Launchpads — Ansicht von der Unterseite

STM Discovery F3 und F4

An dieser Stelle soll ein kleiner Abstecher gewagt werden, um zu zeigen, wie unterschiedlich die Peripherie trotz gleichen Prozessorkernes bei den verschiedenen Herstellern sein kann.

Während das Stellaris–Launchpad nur 8 Leitungen an jedem Port hat, sind die Ports hier gleich 16 Bits breit. Die Leuchtdioden sind beim Discovery F3 an Port E (Basisadresse $48001000) an den Pins PE8–PE15: Blau, Rot, Orange, Grün, Blau, Rot, Orange, Grün. Beim Discovery F4 sind sie an Port D (hier mit Basisadresse $40020C00) an den Pins PD12–PD15: Grün, Orange, Rot, Blau. Bei beiden haben die Ports den gleichen Registeraufbau, von dem hier ein Auszug gezeigt werden soll:

```
PORTE_BASE $00 + constant PORTE_MODER
 \ 0 Port Mode Register
 \ 00=Input 01=Output 10=Alternate 11=Analog
PORTE_BASE $04 + constant PORTE_OTYPER
 \ 0 Port Output type register
 \ (0) Push/Pull vs. (1) Open Drain
PORTE_BASE $08 + constant PORTE_OSPEEDR
 \ 0 Output Speed Register
 \ 00=2 MHz 01=25 MHz 10=50 MHz 11=100 MHz
PORTE_BASE $0C + constant PORTE_PUPDR
 \ 0 Pullup / Pulldown
 \ 00=none 01=Pullup 10=Pulldown
PORTE_BASE $10 + constant PORTE_IDR
 \ RO Input Data Register
PORTE_BASE $14 + constant PORTE_ODR
 \ 0 Output Data Register
PORTE_BASE $18 + constant PORTE_BSRR
 \ WO Bit set/reset register
 \ 31:16 Reset, 15:0 Set
```

Beim STM sind die Pins standardmäßig als digitale Eingänge konfiguriert, so dass der blaue Taster, der bei beiden an `PA0` angeschlossen ist, direkt aus PORTA_IDR (Im F3: $48000000 Im F4: $40020000) gelesen werden kann.

Um die Leuchtdioden als Ausgang zu setzen, gibt es für die 16 Leitungen jeweils Bitpaare in dem 32–Bit–Register PORTx_MODER, die auf %01 gesetzt werden müssen. Ein Aktivieren aller Leuchtdioden–Ausgänge ist somit im F3 mit `$55550000 PORTE_MODER !` und im F4 mit `$55000000 PORTD_MODER !` getan.

Nach diesen Vorbereitungen können schließlich die gewünschten Farben aktiviert werden. Zum Beispiel Blau im F3 mit `1 8 lshift PORTE_ODR !` und Grün im F4 mit `1 12 lshift PORTD_ODR !` Wer nur einzelne Farben ein– oder ausschalten möchte, ohne die anderen zu beeinflussen, kann dies im BSRR–Register tun. Eine Eins, die in die unteren 16 Bits des BSRR–Registers geschrieben wird, setzt das entsprechende Bit im ODR–Register, in den oberen 16 Bits führt eine geschriebene Eins zum Löschen der Leitung.

Terminal anschließen

Zum Ausprobieren noch ein kleiner, aber entscheidender Hinweis: Während beim Stellaris–Launchpad eine USB–Seriell–Brücke für das Terminal bereits eingebaut ist, müssen hier die Terminalleitungen selbst angeschlossen werden. Dafür wird ein USB–Seriell–Adapter mit 3,3V–Pegeln benötigt. Beim Discovery F3 ist TX an PA9 und RX an PA10, während beim Discovery F4 TX an PA2 und RX an PA3 gefunden werden kann.

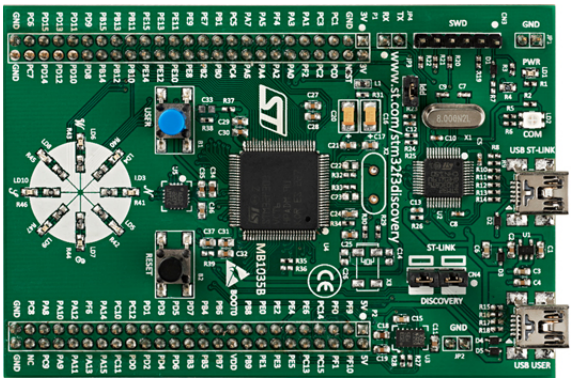


Abbildung 1: STM Discovery F3

Discovery F4:

	GND	1	GND			GND	1	GND	
	VDD	2	VDD			5V	2	5V	
	GND	3	NRST	Resettaster		3V	3	3V	
	PC1	4	PC0	USB		PH0	4	PH1	
Mic DATA	PC3	5	PC2			PC14	5	PC15	
	PA1	6	PA0	Taster		PE6	6	PC13	
Terminal RX	PA3	7	PA2	Terminal TX		PE4	7	PE5	
Acc SCK, DAC 2	PA5	8	PA4	I2S WS, DAC 1		PE2	8	PE3	Accel CS
Accel MOSI	PA7	9	PA6	Accel MISO	Accel INT1	PE0	9	PE1	Accel INT2
	PC5	10	PC4			PB8	10	PB9	Audio SDA
	PB1	11	PB0			BOOT0	11	VDD	
	GND	12	PB2		Audio SCL	PB6	12	PB7	
	PE7	13	PE8			PB4	13	PB5	
	PE9	14	PE10			PD7	14	PB3	
	PE11	15	PE12			PD5	15	PD6	
	PE13	16	PE14			PD3	16	PD4	
	PE15	17	PB10	Mic CLK		PD1	17	PD2	
	PB11	18	PB12		I2S SD	PC12	18	PD0	
	PB13	19	PB14		I2S SCK	PC10	19	PC11	
	PB15	20	PD8			PA14	20	PA15	
	PD9	21	PD10		USB	PA10	21	PA13	
	PD11	22	PD12	Led Grün		PA8	22	PA9	USB
Led Orange	PD13	23	PD14	Led Rot		PC8	23	PC9	
Led Blau	PD15	24	NC			PC6	24	PC7	I2S MCK
	GND	25	GND			GND	25	GND	

Discovery F3:

	3V	1	3V			5V	1	5V	
	GND	2	NRST	Resettaster		PF9	2	PF10	
	PC1	3	PC0			PF0	3	PF1	
	PC3	4	PC2			PC14	4	PC15	
	PA1	5	PF2			PE6	5	PC13	
	PA3	6	PA0	Taster	Mag INT1	PE4	6	PE5	Mag INT2
	PF4	7	PA2		Mag DRDY	PE2	7	PE3	Gyro CS
Gyro SCK, DAC 2	PA5	8	PA4	DAC 1	Gyro INT1	PE0	8	PE1	Gyro INT2/DRDY
Gyro MOSI	PA7	9	PA6	Gyro MISO		PB8	9	PB9	
	PC5	10	PC4			BOOT0	10	VDD	
	PB1	11	PB0		Mag SCL	PB6	11	PB7	Mag SDA
	PE7	12	PB2			PB4	12	PB5	
Led Rot N	PE9	13	PE8	Led Blau NW		PD7	13	PB3	
Led Grün O	PE11	14	PE10	Led Orange NO		PD5	14	PD6	
Led Rot S	PE13	15	PE12	Led Blau SO		PD3	15	PD4	
Led Grün W	PE15	16	PE14	Led Orange SW		PD1	16	PD2	
	PB11	17	PB10			PC12	17	PD0	
	PB13	18	PB12			PC10	18	PC11	
	PB15	19	PB14			PA14	19	PA15	
	PD9	20	PD8			PF6	20	PA13	
	PD11	21	PD10		USB	PA12	21	PA11	USB
	PD13	22	PD12		Terminal RX	PA10	22	PA9	Terminal TX
	PD15	23	PD14			PA8	23	PC9	
	PC6	24	PC7			PC8	24	NC	
	GND	25	GND			GND	25	GND	

Abbildung 2: Pinbelegungen der beiden STM Discovery Platinen

Flags, Konstantenfaltung und Optimierungen

Nachdem die vorherigen Seiten vor allem traditionelle Konzepte ein bisschen angepasst und erweitert haben, damit Forth besonders gut auf Mikrocontrollern laufen kann, sollen jetzt die Optimierungen und Tricks gezeigt werden, die Mecrisp–Stellaris während des Kompilierens in Maschinensprache verwendet.

Bestimmt ist die bunte Vielfalt an Flags bereits aufgefallen — sie lässt sich aus der doch recht ungewöhnlichen Funktionsweise von Interpret erklären.

Liste der Flags, Beschreibung, Beispiele

$FFFF: Unsichtbar — Frisch gelöschte Flash–Zelle. Noch nicht fertige oder wegen eines Fehlers unfertig gebliebene Definitionen.

$0000: Sichtbar — Ganz normal, nichts Besonderes. Beispielsweise `emit type : .s`

$0010: Immediate — Das klassische Immediate. Wird beim Kompilieren ausgeführt. `postpone s"`

$0020: Inline — Der winzige Code der Definition wird beim Kompilieren anstelle eines Subroutinenaufrufes direkt eingefügt. `@ pick >r i j k`

$0030: Immediate+Inline — Diese eigentlich sinnlose Kombination wird für „Immediate, Compile only“ verwendet. `exit recurse ;` Kontrollstrukturen wie `if`

Außer den „klassischen“ Flags gibt es eine ganze Reihe von Flags für die Konstantenfaltung:

$0040: 0–Faltbar — Kann bereits beim Kompilieren ausgeführt werden, wenn mindestens 0 Konstanten bereitliegen. `base` Variablen und Konstanten

$0041: 1–Faltbar — –”– , wenn mindestens eine Konstante bereitliegt. `aligned`

$0042: 2–Faltbar — –”– , wenn mindestens zwei Konstanten bereitliegen. `2dup dabs dnegate`

$0043: 3–Faltbar — –”– , wenn mindestens drei Konstanten bereitliegen. `*/ */mod`

$0044: 4–Faltbar — –”– , wenn mindestens vier Konstanten bereitliegen. `2swap 2over d/ d* d> d<`

...

$0047: 7–Faltbar — –”– , wenn mindestens sieben Konstanten bereitliegen. Mehr als 4–faltbar kommt im Kern allerdings (noch?) nicht vor.

Oft kommen die Faltbar–Flags in Kombination mit Inline vor:

$0060: 0–Faltbar+Inline `true false`

$0061: 1–Faltbar+Inline `dup ?dup drop not 0= 1+`

$0062: 2–Faltbar+Inline `2drop nip d0<`

$0063: 3–Faltbar+Inline `rot -rot`

...

$0067: 7–Faltbar+Inline Unbenutzt

Für einige besondere Fälle können je nach vorhandenen Konstanten direkt Opcodes generiert werden. Tritt meist in Kombination mit Inline auf, falls nicht genug Konstanten für die Optimierung vorhanden sind:

$0048 oder $0068 (+Inline): Spezialfälle mit eigenem Codegenerator. `bit@ hbit@ cbit@`

$0049 oder $0069 (+Inline): Plus und Minus: `+ -`

$004A oder $006A (+Inline): Rechnungen (und Logik): `*` (und im M0 auch `and bic or xor`)

$004B oder $006B (+Inline): `= <>`

$004C oder $006C (+Inline): Schieben: `lshift arshift rshift`

$004D oder $006D (+Inline): Speicherschreiben: `! +! h! h+! c! c+!`

$004E oder $006E (+Inline): Nur in M3/M4: Logik: `and bic or xor`

Für die Reservierung von Puffer–RAM und das Anlegen von Variablen beim Kompilieren in Flash–Speicher:

$0080... Initialisiertes RAM anfordern. Automatisch stets auch 0–Faltbar (Variablenadresse ist konstant!).

$0081: 4 Bytes initialisiertes RAM. Mit `variable` im Flash generierte Definitionen

$0082: 8 Bytes initialisiertes RAM. Mit `2variable` im Flash generierte Definitionen

...

$008F: 15*4 = 60 Bytes initialisiertes RAM. Mit `nvariable` im Flash generierte Definitionen.

Die Initialisierungswerte liegen dabei am Ende der Definition nach dem Rücksprung–Opcode und werden beim Start von Mecrisp–Stellaris während des *Dictionarysuchlaufs* an eine feste Stelle ins RAM kopiert. Dies funktioniert so, dass zu Beginn ein *Variablenzeiger* auf das Ende des RAMs zeigt. Der Suchlauf zieht nun für jede definierte Variable die gewünschte Größe vom Variablenzeiger ab und kopiert den Bereich nach dem Rücksprung–Opcode mit den Initialisierungswerten, die im Flash abgelegt sind, dahin. Neue Variablen verkleinern also diesen Zeiger und kompilieren die errechnete Adresse fest ein. Da sich durch die Reihenfolge der Suche vom ältesten hin zum aktuellen Dictionaryeintrag stets die gleichen Adressen ergeben, muss dieser Variablenzeiger ebenfalls nicht gesichert werden. Bei entsprechendem Flash–Controller ist es sogar möglich, einen Teil der aktuellen Definitionen zusammen mit deren reservierten Speicherstellen zu löschen.

$0100: Reserviert einen großen, uninitialisierten RAM–Block, dessen Größe nach dem Rücksprung–Opcode abgelegt ist. Dabei wird intern dafür gesorgt, dass die Startadresse stets auf vier gerade ist.

Konstantenfaltung

Konstantenfaltung bedeutet, dass einige Operationen bereits während des Kompilierens erledigt werden können, falls die dafür nötigen Elemente Konstanten sind. Zum Beispiel bewirkt `2 3 * +` dasselbe wie `6 +` und ebenso könnte die Sequenz `1 8 lshift 1 or` auch durch die Konstante `257` ersetzt werden.

Um dies zu erreichen, werden während des Kompilierens alle auftretenden Konstanten auf dem Stack gesammelt und Definitionen, die mit einer bestimmten Mindestzahl von Konstanten als *faltbar* markiert sind und keine Nebenwirkungen haben, gleich ausgeführt. Dabei ist es nur wichtig zu wissen, wie viele Eingabeelemente benötigt werden, da die Zahl der Ergebnis–Elemente anschließend über den Stackfüllstand bestimmt werden kann. Somit ist auch `?dup` 1–faltbar! Wird anschließend eine normale Definition kompiliert, werden zuvor alle bis dahin noch auf dem Stack verbliebenden Konstanten traditionell einkompiliert.

Um die Funktionsweise zu veranschaulichen sei hier ein ausführliches Beispiel gegeben — wobei die Stackkommentare *während* des Kompilierens gelten ! Es empfiehlt sich, dies zum Verständnis auch einmal in der Struktur von Interpret zu verfolgen [Abbildung1].

```
42 constant antwort
\ Konstanten werden automatisch
\ als 0-faltbar markiert.
: 6+   2 3 swap * + antwort drop ;
```

`:` 6+ Dictionaryeintrag erstellen (—) Wechsel in den Compile–Modus

`2` wandert mit frisch gesetztem Füllstandszeiger auf den Stack (2)

`3` wandert auf den Stack (2 3)

`swap` ist 2–faltbar, es sind gerade zwei Konstanten vorhanden, kann also gleich erledigt werden. swap wird ausgeführt, lässt Ergebnisse auf dem Stack zurück. (3 2)

`*` ist 2–faltbar. Es sind gerade zwei Konstanten vorhanden, kann also erledigt werden. Mal wird ausgeführt, und lässt das Ergebnis auf dem Stack liegen. Füllstandszeiger war die ganze Zeit über gesetzt... (6). Es ist also nur wichtig zu wissen, wie viele Konstanten ein Wort als Eingabe braucht, die Zahl der Ausgabeelemente ist beliebig, da der Füllstandszeiger die Zahl der vorhandenen oder frisch generierten Konstanten bestimmen lässt.

`+` ist 2–faltbar. Es ist aber nur eine Konstante da, geht also nicht. Schreibe die noch herumliegende 6 ins Dictionary (—). Lösche Füllstandszeiger (Damit beispielsweise die Sprungziele von IF nicht später versehentlich als Literal behandelt werden...). Kompiliere + und der Füllstandszeiger wird frisch gesetzt. (—)

`antwort` ist 0–faltbar. Kann also immer ausgeführt werden, auch ohne dass etwas bereitliegt. Wandert mit dem frisch gesetztem Füllstandszeiger auf den Stack (42)

`drop` ist 1–faltbar, eine Konstante ist da, erledige es sogleich (—)

`;` Keine Konstanten mehr da, nichts mehr zu tun, schließe Definition ab.

Im Endeffekt wird also nur `: 6+ 6 + ;` tatsächlich ins Dictionary geschrieben. Die Abbildung 1 zeigt wie die Konstantenfaltung in der INTERPRET–Schleife abläuft.

Opcodierungen

Während Konstantenfaltung von der Idee her nicht näher an den verwendeten Prozessorkern gebunden ist, sind die möglichen Opcodierungen, also Optimierungen durch das Erzeugen von konstantenspezifischen Opcodes, sehr eng an den Befehlssatz und dessen Möglichkeiten gebunden.

Im vorherigen Beispiel würde beim + mit einer Konstanten ein Opcodierfall in Kraft treten und somit nur ein einziger Opcode geschrieben werden: `adds tos, #6`.

Wer besonders neugierig ist, dem sei ans Herz gelegt, den Disassembler[1] zu laden und einige frisch kompilierte Definitionen selbst anzusehen.

Um den Einblick in den Dschungel ein bisschen zu erleichtern, hier einige Tipps zum Zurechtfinden:

- Registerbelegung:
  - r0–r3 sind temporäre Arbeitsregister, die nur bei einem Interrupt–Einsprung gesichert werden.
  - r4 und r5 enthalten Index und Limit der innersten do–Schleife.
  - r6 ist TOS und r7 enthält den Stackpointer.
  - r8–r12 sind in Mecrisp–Stellaris unbenutzt und können für eigene Erweiterungen verwendet werden.
  - r13=sp ist der Return–Stackpointer.

[1] In dem Mecrisp–Stellaris–Paket gibt es einen Ordner namens *common*, in dem all die nützlichen Helferlein und Werkzeuge zu finden sind, welche querbeet auf allen Portierungen verwendet werden können. Alle Disassembler–Beispiele sind entweder mit dem `disassembler-m0` oder dem `disassembler-m3` entstanden, doch es gibt noch viel mehr zu entdecken: Sinus und Cosinus sind dabei, ein Blockeditor mit Unicode–Unterstützung sowie ein Hexdump–Werkzeug liegen bereit und Grafikroutinen warten auf ihren künstlerischen Einsatz.

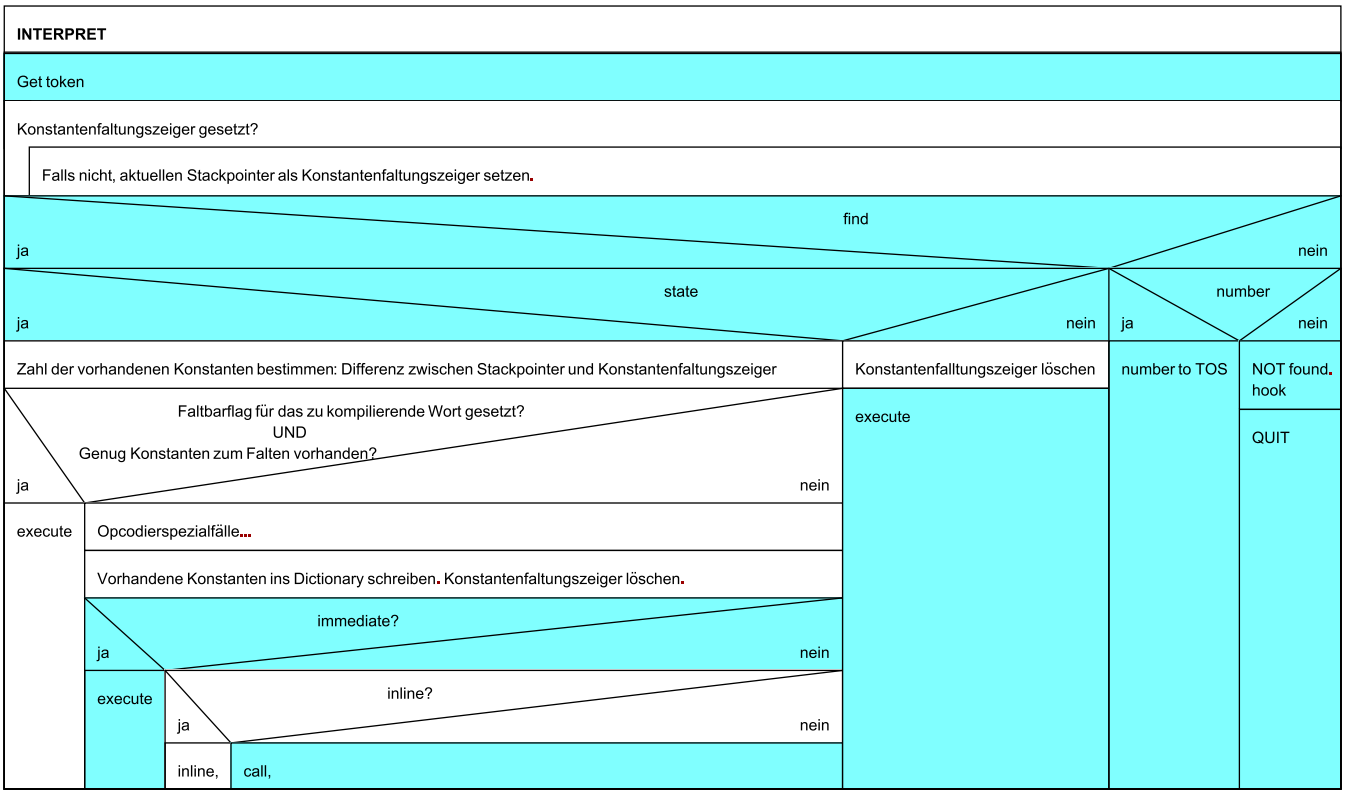


Abbildung 1: Interpret und die Konstantenfaltung im Nassi–Shneiderman–Diagramm.

  - r14=lr ist ein festverdrahtetes Rücksprung–Link–Register, worin die letzte Subroutinenebene ohne Stackbewegung angesprungen werden kann. Viele Kerndefinitionen, die keine weiteren Subroutinen aufrufen, enden mit bx lr.

  - r15=pc ist der Programmzähler, der stets ungerade ist, um die Verwendung des Thumb–Befehlssatzes zu kennzeichnen.

- Selbst kompilierte Definitionen beginnen stets mit `push {lr}` und enden mit `pop {pc}`, wobei auch `exit` diesen Opcode erzeugt.

- Konstanten werden, falls sie nicht in einen einzelnen `movs` Opcode passen, entweder im M3/M4 als `movw movt` Folge kompiliert oder im M0 aus `movs, lsls` und `adds` Opcodes zusammengesetzt.

- Der relative `bl` Opcode wird für Subroutinen verwendet, die nahe an der aktuellen Ausführungsstelle sind. Der Einsprung vom RAM ins Flash ist zu weit und wird deshalb stets in der Form `Adresse-->r0 blx r0` kompiliert.

- Alle Kontrollstrukturen werden nativ einkompiliert und sind deshalb im Disassembler–Listing nicht gleich offensichtlich.

```
: random ( -- u )
\ Generiert Zufallszahlen mit dem Rauschen
\ vom Temperatursensor am ADC
  0
  32 0 do  shl temperature 1 and xor
       loop ;
see random
200009C2: B500  push { lr }
200009C4: B430  push { r4  r5 }
200009C6: 2400  movs r4 #0
200009C8: 2520  movs r5 #20
200009CA: F847  str r6 [ r7 #-4 ]!
200009CC: 6D04
200009CE: 2600  movs r6 #0
200009D0: 0076  lsls r6 r6 #1
200009D2: F7FF  bl
20000936  --> temperature
200009D4: FFB0
200009D6: F016  ands r6 r6 #1
200009D8: 0601
200009DA: CF01  ldmia r7 { r0 }
200009DC: 4046  eors r6 r0
200009DE: 3401  adds r4 #1
200009E0: 42AC  cmp r4 r5
200009E2: D1F5  bne 200009D0
200009E4: BC30  pop { r4  r5 }
200009E6: BD00  pop { pc }  ok.
```

Multitasking

Multitasking bedeutet, mehrere Aufgaben zugleich zu erledigen. Für einen Chip mit genau einem Prozessor, der genau einen Befehl auf einmal bearbeiten kann, ist dies streng genommen unmöglich. Doch in vielen Fällen kommt es nicht so sehr auf die exakte Gleichzeitigkeit an, so dass Multitasking dadurch implementiert werden kann, schnell und in kurzen Zeitabständen zwischen den einzelnen Aufgaben zu wechseln. Für diesen Wechsel ist es nötig, den momentanen Zustand des Prozessors zu sichern, den folgenden Task auszuwählen und dessen gesicherten Zustand zurückzuholen.

Im Großen und Ganzen gibt es zwei Varianten, die sich in der Art unterscheiden, wie der „Aufgabenwechsel“ angestoßen wird. Beide Varianten haben ihre Stärken und Schwächen, auf die in einem Vergleich kurz eingegangen werden soll:

Beim präemptiven Multitasking, wie es beispielsweise in Linux verwendet wird, gibt es einen Timer–Interrupt, der periodische Taskwechsel auslöst, wobei der Prozessorzustand vollständig gesichert wird. Dies hat den großen Vorteil, dass der Taktwechsel garantiert kommt — wenn also ein Task abstürzt oder langwierige Berechnungen durchführt und dabei „vergisst“, eine Pause einzulegen, läuft das System trotzdem ohne vollständig einzufrieren weiter. Der Nachteil ist, dass der Taskwechsel so an jeder beliebigen Stelle auftreten kann, es ist also nicht mehr garantiert, dass eine Reihe von kritischen Instruktionen ohne Unterbrechung ausgeführt wird.

Beim kooperativen Multitasking, wie es meist in Forth verwendet wird, ist jeder Task selbst dafür verantwortlich, ab und an eine Pause einzulegen und den nächsten Task an die Reihe zu lassen. Dies erfordert Ordnung und Disziplin, dafür tritt ein Taskwechsel nur an den dafür vorgesehen Stellen auf. So gibt es keine Schwierigkeiten mit der unterbrechungsfreien Ausführung einer Reihe von Befehlen, außerdem ist für den Taskwechsel weniger Aufwand nötig, da beispielsweise temporäre Register oder die Flags nicht gesichert zu werden brauchen. Außerdem bleiben die Interrupts frei, so dass ein Interrupthandler mit Echtzeitanforderungen, der vielleicht derweil ein Signal mit dem Digital–Analog–Wandler generiert, trotz Multitasking ungestört laufen kann.

Hier soll das kooperative Multitasking vorgestellt werden, doch der Schritt zum präemptiven Multitasking ist nicht weit.

Zum Benutzen

Die langjährige Erfahrung vieler Forth–Programmierer zeigt, dass es meist genügt, wenn in der Ein- und Ausgabe Pausen eingelegt werden — im Terminal, in Warteschleifen, in Block. Nur wer lange Rechnungen durchführen lässt oder große Schleifen programmiert, sollte auch hier ab und an eine Pause einfügen.

Viel mehr ist nicht zu beachten, und es kann konkret an die Implementation eigener Tasks gehen !

Nach dem Laden des Multitaskers, in dem vorher noch die gewünschte Stackgröße angepasst werden sollte, kann dieser mit `multitask` und `singletask` ein- und ausgeschaltet werden. `Tasks` zeigt die Liste der aktuell laufenden und ruhenden Tasks an.

```
0 variable ticks  \ Dies Beispiel benötigt eine Variable
task: tickzähler   \ Platz für die neuen Stacks reservieren

: tickzähler& ( -- )
  0 ticks !           \ Vorbereitungen für den neuen Task
  tickzähler activate  \ Task präparieren, als aktiv markieren und einhängen.
                        \ Activate beendet tickzähler& hier,
                         \ direkt danach beginnt der Code für den neuen Task!
    begin        \ Hier beginnt der neue Task
      1 ticks +!  \ Etwas unternehmen...
      pause        \ Und eine Pause einlegen
    again           \ Tasks als Endlosschleife formulieren !
;

multitask  \ Multitasking einschalten
tickzähler& \ Task starten
```

Anschließend läuft der `Tickzähler` im Hintergrund und zählt mit, wie oft ein Taskwechsel bereits ausgeführt wurde. Um nur diesen Task für eine Weile anzuhalten, kann `tickzähler idle` verwendet werden, `tickzähler wake` lässt ihn weiterlaufen. Mit `background` statt `activate` lässt sich ein Task im „Ruhezustand“ in die Taskliste einfügen und wartet dort auf seine Aktivierung mittels `wake`. Das ist nützlich, wenn beispielsweise ein Task aus einem Interrupt heraus aktiviert werden soll — schließlich sind `wake` und `idle` interruptsicher, während das Definieren eines neuen Tasks nicht in einem Interrupt geschehen

sollte. Um den aktuell laufenden Task anzuhalten, kann auch `stop` verwendet werden. Wer jetzt neugierig `stop` eintippt, verabschiedet sich von Forth — dadurch wird das gewohnte Terminal selbst beendet.

Ja, so einfach ist Multitasking anzuwenden !

Innendrin

Zuallererst wird eine verlinkte Ringliste der Tasks benötigt, für dessen Pflege es mehrere Helferlein gibt, außerdem müssen die Tasks zu Beginn passend initialisiert werden und benötigen eigene Stacks. Dafür bekommt jeder Task eine Datenstruktur, die folgendes enthält:

- Zeiger auf den nächsten Task
- Flag: Ist der Task aktiv ?
- Gesicherter Stackpointer
- Ein Haken für Catch & Throw (nur für Fehlerbehandlung)
- Platz für den Datenstack
- Platz für den Returnstack

Zu Beginn, wenn es nur einen Task gibt, zeigt der erste Zeiger auf sich selbst, bei mehreren Tasks bilden die Zeiger einen Ring, so dass mit diesem Zeiger stets der nächste Task aus der Liste gefunden werden kann. Das folgende Flag gibt an, ob der Task gerade aktiv ist — natürlich wäre es denkbar, den Task einfach aus der Ringliste zu entfernen, doch das Setzen eines Flags ist im Gegensatz zum Einfügen und Entfernen im Ring auch aus einem Interrupt heraus sicher zu bewerkstelligen. Der Haken für Catch & Throw dient dem Abfangen von Fehlern und ist für die Funktion des Multitaskers selbst unwichtig. Anschließend folgen die Stacks des Tasks, die zu Beginn natürlich vorbereitet werden müssen. Doch zuerst soll noch eine sehr wichtige Definition vorgestellt werden, an der sich alle Vorbereitungen orientieren müssen.

Die Hauptarbeit wird von `(pause)` übernommen, die Routine, die für den Taskwechsel zuständig ist:

```
boot-task variable up \ User Pointer
: next-task  ( -- task )    up @ inline ;
: task-state ( -- state )   up @ 1 cells + inline ;
: save-task  ( -- save )    up @ 2 cells + inline ;
: handler    ( -- handler ) up @ 3 cells + inline ;

: (pause) ( stacks fly around )
    [ $B430 h, ]        \ push { r4  r5 } to save I and I'
    rp@ sp@ save-task !  \ save return stack and stack pointer

    begin
      next-task @ up !     \ switch to next running task
    task-state @ until

    save-task @ sp! rp!  \ restore pointers
    unloop ;              \ pop { r4  r5 } to restore the loop registers
```

`(pause)` besteht aus drei Teilen:

1. Zuerst werden alle wichtigen Register und der aktuelle Returnstackpointer auf den Stack geholt, anschließend wird der Stackpointer in der Task–Struktur gesichert.
2. Aus der Ringliste wird der nächste aktive Task ausgesucht.
3. Mit den in der Task–Struktur gesicherten Werten werden die Stacks und alle wichtigen Register wiederhergestellt. Der *Rücksprung* erfolgt jetzt über den gerade frisch eingehängten Returnstack !

Falls es nur einen Task gibt, kehrt Schritt 3 einfach nur den Schritt 1 um, doch bei mehreren Tasks befinden wir uns anschließend mit neuen Stacks mitten in einer anderen Definition.

Anhand von `(pause)` ist nun ersichtlich, was von `activate` oder `background` vorbereitet werden muss: Der Returnstackpointer muss auf dem Datenstack liegen, im Returnstack müssen Schleifenindex, Schleifenlimit und die Einsprungadresse bereitliegen, bevor der neue Task in die Ringliste eingefügt werden kann.

Insgesamt sind die im folgenden *Multitasking–Glossar* gezeigten Definitionen für den Benutzer wichtig.

Multitasking Glossar

Interruptsicher:

wake (task −−)	Lässt einen Task loslaufen
idle (task −−)	Legt einen Task schlafen

Nicht aus einem Interrupt heraus zu benutzen:

stop (−−)	Legt den aktuellen Task schlafen
insert (task −−)	Fügt einen Task in die Ringliste ein
remove (task −−)	Entfernt einen Task aus der Ringliste
task: Name (−−)	Reserviert Speicher für einen neuen Task
activate (task −− R: continue −−)	Bereitet einen neuen laufenden Task vor
background (task −− R: continue −−)	Bereitet einen neuen schlafenden Task vor
pause (−−)	Legt eine Pause ein, um einen Taskwechsel zu ermöglichen
tasks (−−)	Zeigt die Ringliste aller Tasks an
multitask (−−)	Multitasking aktivieren
singletask (−−)	Multitasking ausschalten

Interrupts

Auch wenn Multitasking ganz ohne Interrupts auskommt, lassen sich Interrupts aus Forth heraus bequem benutzen. Hier ein kleines Beispiel, welches auf allen Portierungen einsatzbereit ist:

```
false variable Pendel
: Pendeluhr ( -- )
  Pendel @ dup if ." Tick"
               else ." Tack" then
  0= Pendel ! cr ;

' Pendeluhr irq-systick !

\ Ergibt 1 Hz bei 16 MHz Takt:
  16000000 $E000E014 !

\ Systick-Interrupt aktivieren
        7 $E000E010 !

\ Interrupts aktivieren,
\ falls noch nicht geschehen.
eint
```

Innendrin funktioniert es so, dass für die wichtigsten Interrupts Variablen existieren, die Zeiger auf die jeweiligen Interruptroutinen beinhalten. Beim Start werden diese Variablen mit der Adresse von `UNHANDLED` initialisiert, welches die „Unhandled Interrupt...“ Meldung ausgibt.

Manche Chips haben so viele Interruptvektoren, dass es vom Platz her kaum möglich wäre, allen eine eigene Variable zuzuordnen — diese werden dann zum „Sammelinterrupt“ `irq-collection` zusammengefasst. Dennoch können sämtliche vorhandenen Interrupts verwendet und unterschieden werden: Die kleine Definition `IPSR` (Interrupt Program Status Register) gibt die Nummer des gerade laufenden Interrupts zurück, so dass die Interruptroutine für `irq-collection` einfach mit `IPSR CASE...` begonnen werden kann. Genauso existiert mit `irq-fault` auch ein Sammelinterrupt für Fehler aller Art.

Es muss nur darauf geachtet werden, dass die Interruptroutinen in Forth den Stackeffekt `( -- )` haben — um alles andere kümmert sich Mecrisp–Stellaris von selbst. Mit `EINT` und `DINT` können die Interrupts ein- und ausgeschaltet werden, der aktuelle Zustand lässt sich mit `EINT?` herausfinden.

Während `irq-systick` keine weiteren Vorbereitungen benötigt, müssen die meisten anderen Interrups außer in ihrem jeweiligen Peripheriemodul zusätzlich noch „persönlich“ im NVIC, der „zentralen Interruptverwaltung“, aktiviert werden, worin auch die Prioritäten der unterschiedlichen Interrupts eingestellt werden können. Doch an dieser Stelle beginnen sich die einzelnen Chips von unterschiedlichen Herstellern bereits zu unterscheiden, so dass ein Blick in die beiligenden Beispiele und auch in die Datenblätter empfehlenswert ist.

Mehr Register!

Nun soll es noch einmal richtig in die Tiefe der Implementation gehen — wobei wir uns jedoch auch weit vom traditionellen Forth entfernen werden, noch weiter als bisher in diesem Sonderheft. Der Kerngedanke ist, die Operationen, die Forth auf dem Stack ausführt, so gut wie möglich für einen eher auf Registern basierenden Prozessor zu übersetzen. Beim ARM ist es ganz klar: Weniger Stackzugriffe sparen Platz und Ausführungszeit — nur wie soll Forth für den Programmierer transparent auf Register–Befehle übersetzt werden? Die dahintersteckende Idee ist eigentlich nicht schwierig und soll hier erläutert werden.

Während der Compiler eine Definition abarbeitet, verfolgt er im Hintergrund mit einem *Modell des Stacks* so weit wie möglich, welche Stackelemente gerade bearbeitet werden und welche Operationen dazwischen wirken sollen, damit Zwischenergebnisse in Registern verbleiben können und beispielsweise Rechen- oder Logikoperationen direkt zwischen diesen Registern wirken können. Immer zu Beginn und am Ende einer Definition, oder wenn eine andere Definition aufgerufen werden soll, müssen die frei in Registern liegenden Elemente wieder auf den Stack gelegt werden, um eine einheitliche, kanonische Schnittstelle zu ermöglichen, der Grundzustand des Stacks, wie er auch in klassischen Forth–Implementationen verwendet wird.

Wichtig ist dafür zu wissen, wo sich die Stackelemente eigentlich gerade *tatsächlich* befinden — dafür ist das *Stackmodell* zuständig. Normalerweise sieht „der Stack“ ja so aus, dass die Werte im RAM aufgestapelt liegen, und nur der TOS immer schon in einem Register gehalten wurde. Im folgenden Schema ist TOS daher im Register r6 aufgehoben und NOS bis 5OS und alle tieferen Werte sind nach wie vor im RAM gestapelt und (noch) nicht in ein Register verlagert worden. Der „echte Stack“, also der *physikalisch* tatächlich vorhandene Platz der Werte, ist also immer eine Mischung aus Werten im Register und Werten im RAM–Stapel. Für den Forth–Anwender ist das für gewöhnlich gar nicht sichbar. Da erscheint der Stack *logisch* geordnet, TOS oben, und darunter alle anderen Werte. Die oberhalb des TOS im RAM aufgestapelten Faltkonstanten sind nur während der Compilation vorhanden.

Logische Stacknotation:	...	5OS	4OS	3OS	NOS	TOS	Faltkonstanten
Der echte Stack:	...	Stack	Stack	Stack	Stack	r6	(Nur während der Kompilation)

Um die während der Kompilation auftretenden Zahlen und Konstanten kümmert sich die Konstantenfaltung, die hier zur Voraussetzung fürs Gelingen geworden ist. Doch die Konstantenfaltung ist nur der „Taschenrechner“. Um etwas zu bewirken, wird jede Definition früher oder später auf nicht mehr faltbare Operationen stoßen, die eben keine konstanten Ergebnisse mehr liefern. Und dann wird es richtig spannend!

Zuallererst ein kleines Beispiel, welches eine Binärzahl in den *Gray–Code* umwandelt. Der Gray–Code hat übrigens die besondere Eigenschaft, dass sich beim Zählen mit jedem Schritt nur ein einziges Bit ändert! Für Neugierige empfiehlt sich die Lektüre des Wikipedia–Artikels [Gray-Code] dazu.

Zum besseren Verständnis stellen wir den Forth–Code

```
: >gray ( u -- x ) dup 1 rshift xor ;
```

und den vom Registerallokator daraus generierten Maschinencode in einer Tabelle gegenüber.

Forth	3OS	NOS	TOS	Faltung	Assembler	Kommentar
: >gray						
	Stack	Stack	r6	keine		Zu Beginn ist der Stack im altbekannten Grundzustand.
dup	Stack	r6	r6	keine		DUP benötigt ein Element, und das ist schon vorhanden. So muss nur noch aktualisiert werden, wo sich NOS gerade befindet.
1	Stack	r6	r6	#1		Die Konstante wird zunächst über die Faltung entgegengenommen, ...
rshift	r6	r6	#1	keine		... und muss, weil kein zweiter Operand fürs Falten zur Verfügung steht, in das Stackmodell hineingeschoben werden. 3OS war noch auf dem Stack, also muss beim Hineinschieben selbst noch kein Opcode geschrieben werden.
					lsrs r3 r6 #1	Da r6 zweimal im Modell auftaucht, darf dieses Register nicht verändert werden, ...

	Stack	r6	r3	keine		...das Ergebnis der Rechnung benötigt also ein neues Register. r3 war gerade frei.
xor					eors r6 r3	XOR ist kommutativ — in solchen Fällen wird wenn möglich immer r6 gewählt.
	Stack	Stack	r6	keine		
;	Stack	Stack	r6	keine	bx lr	Der Stack war von selbst schon wieder im Grundzustand — Aufräumen unnötig!

Die Bedeutung der Spalten der vorstehenden Tabelle ist:

Forth	Das auszuführende Forthwort.
3OS	Der dritte Wert auf dem Stack.
NOS	Der zweite (nächste) Wert auf dem Stack.
TOS	Top of Stack, der oberste Wert.
Falten	Die Faltkonstante(n).
Assembler	Das, was assembliert werden wird, in mnemonische Form. Also der Maschinencode, den der Registerallokator daraus generiert hat.

Vermutlich ist es jetzt schon ein bisschen klarer geworden: Im Hintergrund aktualisieren die Stackjongleure nur noch das Stackmodell, welches stets für sich mitschreibt, wo sich welches Element gerade befindet. Operationen wirken dann wenn möglich direkt zwischen den Elementen, wo immer sie auch gerade sind — und das kann eben nicht nur auf dem Stack, sondern auch in einem Register sein. Außerdem wird dabei im Stackmodell noch unterschieden, ob es sich um Register oder Konstanten handelt, anstelle diese gleich in Register zu laden, denn manche Opcodes können (manche) Konstanten direkt in sich aufnehmen. Ganz am Ende muss aufgeräumt werden, damit die Schnittstelle zwischen den Definitionen einheitlich bleibt.

Damit es klappt, müssen die Definitionen, für die es im Prozessor entsprechende Befehle gibt, die auch zwischen Registern wirken können, jeweils einen Anhang bekommen, der einen Blick auf den aktuellen Zustand des Stackmodells wirft, passende Befehle generiert und anschließend das Stackmodell aktualisiert.

Wenn Kontrollstrukturen hinzukommen oder andere Definitionen aufgerufen werden sollen, die das Stackmodell nicht verstehen können, muss auch mittendrin immer mal wieder aufgeräumt werden.

Die folgende Tabelle zeigt ein etwas längeres Beispiel, für eine Art Exponentialfunktion. Hier zunächst der Code:

```
: bitexp ( u -- u )
  dup 247 u>
  if drop $F0000000
  else
    dup 16 u<= if 1 rshift
               else
                 dup ( u u )
                 7 and 8 or ( u b )
                 swap ( b u )
                 3 rshift 2 - lshift
               then
  then
  1-foldable ;
```

Forth	3OS	NOS	TOS	Faltung	Assembler	Kommentar
: bitexp						
	—	—	r6	—		Zu Beginn ist der Stack im Grundzustand.
dup	—	r6	r6	—		Ein Element war schon da — Stackmodell aktualisieren reicht.
247	—	r6	r6	#247		
u>	r6	r6	#247	—		u> kann nicht gefaltet werden, also wird die Konstante in das Stackmodell hineingeschoben.
	—	—	r6	—	cmp r6 #247	Der Vergleich wird generiert... Und es wird vermerkt, dass jetzt entweder ein bedingter Sprung oder ein *Flagschreiber* folgen muss:
if	—	—	r6	—	push {lr}	
	—	—	r6	—	bls ...	Da IF folgt, kann der bedingte Sprung direkt, ohne ein Forth–Flag zu generieren, benutzt werden! IF hätte den Stack nun eigentlich aufräumen müssen — aber das war er ja bereits von selbst.
drop	—	—	—	—		DROP wirft ein Element weg, eins war auch gerade da, passt.
$F0000000	—	—	—	$F0000000		Eine Konstante kommt geflogen...

else						ELSE muss als Kontrollstruktur nun aufräumen, und schreibt die Konstante in r6, um zum Grundzustand zurückzukehren.
					movs r6 $F0	Leider sind Konstanten im Cortex M0 nicht so ganz einfach zu generieren, ...
					lsls r6 r6 $18	... deshalb sind zwei Opcodes dafür nötig.
	—	—	r6	—		Fertig aufgeräumt!
					b ...	ELSE bereitet nun den unbedingten Sprung ans Ende der Struktur vor und vervollständigt den schon von IF vorbereiteten bedingten Sprung an sein Ziel.
dup	—	r6	r6	—		Gleich nochmal!
16	—	r6	r6	#16		
	r6	r6	#16	—		
u<=	—	—	r6	—	cmp r6 #16	
if	—	—	r6	—	bhi ...	Sprung zum Else–Zweig vorbereiten.
1	—	—	r6	#1		
rshift	—	r6	#1	—		
					lsrs r6 r6 #1	
	—	—	r6	—		
else	—	—	r6	—	b ...	ELSE findet den Stack schon ordentlich vor, vervollständigt den bhi–Sprung und bereitet den eigenen Sprung ans Ende der Struktur vor.
dup	—	r6	r6	—		
7	—	r6	r6	#7		
and	r6	r6	#7	—	lsls r3 r6 #0	Jetzt wird es interessant: Bevor der Befehl ausgeführt werden kann, muss eine Kopie des doppelt verwendeten Registers angelegt werden. Leider kann der M0 keine Konstanten in den AND–Opcode mit aufnehmen.
					movs r0 #7	Sie wird deshalb in Register r0 generiert, welches als Konstantenregister fungiert.
					ands r3 r0	Endlich kann der eigentliche Befehl geschrieben werden!
	—	r6	r3	—		
8	—	r6	r3	#8		So ähnlich wiederholt es sich sogleich noch einmal:
or	r6	r3	#8	—		Ins Stackmodell hineinschieben, ...
					movs r0 #8	... Konstante in r0 vorbereiten, ...
					orrs r3 r0	... und den Oder–Befehl schreiben.
	—	r6	r3	—		
swap	—	r3	r6	—		SWAP vertauscht nur die beiden oberen Elemente im Stackmodell.
3	—	r3	r6	#3		
rshift	r3	r6	#3	—	lsrs r6 r6 #3	Das Ergebnis kann gleich in r6 bleiben — das Register wird ja gerade nur einmal verwendet.
	—	r3	r6	—		
2	—	r3	r6	#2		
—	r3	r6	#2		subs r6 #2	Der Minus–Befehl kann Konstanten in sich aufnehmen — praktisch an dieser Stelle.
	—	r3	r6	—		
lshift					lsls r3 r6	LSHIFT braucht zwei Elemente, beide sind schon in Registern.
	—	—	r3	—		Jetzt allerdings wurde einmal nicht der Grundzustand erreicht: TOS ist momentan in r3!
then					movs r6 r3	THEN muss also aufräumen, um wieder den Grundzustand zu erreichen.

	—	—	r6	—		Anschließend ist noch der unbedingte Sprung fertig zu machen, den ELSE vorbereitet hat.
then	—	—	r6	—		Wir hatten noch einen weiteren ELSE–Zweig offen! Auch dort den Sprung fertig machen.
1–foldable						Dies ist immediate und wird mit dem Stack–Grundzustand klassisch aufgerufen, jedoch wird nur ein Flag gesetzt, so dass hier keine Befehle entstehen.
;	—	—	r6	—	pop {pc}	Alles ordentlich — Fertig!

Das „Ergebnis“ noch einmal kurz zusammengefasst:

```
see bitexp
00006C5A: 2EF7 cmp r6 #F7
00006C5C: B500 push { lr }
00006C5E: D902 bls 00006C66
00006C60: 26F0 movs r6 #F0
00006C62: 0636 lsls r6 r6 #18
00006C64: E00C b 00006C80
00006C66: 2E10 cmp r6 #10
00006C68: D801 bhi 00006C6E
00006C6A: 0876 lsrs r6 r6 #1
00006C6C: E008 b 00006C80
00006C6E: 0033 lsls r3 r6 #0
00006C70: 2207 movs r2 #7
00006C72: 4013 ands r3 r2
00006C74: 2208 movs r2 #8
00006C76: 4313 orrs r3 r2
00006C78: 08F6 lsrs r6 r6 #3
00006C7A: 3E02 subs r6 #2
00006C7C: 40B3 lsls r3 r6
00006C7E: 461E mov r6 r3
00006C80: BD00 pop { pc }
```

Es ist erstaunlich, aber wahr: Die Definition wurde gänzlich ohne Stackbewegungen kompiliert!

Spätestens jetzt wird jedoch auch klar, wie viele kleine und große Fälle bei dieser Art der Optimierung berücksichtigt werden müssen. Dazu kommt, dass der Thumb–Befehlssatz vom ARM leider ziemlich unregelmäßig ist, weswegen der Teufel wieder mal im Detail steckt. Leider ist auch das Generieren von Konstanten auf dieser Architektur notorisch schwierig und mit vielen Sonderfällen verbunden, wenn eine möglichst kurze Lösung angestrebt wird.

Um so etwas zu implementieren, ist also zunächst einmal als Infrastruktur eine gut funktionierende Konstantenfaltung nötig, von der ausgehend die Helferlein entwickelt werden können, die das Stackmodell pflegen und sich um die Rückkehr in den Grundzustand kümmern. Da diese Funktion bei größeren Definitionen sehr intensiv verwendet wird, muss das Aufräumen so kurz wie möglich funktionieren, um nicht den Vorteil aller Optimierungen durch lange Aufräum–Befehlssequenzen wieder zunichte zu machen. Anschließend müssen die Stackjongleure auf das Stackmodell wirken — wobei es nützlich ist, wenn Register auch mehrfach auftreten dürfen und nur bei Bedarf eine Kopie angelegt wird. Beim Quadrieren

```
: sqr ( n -- n^2 ) dup * ;
```

wird es sofort anschaulich klar, wieso:

```
see sqr
00006B16: 4376 muls r6 r6
00006B18: 4770 bx lr
ok.
```

Wenn Faltung, das Stackmodell und die Stackjongleure funktionieren und auch das Aufräumen klappt, können dann Stück für Stück alle anderen optimierbaren Fälle mit eingepflegt werden. Auch einige der Kontrollstrukturen nehmen aktiv teil — `IF` kann einen bedingten Sprung annehmen, oder `DO` kann die Schleifenparameter direkt übernehmen!

```
: Einmalzwei ( n -- ) 1+ 0 do i 2 * . loop ;
5 Einmalzwei 0 2 4 6 8 10  ok.
```

```
see Einmalzwei
00007C3A: 1C76  adds r6 r6 #1
00007C3C: B500  push { lr }
00007C3E: B430  push { r4  r5 }
00007C40: 2400  movs r4 #0
00007C42: 0035  lsls r5 r6 #0
00007C44: CF40  ldmia r7 { r6 }
00007C46: 0023  lsls r3 r4 #0
00007C48: 2002  movs r0 #2
00007C4A: 4343  muls r3 r0
00007C4C: 3F04  subs r7 #4
00007C4E: 603E  str r6 [ r7 #0 ]
```

```
: Einmalzwei
  1+
  \ Kontrollstruktur erfordert das Sichern von LR für den Rücksprung
  do
  \ Index = 0
  \ Limit direkt aus r6 übernehmen
  \ Aufräumen in den Grundzustand
  i
  2
  *
  \ Aufräumen in den Grundzustand
```

```
00007C50: 461E  mov r6 r3
00007C52: F7FC  bl  00004654  --> .    .
00007C54: FCFF
00007C56: 3401  adds r4 #1             loop
00007C58: 42AC  cmp r4 r5
00007C5A: D1F4  bne 00007C46
00007C5C: BC30  pop { r4  r5 }
00007C5E: BD00  pop { pc }           ;
```

Grundsätzlich können so viele Elemente wie gewünscht in Registern gehalten werden — wichtig ist jedoch, von Anfang an Interrupts mit im Auge zu behalten. Während nämlich ein sauber implementierter Stack schon von sich aus interruptsicher ist, verhalten sich die bei dieser Optimierung herumfliegenden Register zwischendurch nicht mehr wie ein Stack. Der Forth–Programmierer bemerkt davon nichts, schließlich befindet sich das Stackmodell an allen Kanten und Übergängen im Grundzustand — ein Interrupt kann jedoch jederzeit auftreten, und sämtliche Register, die für die Optimierungen verwendet werden, müssen dementsprechend gesichert werden. Im ARM werden die Register r0, r1, r2, r3 und r12 automatisch bei einem Interrupt auf den Returnstack gelegt.

Mecrisp–Stellaris verwendet r0, r1, r2 und r3 als zusätzliche Stackelemente, r4 und r5 für do–loop–Schleifen, r6 als TOS und r7 als Stackpointer. Da am Anfang und am Ende stets der Grundzustand angenommen und verwendet wird, wird somit r6 auch stets wie „TOS“ behandelt und geht damit nicht verloren — auch, wenn zum Zeitpunkt des Interrupts r6 eigentlich etwas ganz anderes gewesen sein könnte. Da die höheren Register r8 bis r12 im Cortex M0 nur schwer zugänglich sind, werden sie gar nicht verwendet. r13, r14 und r15 schließlich sind als Link–Register, Return–Stackpointer und Programmzähler funktional festgelegt.

Mit fünf Elementen lassen sich fast alle grundlegenden Forth–Aufgaben recht gut optimieren; die größte Tiefe, die im Kern noch regulär auftaucht, ist bei 2rot mit sechs Elementen erreicht.

Beim MSP430 wäre es viel leichter zu implementieren gewesen — die Architektur hat mehr Register, die direkt in Rechnungen verfügbar sind, der Befehlssatz ist orthogonal, alle Opcodes können direkt auf den Speicher zugreifen oder Konstanten tragen... Doch genau diese Vorzüge sind auch der Grund, weswegen es dort nicht so dringend nötig ist: Im MSP430 lassen sich Rechnungen und Logik auch direkt auf dem Stack halbwegs effizient implementieren! Beim ARM hingegen, wo jedes Element zuerst aus dem Speicher in ein Register geladen werden muss, bevor darauf zugegriffen werden kann, bietet sich diese Art der Optimierung einfach an.

Eins soll allerdings nicht verschwiegen werden: Auch wenn die kompilierten Definitionen kürzer und schneller sind, so wiegen die vielen nötigen Optimierfälle im Kern doch schwer. Platzmäßig können so erst bei sehr großen Forth–Projekten Vorteile erzielt werden, weswegen sich die neue Variante von Mecrisp–Stellaris eher für diejenigen Projekte eignet, bei denen es vor allem auf Geschwindigkeit ankommt.

Wer bis hierhin tapfer gefolgt ist, mag vielleicht noch einen Schritt weiter denken: Wieso können nicht die Stackelemente auch über die Grenzen von Kontrollstrukturen hinweg direkt in Registern übergeben werden ? Der Gedanke ist gut und richtig — grundsätzlich lässt sich umso mehr optimieren, je mehr gepuffert und abgewartet wird, bevor tatsächlich Befehle generiert werden. Allerdings müsste dafür viel Speicher reserviert werden, prinzipiell müsste die komplette Definition zwischengelagert werden, bevor mit dem Kompilieren begonnen werden kann. Als nächster Schritt — womit wir bei globalen Optimierungen angelangt wären — könnten beispielsweise auch der Lesbarkeit halber ausgelagerte Helfer–Definitionen, die nur einmal verwendet werden, direkt an ihrem Einsatzort eingefügt und überoptimiert werden und bräuchten keinen eigenen Eintrag im Dictionary mehr.

Doch hier ist nun eine Grenze zu ziehen zwischen dem Platzbedarf des Compilers selbst und dem Gewinn, der mit ihm erzielt werden kann. Bei kleinen Chips, deren Speicher im Bereich einiger Kilobytes liegt, ist die Grenze mit dem hier beschriebenen Ansatz schon mehr als erreicht, bei größeren Microcontrollern könnte es sich bei großen Projekten auch beim Platzbedarf rechnen. Doch selbst, wenn noch einiges möglich wäre, so sind die immer kleiner werdenden Verbesserungen nur noch mit einem immer größeren Aufwand im Compiler zu erreichen. Dies ist übrigens auch der Grund, weswegen die kommerziell verfügbaren optimierenden Forth–Compiler allesamt auf einem großen Host–System laufen!

Für die ganz Neugierigen folgt nun noch ein tieferer Einblick in die Implementation und die innere Funktionsweise des Stackmodells !

Innendrin besteht der Registerallokator zunächst einmal aus einem Satz von Variablen, die das Stackmodell formen:

```
ramallot state_tos, 4
ramallot constant_tos, 4
ramallot state_nos, 4
ramallot constant_nos, 4
ramallot state_3os, 4
ramallot constant_3os, 4
ramallot state_4os, 4
ramallot constant_4os, 4
ramallot state_5os, 4
```

```
ramallot constant_5os, 4
ramallot sprungtrampolin, 4
ramallot state_r0, 4
ramallot constant_r0, 4
```

In den State–Variablen wird vermerkt, in welchem Zustand sich die einzelnen Stackelemente gerade befinden. Diese können entweder noch auf dem Stack liegen und damit unbekannt sein, sich in einem Register befinden, oder eine bekannte Konstante sein, die dann in dem dazugehörigen Constant–Register vermerkt wird.

Der kanonische Zustand, also der Grundzustand, der am Anfang und am Ende, vor Kontrollstrukturen und Definitionsaufrufen angestrebt wird, ist: TOS in r6, alle anderen Stackelemente auf dem Stack und unbekannt.

Sämtliche Definitionen, die über den Registerallokator optimiert werden können, tragen am Ende der Definition noch einen „Optimieranhang“, in dem die Instruktionen generiert werden, die die Funktion der Definition mit den gerade im Stackmodell befindlichen Elementen erfüllen.

Zwei kleine Beispiele helfen bestimmt, es zu veranschaulichen. Beginnen wir mit der Definition von `SWAP` .

```
@ -------------------------------------------------------------------------
  Wortbirne Flag_foldable_2|Flag_inline|Flag_allocator, "swap" @ ( x y -- y x )
@ -------------------------------------------------------------------------
  ldr r1,  [psp]
  str tos, [psp]
  movs tos, r1
  bx lr
    push {r2, r3, lr} @ Spezialeinsprung des Registerallokators:
    bl expect_two_elements
    @ TOS und NOS vertauschen.
    ldr r2, [r0, #offset_state_tos]
    ldr r3, [r0, #offset_state_nos]
    str r3, [r0, #offset_state_tos]
    str r2, [r0, #offset_state_nos]
    ldr r2, [r0, #offset_constant_tos]
    ldr r3, [r0, #offset_constant_nos]
    str r3, [r0, #offset_constant_tos]
    str r2, [r0, #offset_constant_nos]
    pop {r2, r3, pc}
```

Zu Beginn steht die ganz gewöhnliche Definition, die im Ausführmodus beim Interpretieren aufgerufen wird. Nach deren Ende, also nach dem `bx lr` Opcode, beginnt der Optimieranhang, welcher beim Kompilieren ausgeführt wird. Zunächst wird um zwei Elemente gebeten, die von der Infrastruktur des Allokators bereitgelegt werden, vielleicht sind sie schon vorhanden, vielleicht sind es Konstanten, vielleicht müssen die Elemente erst vom Stack in Register nachgeladen werden. Auf jeden Fall sind danach garantiert mindestens zwei Elemente im Stackmodell vorrätig, und Swap muss nur noch die oberen beiden Stackelemente vertauschen.

```
@ -------------------------------------------------------------------------
  Wortbirne Flag_foldable_1|Flag_inline|Flag_allocator, "negate" @ ( n1 -- -n1 )
@ -------------------------------------------------------------------------
  rsbs tos, tos, #0
  bx lr
    pushdaconstw 0x4240 @ rsbs r0, r0, #0
smalltworegisters:
    push {lr}
    bl expect_one_element @ Da nicht gefaltet worden ist, muss es sich um ein Register handeln.
    ldr r1, [r0, #offset_state_tos]
    lsls r1, #3
    orrs tos, r1
    bl tos_registerwechsel
    orrs tos, r3
    bl hkomma
    pop {pc}
```

Negate ist schon ein bisschen aufwändiger. Zunächst wieder die klassische Definition, welche als 1–faltbar markiert ist — das bedeutet, dass negate, sollte es auf eine Konstante wirken, von der Konstantenfaltung, welche vor dem Stackmodell tätig wird, sofort ausgeführt wird. Damit ist es nicht nötig, im Optimierteil den Spezialfall einer Konstanten in TOS zu betrachten. Mindestens

ein Element wird also bereitgelegt, und es ist garantiert keine Konstante. Damit muss es sich um einen Register handeln. Zunächst wird also das Quellregister in den Opcode eingebaut, anschließend wird ein Zielregister angefordert und ebenso hinzugefügt. Dieses kann, muss aber nicht, das gleiche Register sein wie jenes, das zu Beginn TOS enthalten hat — aus zwei Gründen: Nach einem `DUP` oder `OVER` kann es sein, dass zwei Stackelemente im gleichen Register liegen, um ein vielleicht unnötiges Hin–und–Her–Kopieren der Register zu vermeiden. In diesem Fall kann hier die Möglichkeit genutzt werden, dass für den RSBS–Opcode von Negate Quell- und Zielregister separat angegeben werden können. Sollte also das TOS–enthaltende Register noch für ein anderes Element genutzt werden, darf der Registerinhalt nicht verändert werden, in diesem Fall wird einfach ein anderes, gerade freies Zielregister zurückgegeben, so dass für das vorangehende DUP gar kein Befehl generiert zu werden braucht. Der zweite Grund liegt darin, dass es am Ende der Definition zur Rückkehr in den kanonischen Stackzustand nützlich ist, wenn das oberste Element TOS schon von selbst in r6 liegt. Sollte also r6 zufällig gerade nicht belegt sein, so wird das Ergebnis von `negate` damit bevorzugt in r6 abgelegt. Anschließend kann der Opcode geschrieben werden und die Kompilation kann fortschreiten.

Für andere Instruktionen, wie beispielsweise `XOR` oder `*` muss im Cortex M0 eines der Quellregister mit dem Zielregister identisch sein — in solchen Fällen wird dann vorher noch ein Kopierbefehl in ein gerade freies Register eingefügt, falls das Element danach noch weiter benötigt wird. Dadurch, dass dies jedoch nur bei Bedarf und nicht stets bei einem `DUP` oder `OVER` geschieht, kann viel Platz gespart werden, wie bereits weiter vorne in `SQR` gezeigt worden ist.

Das Sprungtrampolin dient dazu, die häufig auftretende Kombination eines Vergleiches mit einer ein Flag erwartenden Kontrollstruktur zu optimieren, indem das Flag über das Statusregister des Prozessors und nicht über ein „richtiges“ Forth–Flag auf dem Stack übergeben wird.

```
: 42u< ( u -- ? )
  42 u< ; ok.
see 42u<
00006B80: 2E2A cmp r6 #2A
00006B82: 41B6 sbcs r6 r6
00006B84: 4770 bx lr
ok.

: 42u<? ( u -- u )
  dup 42 u< if ." Kleiner" then ; ok.
see 42u<?
00006BC4: 2E2A cmp r6 #2A
00006BC6: B500 push { lr }
00006BC8: D205 bcs 00006BD6
00006BCA: F7FC bl 00002C50 --> .' Kleiner'
00006BCC: F841
00006BCE: 4B07
00006BD0: 656C
00006BD2: 6E69
00006BD4: 7265
00006BD6: BD00 pop { pc }
ok.
```

Während im ersten Fall ein Forth–Flag entsteht, wird im zweiten Fall das Ergebnis des Vergleichs ohne Umweg über den Stack direkt mit einem bedingten Sprung ausgewertet. Da eine Kontrollstruktur folgt, muss für einen vielleicht folgenden Definitionsaufruf in einem der Zweige das LR–Register für den Rücksprung vorher gesichert werden.

Schließlich existiert noch das Paar `state_r0` und `constant_r0` — oft geschieht es nämlich, dass die Adressen von Variablen oder IO–Ports nahe beieinander liegen und mehrmals verwendet werden, sollte also r0 gerade nicht vom Stackmodell benötigt werden, steht dieses Register zur Verfügung, eine Adresskonstante zwischenzuspeichern.

```
27 variable e ok.
31 variable p ok.
0 variable s ok.
ok.
: summe ( -- )
  e @ p @ + s ! ; ok.
see summe
00006C38: 2080 movs r0 #80
00006C3A: 0400 lsls r0 r0 #10
00006C3C: 30BE adds r0 #BE
00006C3E: 0180 lsls r0 r0 #6
00006C40: 6A83 ldr r3 [ r0 #28 ]
00006C42: 6A42 ldr r2 [ r0 #24 ]
00006C44: 189B adds r3 r3 r2
00006C46: 6203 str r3 [ r0 #20 ]
00006C48: 4770 bx lr
ok.
```

Wie hier zu sehen ist, wird nur einmal eine Adresse in r0 geladen — die einzelnen Variablen können dann vorteilhaft über entsprechende Offsets in den LDR und STR Opcodes direkt adressiert werden.

Natürlich arbeiten alle hier beschriebenen Funktionen auch reibungslos zusammen, was mit einem „echten“ Beispiel gezeigt werden soll:

```
0 variable disasm-$
: disasm-fetch ( -- Befehl )
  disasm-$ @ h@
  2 disasm-$ +!
;
see disasm-fetch
00005070: 2080 movs r0 #80
00005072: 0400 lsls r0 r0 #10
00005074: 30BE adds r0 #BE
00005076: 0180 lsls r0 r0 #6
00005078: 6B03 ldr r3 [ r0 #30 ]
0000507A: 881B ldrh r3 [ r3 #0 ]
0000507C: 6B02 ldr r2 [ r0 #30 ]
0000507E: 3202 adds r2 #2
00005080: 6302 str r2 [ r0 #30 ]
00005082: 3F04 subs r7 #4
00005084: 603E str r6 [ r7 #0 ]
```

```
00005086: 461E mov r6 r3
00005088: 4770 bx lr
```

In den ersten vier Befehlen wird die Adresse von disasm–$ in r0 geladen, anschließend wird in zwei Befehlen die Instruktion geholt, auf die die Variable gerade gezeigt hat.

Anschließend wird disasm–$ in drei Befehlen um zwei erhöht, schließlich wird der Stack am Ende der Definition wieder in den kanonischen Zustand versetzt.

Referenzen

[Gray-Code] `https://de.wikipedia.org/wiki/Gray-Code`

ARM–Cortex und die Steckplatine

Und zum guten Schluss noch ein Kapitel für all diejenigen, die schon immer einmal zum Kennenlernen mit einem kleinen ARM–Cortex auf Lochraster oder Steckplatine basteln wollten.

Ganz zu Beginn: Keine Angst!

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

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

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

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

Sowie für die hier beschriebenen Experimente noch

1 RGB–Leuchtdiode mit gemeinsamer Kathode.

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

Die beiden Dioden sorgen für je 0.7V Spannungsabfall und so bleiben von den 5 V aus dem USB–Anschluss 3,6 V übrig. Die Leuchtdiode und ihr Vorwiderstand sind wirklich wichtig, weil sonst die Spannung ohne Last langsam ansteigen könnte. Wer mag und hat, kann natürlich stattdessen einen Spannungsregler verwenden. Wenn während des Loslassens des Reset–Tasters der Bootloader–Taster gedrückt ist, springt der Chip in den Bootloader–Modus und erwartet unsere Wünsche. Jetzt kann bereits Mecrisp–Stellaris geflasht werden. Dafür gibt es unter Linux LPC21ISP (`http://sourceforge.net/projects/lpc21isp/`) welches so aufgerufen wird: `lpc21isp mecrisp-stellaris-lpc1114fn28.hex /dev/ttyUSB0 9600 12000` Wer Windows benutzt, wird mit Flash Magic (`http://www.flashmagictool.com/`) fündig. Anschließend ist ein Terminal mit 115200 Baud 8N1 zu öffnen und Mecrisp–Stellaris begrüßt uns nach einem weiteren Druck des Reset–Tasters.

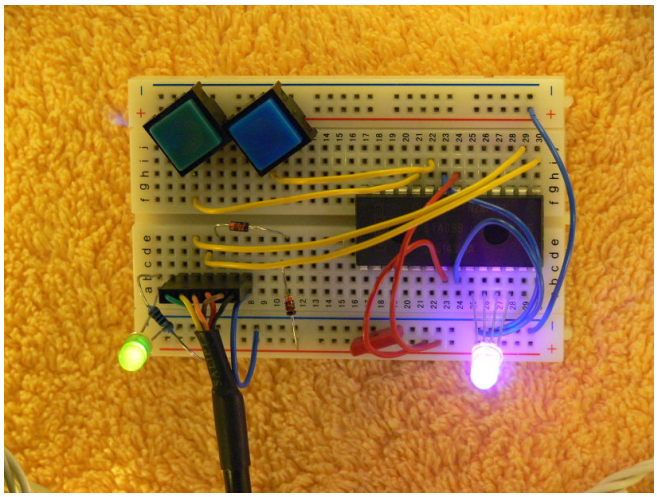


Abbildung 2: Steckbrett

Auf Entdeckungsreise

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

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

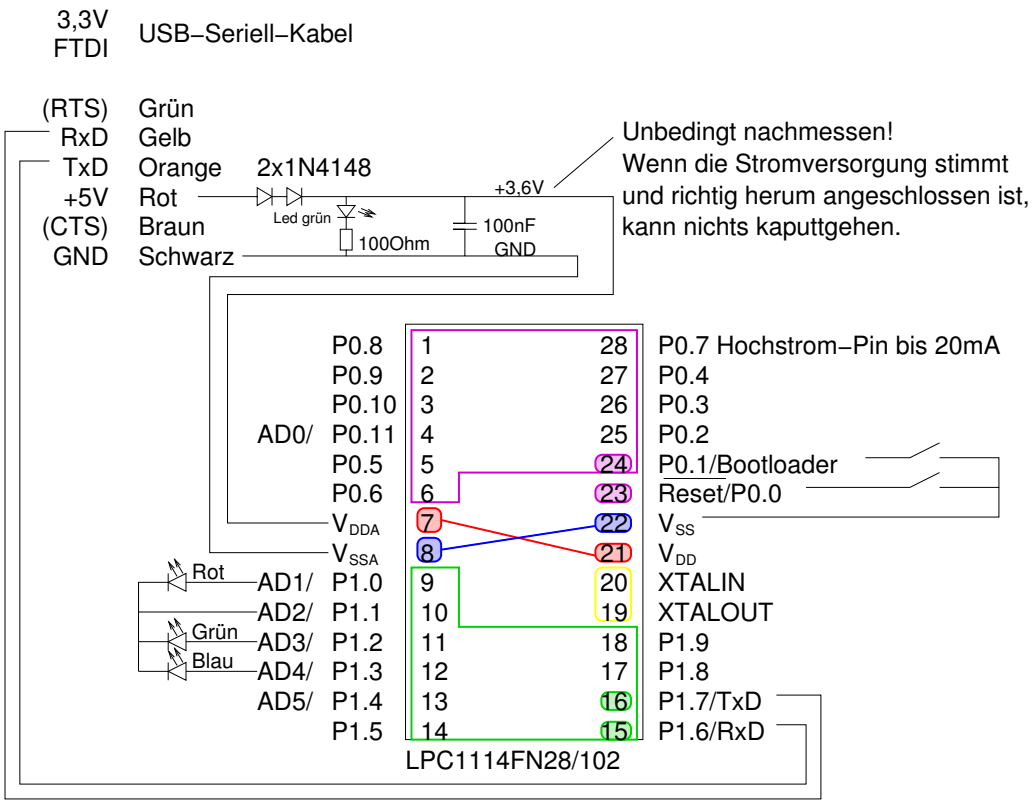


Abbildung 1: Schaltplan

seien hier die passenden Werte für die ersten Experimente fertig herausgesucht:

```
\ P0.1 als digitalen IO-Pin mit
\ internem Pullup für den Taster konfigurieren:
$D0 $40044010 !
\ Pins als digitale IO-Pins ohne
\ Sonderfunktionen konfigurieren:
$C1 $40044078 ! ( P1.0 ... )
$C1 $4004407C ! ( P1.1 ... )
$C1 $40044080 ! ( P1.2 ... )
$C1 $40044090 ! ( P1.3 ... )
```

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

```
$50003FFC constant P0DATA
$50008000 constant P0DIR
$50013FFC constant P1DATA
$50018000 constant P1DIR
```

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

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

```
: rotgrün ( -- ) begin taster?
    if   key? until ;
```

Ein kleiner Trick

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

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

```
$40048080 constant SYSAHBCLKCTRL
  \ Clock control register
$40048238 constant PDRUNCFG
  \ Power-down configuration register
$40044078 constant IOCON_PIO1_0
$4004407C constant IOCON_PIO1_1
```

```
$40044080 constant IOCON_PIO1_2
$40044090 constant IOCON_PIO1_3
$50013FFC constant P1DATA
$50018000 constant P1DIR
$4001C000 constant ADOCR
  \  AD-Wandler Control Register
$4001C004 constant ADOGDR
  \            Global Data Register
: farbsensor-init ( -- )
       \ Vorbereitungen für den Farbsensor
  1 13 lshift SYSAHBCLKCTRL bis!
       \ Takt für AD-Wandler einschalten
  1  4 lshift PDRUNCFG bic!
       \ Stromaus-Bit des AD-Wandlers löschen
  $C1 IOCON_PIO1_1 !
       \ P1.1 sei digitaler IO-Pin
       \ ohne Sonderfunktionen,
  2 P1DIR !      \     Ausgang
  0 P1DATA !     \           low
  $42 IOCON_PIO1_0 ! \ P1.0 sei analoger Eingang
  $42 IOCON_PIO1_2 ! \ P1.2 ...
  $42 IOCON_PIO1_3 ! \ P1.3 ... ;
: analog ( Kanal -- Messwert )
\ Einen analogen Eingang wandeln
  1 swap lshift \ Kanal wählen
  4  8 lshift or
  \ CLKDIV = 4 --> 12 Mhz / 4
  \        = 3 MHz < 4.5 MHz Maximum.
  1 24 lshift or  \ Messung starten
  ADOCR !
  begin 1 31 lshift ADOGDR bit@ until
      \ Auf das Wandlungsende warten
  ADOGDR @ 6 rshift $3FF and
      \ Messergebnis passend schieben ;
: farbsensor ( -- )
  farbsensor-init
  begin
    1 analog .-Rot: -u.
    3 analog .-Grün: -u.
    4 analog .-Blau: -u.
    cr
  key? until ;
```

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

Wem dieses Kapitel irgendwie bekannt vorkommt, ja, dieser Teil wurde schon im Forth-Magazin *Vierte Dimension*, Heft 3–4/2014, vorab gedruckt. Und nun hinzugefügt, damit alles beisammen ist. Viel Vergnügen damit!

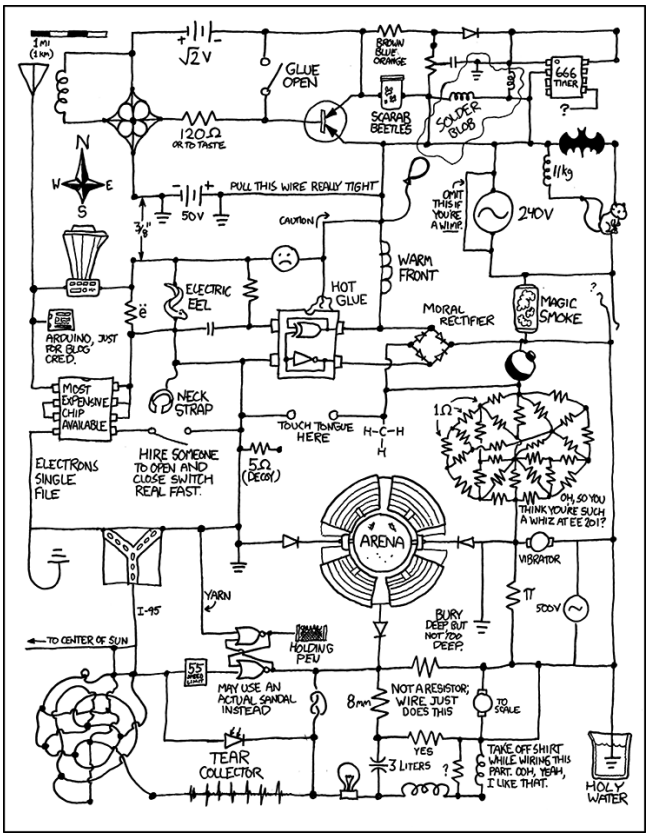


Abbildung 3: `http://xkcd.com/730`

Mecrisp Glossar 2.2.0

Liste der Worte, versammelt in funktionellen Gruppen

Terminal-IO (exactly ANS, some logical extensions)

```
emit?           ( -- Flag ) Ready to send a character ?
key?            ( -- Flag ) Checks if a key is waiting
key             ( -- Char ) Waits for and fetches the pressed key
emit            ( Char -- ) Emits a character.

hook-emit?      ( -- a-addr ) Hooks for redirecting
hook-key?       ( -- a-addr )   terminal IO
hook-key        ( -- a-addr )     on the fly
hook-emit       ( -- a-addr )

serial-emit?    ( -- Flag )  Serial interface
serial-key?     ( -- Flag )    terminal routines
serial-key      ( -- Char )      as default communications
serial-emit     ( Char -- )

hook-pause      ( -- a-addr ) Hook for a multitasker
pause           ( -- )        Task switch, none for default
```

Stack Jugglers (exactly ANS, some logical extensions)

Single-Jugglers:

```
depth           ( -- +n ) Gives number of single-cell stack items.
nip             ( x1 x2 -- x2 )
drop            ( x -- )
rot             ( x1 x2 x3 -- x2 x3 x1 )
-rot            ( x1 x2 x3 -- x3 x1 x2 )
swap            ( x1 x2 -- x2 x1 )
tuck            ( x1 x2 -- x2 x1 x2 )
over            ( x1 x2 -- x1 x2 x1 )
?dup            ( x -- 0 | x x )
dup             ( x -- x x )
pick            ( ... xi+1 xi ... x1 x0 i -- ... x1 x0 xi )
                          Picks one element from deep below

>r              ( x -- ) (R: -- x )
r>              ( -- x ) (R: x -- )
r@              ( -- x ) (R: x -- x )
rdrop           (  --  ) (R: x -- )
rdepth          ( -- +n ) Gives number of return stack items.
rpick           ( i -- xi ) R: ( ... xi ... x0 -- ... xi ... x0 )
```

Double-Jugglers: They perform the same for double numbers.

```
2nip            ( x1 x2 x3 x4 -- x3 x4 )
2drop           ( x1 x2 -- )
2rot            ( x1 x2 x3 x4 x5 x6 -- x3 x4 x5 x6 x1 x2 )
2-rot           ( x1 x2 x3 x4 x5 x6 -- x5 x6 x1 x2 x3 x4 )
2swap           ( x1 x2 x3 x4 -- x3 x4 x1 x2 )
2tuck           ( x1 x2 x3 x4 -- x3 x4 x1 x2 x3 x4 )
2over           ( x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 )
2dup            ( x1 x2 -- x1 x2 x1 x2 )

2>r             ( x1 x2 -- ) (R: -- x1 x2 )
```

```
2r>             ( -- x1 x2 ) (R: x1 x2 -- )
2r@             ( -- x1 x2 ) (R: x1 x2 -- x1 x2 )
2rdrop          ( -- )       (R: x1 x2 -- )
```

Stack pointers:

```
sp@             ( -- a-addr )  Fetch  data stack pointer
sp!             ( a-addr -- )  Store  data stack pointer
rp@             ( -- a-addr )  Fetch return stack pointer
rp!             ( a-addr -- )  Store return stack pointer
```

Logic (exactly ANS, some logical extensions)

```
arshift         ( x1 u -- x2 ) Arithmetric right-shift of u bit-places
rshift          ( x1 u -- x2 ) Logical right-shift of u bit-places
lshift          ( x1 u -- x2 ) Logical  left-shift of u bit-places
shr             ( x1 -- x2 )   Logical right-shift of one bit-place
shl             ( x1 -- x2 )   Logical  left-shift of one bit-place
ror             ( x1 -- x2 )   Logical right-rotation of one bit-place
rol             ( x1 -- x2 )   Logical  left-rotation of one bit-place
bic             ( x1 x2 -- x3 ) Bit clear, identical to "not and"
not             ( x1 -- x2 )    Invert all bits
xor             ( x1 x2 -- x3 ) Bitwise Exclusive-OR
or              ( x1 x2 -- x3 ) Bitwise OR
and             ( x1 x2 -- x3 ) Bitwise AND
false           ( --  0 ) False-Flag
true            ( -- -1 ) True-Flag
clz             ( x1 -- u ) Count leading zeros
```

Calculus for single numbers (exactly ANS, some logical extensions)

```
u/mod            ( u1 u2 -- u3 u4 ) 32/32 = 32 rem 32 Division
                                    u1 / u2 = u4 remainder u3
/mod            ( n1 n2 -- n3 n4 ) n1 / n2 = n4 rem n3
mod             ( n1 n2 -- n3 ) n1 / n2 = remainder n3
/               ( n1 n2 -- n3 ) n1 / n2 = n3
*               ( u1|n1 u2|n2 -- u3|n3 ) 32*32 = 32 Multiplication
min             ( n1 n2 -- n1|n2 ) Keeps smaller of top two items
max             ( n1 n2 -- n1|n2 ) Keeps greater of top two items
umin            ( u1 u2 -- u1|u2 ) Keeps unsigned smaller
umax            ( u1 u2 -- u1|u2 ) Keeps unsigned greater
2-              ( u1|n1 -- u2|n2 ) Subtracts two, optimized
1-              ( u1|n1 -- u2|n2 ) Subtracts one, optimized
2+              ( u1|n1 -- u2|n2 ) Adds two, optimized
1+              ( u1|n1 -- u2|n2 ) Adds one, optimized
even            ( u1|n1 -- u2|n2 ) Makes even. Adds one if uneven.
2*              ( n1 -- n2 ) Arithmetric  left-shift
2/              ( n1 -- n2 ) Arithmetric right-shift
abs             ( n -- u )   Absolute value
negate          ( n1 -- n2 ) Negate
-               ( u1|n1 u2|n2 -- u3|n3 ) Subtraction
+               ( u1|n1 u2|n2 -- u3|n3 ) Addition
```

Calculus involving double numbers (exactly ANS, some logical extensions)

```
um*             ( u1 u2 -- ud )         32*32 = 64 Multiplication
ud*             ( ud1 ud2 -- ud3 )      64*64 = 64 Multiplication
udm*            ( ud1 ud2 -- ud3-Low ud4-High ) 64*64=128 Multiplication

um/mod          ( ud u1 -- u2 u3 )      ud / u1 = u3 remainder u2
ud/mod          ( ud1 ud2 -- ud3 ud4 ) 64/64 = 64 rem 64 Division
                                        ud1 / ud2 = ud4 remainder ud3
```

```
m*              ( n1 n2 -- d )          n1 * n2 = d
m/mod           ( d  n1 -- n2 n3 )      d  / n1 = n3 remainder r2
d/mod           ( d1 d2 -- d3 d4 )      d1 / d2 = d4 remainder d3
d/              ( d1 d2 -- d3 )         d1 / d2 = d3
*/              ( n1 n2 n3 -- n4 )      n1 * n2 / n3 = n4
u*/             ( u1 u2 u3 -- u4 )      u1 * u2 / u3 = u4
*/mod           ( n1 n2 n3 -- n4 n5 )   n1 * n2 / n3 = n5 remainder n4
u*/mod          ( u1 u2 u3 -- u4 u5 )   u1 * u2 / u3 = u5 remainder u4

d2*             ( d1 -- d2 ) Arithmetric  left-shift
d2/             ( d1 -- d2 ) Arithmetric right-shift
dshl            ( ud1 -- ud2 ) Logical left-shift, same as d2*
dshr            ( ud1 -- ud2 ) Logical right-shift

dabs            ( d -- ud ) Absolute value
dnegate         ( d1 -- d2 ) Negate
d-              ( ud1|d1 ud2|d2 -- ud3|d3 ) Subtraction
d+              ( ud1|d1 ud2|d2 -- ud3|d3 ) Addition
s>d             ( n -- d ) Makes a signed single number double length
```

Comparisions (exactly ANS, some logical extensions)

Single-Comparisions:

```
u<=             ( u1 u2 -- flag )  Unsigned comparisions
u>=             ( u1 u2 -- flag )
u>              ( u1 u2 -- flag )
u<              ( u1 u2 -- flag )
<=              ( n1 n2 -- flag )  Signed comparisions
>=              ( n1 n2 -- flag )
>               ( n1 n2 -- flag )
<               ( n1 n2 -- flag )
0<              ( n - flag )       Negative ?
0<>             ( x -- flag )
0=              ( x -- flag )
<>              ( x1 x2 -- flag )
=               ( x1 x2 -- flag )
```

Double-Comparisions: They perform the same for double numbers.

```
du>             ( ud1 ud2 -- flag )
du<             ( ud1 ud2 -- flag )
d>              ( d1 d2 -- flag )
d<              ( d1 d2 -- flag )
d0<             ( d -- flag )
d0=             ( d -- flag )
d<>             ( d1 d2 -- flag )
d=              ( d1 d2 -- flag )
```

Tools (not only) for s31.32 fixed point numbers (speciality!)

```
Fixpoint numbers are stored ( n-comma n-whole ) and can be handled
like signed double numbers.

f/              ( df1 df2 -- df3 ) Division of two fixpoint numbers
f*              ( df1 df2 -- df3 ) Multiplication

hold<           ( char -- )
                Adds character to pictured number output buffer
                from behind.
f#S             ( n-comma1 -- n-comma2 )
```

```
                Adds 32 comma-digits to number output
f#              ( n-comma1 -- n-comma2 )
                Adds one comma-digit to number output
f.              ( df -- )
                Prints a fixpoint number with 32 fractional digits
f.n             ( df n -- )
                Prints a fixpoint number with n fractional digits

number          ( Counted-String-Address -- 0 )
                  cstr-addr              -- n 1 )
                                         -- n-low n-high 2 )
                Tries to convert a string to a number.
```

Number base (exactly ANS)

```
binary          ( -- ) Sets base to 2
decimal         ( -- ) Sets base to 10
hex             ( -- ) Sets base to 16
base            ( -- a-addr ) Base variable address
```

Memory access (subtle differences to ANS, special cpu-specific extensions)

```
move            ( c-addr1 c-addr2 u -- ) Moves u Bytes in Memory
fill            ( c-addr u c ) Fill u Bytes of Memory with value c

cbit@           ( mask c-addr -- flag ) Test BIts in byte-location
hbit@           ( mask a-addr -- flag ) Test BIts in halfword-location
bit@            ( mask a-addr -- flag ) Test BIts in word-location

cxor!           ( mask c-addr -- ) Toggle bits in byte-location
hxor!           ( mask a-addr -- ) Toggle bits in halfword-location
xor!            ( mask a-addr -- ) Toggle bits in word-location

cbic!           ( mask c-addr -- ) Clear BIts in byte-location
hbic!           ( mask a-addr -- ) Clear BIts in halfword-location
bic!            ( mask a-addr -- ) Clear BIts in word-location

cbis!           ( mask c-addr -- ) Set BIts in byte-location
hbis!           ( mask a-addr -- ) Set BIts in halfword-location
bis!            ( mask a-addr -- ) Set BIts in word-location

2constant name  ( ud|d -- ) Makes a double constant.
constant  name  ( u|n -- )  Makes a single constant.
2variable name  ( ud|d -- ) Makes an initialized double variable
variable  name  ( n|n -- )  Makes an initialized single variable
nvariable name  ( n1*u|n n1 -- ) Makes an initialized variable with
                                 specified size of n1 words
                                 Maximum is 15 words

buffer: name    ( u -- ) Creates a buffer in RAM with u bytes length

2@              ( a-addr -- ud|d ) Fetches double number from memory
2!              ( ud|d a-addr -- ) Stores double number in memory

@               ( a-addr -- u|n ) Fetches single number from memory
!               ( u|n a-addr -- ) Stores single number in memory
+!              ( u|n a-addr -- ) Add to memory location

h@              ( c-addr -- char ) Fetches halfword from memory
h!              ( char c-addr )    Stores halfword in memory
h+!             ( u|n a-addr -- )  Add to halfword memory location
```

```
c@              ( c-addr -- char ) Fetches byte from memory
c!              ( char c-addr )    Stores byte in memory
c+!             ( u|n a-addr -- )  Add to byte memory location
```

Strings and beautiful output (exactly ANS, some logical extensions)

String routines:

```
type            ( c-addr length -- )
                Prints a string.

s" Hello"       Compiles a string and
                ( -- c-addr length )
                gives back its address and length when executed.

." Hello"       Compiles a string and
                ( -- )
                prints it when executed.

( Comment )     Ignore Comment
\ Comment       Comment to end of line

cr              ( -- ) Emits line feed
bl              ( -- 32 ) ASCII code for Space
space           ( -- ) Emits space
spaces          ( n -- ) Emits n spaces if n is positive

compare         ( caddr-1 len-1 c-addr-2 len-2 -- flag )
                Compares two strings

accept          ( c-addr maxlength -- length ) Read input into a string.
```

Counted string routines:

```
ctype           ( cstr-addr -- )
                Prints a counted string.

c" Hello"       Compiles a counted string and
                ( -- cstr-addr )
                gives back its address when executed.

cexpect         ( cstr-addr maxlength -- ) Read input into a counted string.

count           ( cstr-addr -- c-addr length )
                Convert counted string into addr-length string

skipstring      ( cstr-addr -- a-addr )
                Increases the pointer to the aligned end of the string.
```

Pictured numerical output:

```
.digit          ( u -- char ) Converts a digit to a char
digit           ( char -- u true | false ) Converts a char to a digit

[char] *        Compiles code of following char
                ( -- char ) when executed

char *          ( -- char ) gives code of following char
hold            ( char -- ) Adds character to pictured number
                            output buffer from the front.
```

```
sign            ( n -- ) Add a minus sign to pictured number
                         output buffer, if n is negative

#S              ( ud1|d1 -- 0 0 ) Add all remaining digits
                from the double length number to output buffer
#               ( ud1|d1 -- ud2|d2 ) Add one digit from the
                double length number to output buffer
#>              ( ud|d -- c-addr len )
                Drops double-length number and finishes
                pictured numeric output ready for type
<#              ( -- )    Prepare pictured number output buffer
u.              ( u -- )  Print unsigned single number
.               ( n -- )  Print single number
ud.             ( ud -- ) Print unsigned double number
d.              ( d -- )  Print double number
```

Deep insights:

```
words           ( -- ) Prints list of defined words.
.s              ( many -- many ) Prints stack contents, signed
u.s             ( many -- many ) Prints stack contents, unsigned
h.s             ( many -- many ) Prints stack contents, unsigned, hex
hex.            ( u -- ) Prints 32 bit unsigned in hex base,
                         needs emit only.
                         This is independent of number subsystem.
```

User input and its interpretation (exactly ANS, some logical extensions)

```
query           ( -- ) Fetches user input to input buffer
tib             ( -- cstr-addr ) Input buffer

current-source  ( -- addr ) Double-Variable which contains source
setsource       ( c-addr len -- ) Change source
source          ( -- c-addr len ) Current source
>in             ( -- addr ) Variable with current offset into source

token           ( -- c-addr len ) Cuts one token out of input buffer
parse           ( char -- c-addr len )
                Cuts anything delimited by char out of input buffer

evaluate        ( any addr len -- any ) Interpret given string
interpret       ( any -- any ) Execute, compile, fold, optimize...
quit            ( many -- ) (R: many -- ) Resets Stacks
hook-quit       ( -- a-addr ) Hook for changing the inner quit loop
```

Dictionary expansion (exactly ANS, some logical extensions)

```
align           ( -- ) Aligns dictionary pointer
aligned         ( c-addr -- a-addr ) Advances to next aligned address
cell+           ( x -- x+4 ) Add size of one cell
cells           ( n -- 4*n ) Calculate size of n cells

allot           ( n -- ) Tries to advance Dictionary Pointer by n bytes
                         Aborts, if not enough space available
here            ( -- a-addr|c-addr )
                Gives current position in Dictionary

,               ( u|n -- ) Appends a single number to dictionary
><,             ( u|n -- ) Reverses high and low-halfword, then
                           appends it to dictionary
```

```
h,              ( u|n -- ) Appends a halfword to dictionary

compiletoram?   ( -- ? ) Currently compiling into ram ?
compiletoram    ( -- ) Makes ram   the target for compiling
compiletoflash  ( -- ) Makes flash the target for compiling
```

Dictionary expansion (speciality!)

```
string,         ( c-addr len -- ) Inserts a string of maximum 255 characters
                                  without runtime
literal,        ( u|n -- ) Compiles a literal with runtime
inline,         ( a-addr -- ) Inlines the choosen subroutine
call,           ( a-addr -- ) Compiles a call to a subroutine
jump,           ( Hole-for-Opcode Destination )
                Writes an unconditional Jump
                to a-addr-Destination with the given Bitmask as
                Opcode into the halfword sized a-addr-Hole
cjump,          ( Hole-for-Opcode Destination Bitmask )
                Writes a conditional Jump
                to a-addr-Destination with the given Bitmask as
                Opcode into the halfword sized a-addr-Hole
ret,            ( -- ) Compiles a ret opcode

flashvar-here   ( -- a-addr ) Gives current RAM management pointer
dictionarystart ( -- a-addr ) Current entry point for dictionary search
dictionarynext  ( a-addr -- a-addr flag )
                Scans dictionary chain and returns true if end is reached.
```

Available depending on chip capabilities:

```
c,              ( char -- ) Appends a byte to dictionary
halign          ( -- ) Makes Dictionary Pointer even, if uneven.

movwmovt,       ( x Register -- ) Generate a movw/movt-Sequence to
                                  get x into any given Register. M3/M4 only
registerliteral, ( x Register -- ) Generate shortest possible sequence
                                  to get x into given low Register.
                                  On M0: A movs-lsls-adds... sequence
                                  M3/M4: movs / movs-mvns / movw / movw-movt

12bitencoding   ( x -- x false | bitmask true )
                                  Can x be encoded as 12-bit immediate ?
```

Flags and inventory (speciality!)

```
smudge          ( -- ) Makes current definition visible, burns
                       collected flags to flash and
                       takes care of proper ending
inline          ( -- ) Makes current definition inlineable.
                       For flash, place it inside your definition !
immediate       ( -- ) Makes current definition immediate.
                       For flash, place it inside your definition !
compileonly     ( -- ) Makes current definition compileonly.
                       For flash, place it inside your definition !
setflags        ( char -- ) Sets Flags with a mask. This isn't immediate,
                       but for flash, place it inside your definition !
(create) name   ( -- ) Creates and links a new invisible dictionary
                       header that does nothing.
                       Use FIG-style <builds .. does> !
find            ( c-addr len -- a-addr flags )
                       Searches for a String in Dictionary.
```

```
                          Gives back flags, which are different to ANS !

        0-foldable      ( -- ) Current word becomes foldable with zero constants
        1-foldable      ( -- ) Current word becomes foldable with one constants
        2-foldable      ( -- ) Current word becomes foldable with two constants
        3-foldable      ( -- ) Current word becomes foldable with 3   constants
            ...
        7-foldable      ( -- ) Current word becomes foldable with 7   constants
```

Compiler essentials (subtle differences to ANS)

```
        execute         ( a-addr -- ) Calls subroutine
        recurse         ( -- ) Lets the current definition call itself
        ' name          ( -- a-addr ) Tries to find name in dictionary
                                      gives back executable address
        ['] name        ( -- a-addr)  Tick that compiles the executable address
                                      of found word as literal
        postpone name   ( -- ) Helps compiling immediate words.
        does>           ( -- ) executes: ( -- a-addr )
                               Gives address to where you have stored data.
        <builds         ( -- ) Makes Dictionary header and reserves space
                               for special call.
        create name     ( -- ) Create a definition with default action which
                               cannot be changed later. Use <builds does> instead.
                               Equivalent to : create <builds does> ;
        state           ( -- a-addr ) Address of state variable
        ]               ( -- ) Switch to compile state
        [               ( -- ) Switch to execute state
        ;               ( -- ) Finishes new definition
        : name          ( -- ) Opens new definition
```

Control structures (exactly ANS)

Internally, they have complicated compile-time stack effects.

Decisions:

```
flag if ... then
flag if ... else ... then

        then            ( -- )          This is the common
        else            ( -- )          flag if ... [else ...] then
        if              ( flag -- )     structure.
```

Case:

```
n case
     m1   of ... endof
     m2   .. ... .....
   flag  ?of ... endof
    all others
  endcase

        case            ( n -- n )      Begins case structure
        of              ( m -- )        Compares m with n, choose this if n=m
        ?of             ( flag -- )     Flag-of, for custom comparisions
        endof           ( -- )          End of one possibility
        endcase         ( n -- )        Ends case structure, discards n
```

Indefinite Loops:

```
begin ... again
begin ... flag until
begin ... flag while ... repeat

        repeat          ( -- ) Finish of a middle-flag-checking loop.

        while           ( flag -- ) Check a flag in the middle of a loop

        until           ( flag -- ) begin ... flag until
                                    loops as long flag is true
        again           ( -- )  begin ... again
                                is an endless loop
        begin           ( -- )
```

Definite Loops:

```
limit index   do ... [one or more leave(s)] ... loop
             ?do ... [one or more leave(s)] ... loop
              do ... [one or more leave(s)] ... n +loop
             ?do ... [one or more leave(s)] ... n +loop


        k               ( -- u|n ) Gives third  loop index
        j               ( -- u|n ) Gives second loop index
        i               ( -- u|n ) Gives innermost loop index


        unloop          (R: old-limit old-index -- )
                        Drops innermost loop structure,
                        pops back old loop structures to loop registers

        exit            ( -- ) Returns from current definition.
                               Compiles a ret opcode.

        leave           ( -- ) (R: old-limit old-index -- )
                        Leaves current innermost loop promptly

        +loop           ( u|n -- )
                        (R: unchanged | old-limit old-index -- )
                        Adds number to current loop index register
                        and checks whether to continue or not
        loop            ( -- )
                        (R: unchanged | old-limit old-index -- )
                        Increments current loop index register by one
                        and checks whether to continue or not.

        ?do             ( Limit Index -- )
                        (R: unchanged | -- old-limit old-index )
                        Begins a loop if limit and index are not equal

        do              ( Limit Index -- )
                        (R: -- old-limit old-index )
                        Begins a loop
```

Common Hardware access

```
        reset           ( -- ) Reset on hardware level
        dint            ( -- ) Disables Interrupts
        eint            ( -- ) Enables  Interrupts
        eint?           ( -- ) Are Interrupts enabled ?
```

```
nop             ( -- ) No Operation. Hook for unused handlers !

ipsr            ( -- ipsr ) Interrupt Program Status Register
unhandled       ( -- ) Message for unhandled interrupts.

irq-systick     ( -- a-addr ) Memory locations for IRQ-Hooks
irq-fault       ( -- a-addr ) For all faults
irq-collection  ( -- a-addr ) Collection of all unhandled interrupts
```

Specials for LM4F120:

Flash:

```
eraseflash      ( -- ) Erases everything. Clears Ram. Restarts Forth.

eraseflashfrom  ( a-addr -- ) Starts erasing at this address.
                              Clears Ram. Restarts Forth.

flashpageerase  ( a-addr -- ) Erase one 1k flash page only. Take care:
                              No Reset, no dictionary reinitialisation.

cflash!         ( char c-addr -- )  Writes byte to flash
hflash!         ( u|n a-addr -- )   Writes halfword to flash
flash!          ( u|n 4-a-addr -- ) Writes single number to flash,
                                      4 aligned !
```

Interrupts:

```
irq-porta       ( -- a-addr ) Memory locations for IRQ-Hooks
irq-portb
irq-portc
irq-portd
irq-porte
irq-portf
irq-timer0a
irq-timer0b
irq-timer1a
irq-timer1b
irq-timer2a
irq-timer2b
irq-adc0seq0
irq-adc0seq1
irq-adc0seq2
irq-adc0seq3
irq-terminal
```

Miura–Ori (ミウラ折り)

Nach so vielen Forth–Artikeln soll nun noch eine Seite folgen, die mal zur Erfrischung ganz ohne Tastatur, Leuchtdioden und Lötkolben auskommt. Ist es nicht eine gute Tradition, in der Adventszeit ein bisschen mit Papier zu Basteln? Außerdem trifft es doch bestimmt auf alle zu, die gern die Vierte Dimension lesen: Wer Forth mag, hat auch Freude an unkonventionellen Lösungen und praktischen Ideen, so ungewöhnlich sie auch sein mögen! Und so soll — nach Wunsch des Autors — einmal eine Origami–Anleitung in der Vierten Dimension erscheinen.

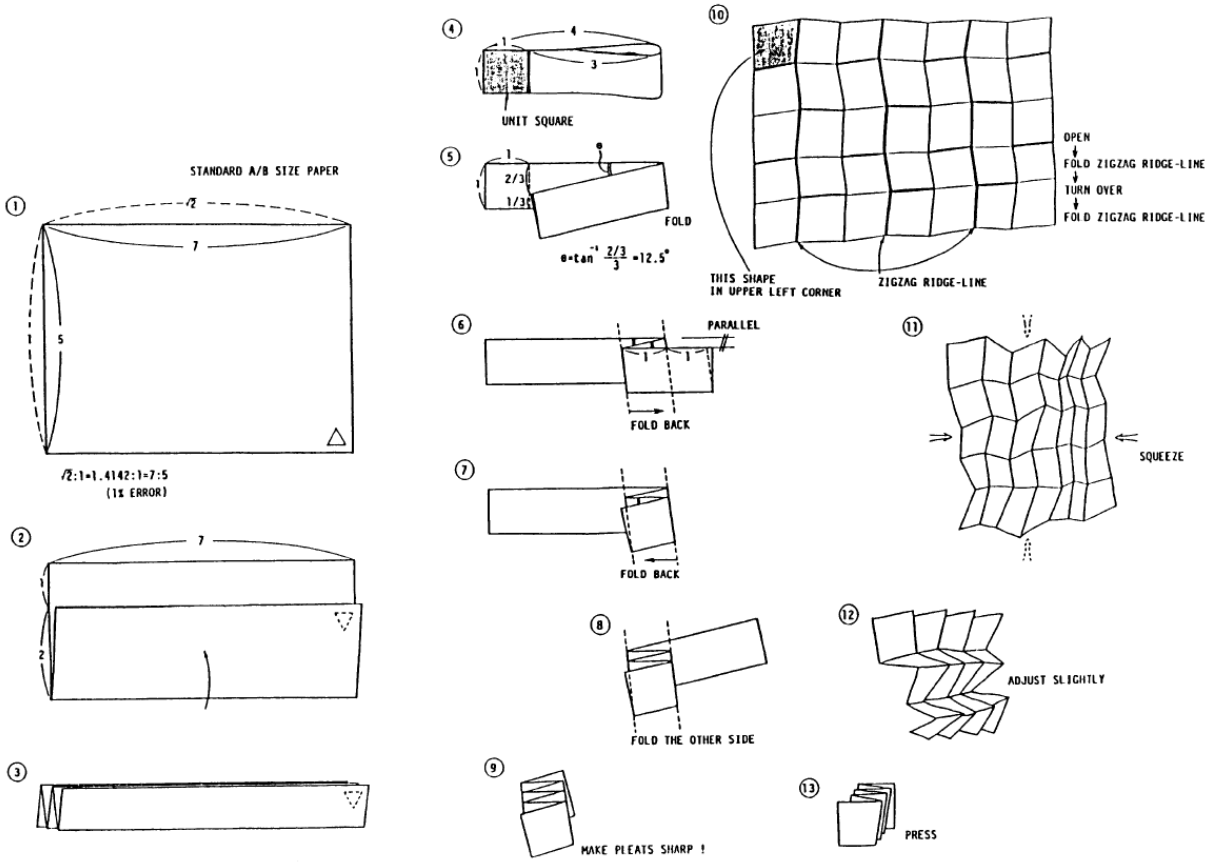


Abbildung 1: How to fold Miura–Ori, Natori's method (Illustration by M. Sakamaki).

Auch wenn die Zeichnung zunächst unscheinbar aussieht, so ist Miura–Ori sehr praktisch, um Notizblätter, Fahrpläne und vielleicht auch einen Liebesbrief auf eine Weise zusammenzufalten, dass sie sich unterwegs ganz schnell auseinanderfalten und wieder zusammenlegen lassen können! Es genügt, die oberste und unterste Ecke des gefalteten Blattes zu nehmen und zu ziehen — schon ist das Papier wieder im Ausgangsformat. Doch der Trick dabei ist, dass es auch umgekehrt funktioniert — das Zusammenfalten ist durch das Zusammenschieben der beiden diagonal gegenüberliegenden Ecken schnell getan. Wer es sich — wie ich — nicht sogleich vorstellen kann, dem möchte ich empfehlen, auf Wikipedia einmal ein Video davon anzusehen.

Ursprünglich wurde diese besondere Art der Faltung übrigens für das Verstauen eines Sonnensegels in einem japanischen Satelliten entwickelt… Aber für uns soll es eine Adventsbastelei bleiben, die auch im Frühling noch gute Dienste leisten wird. In diesem Sinne:

Alles Gute zum Advent und frohe Weihnachten!

Link

`https://en.wikipedia.org/wiki/Miura_fold`

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

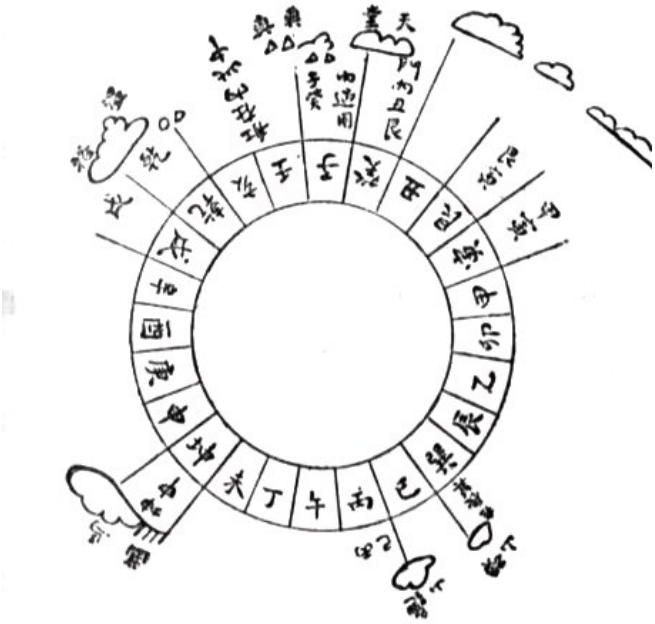
FORTHchips (FRP 1600, RTX, Novix)	**Klaus Schleisiek–Kern** Tel.: **(0 40) – 37 50 08 03** (g)
KI, Object Oriented Forth, Sicherheitskritische Systeme	**Ulrich Hoffmann** Tel.: **(0 43 51) – 71 22 17** (p) Fax: **– 71 22 16**
Forth–Vertrieb volksFORTH ultraFORTH RTX / FG / Super8 KK–FORTH	Ingenieurbüro **Klaus Kohl–Schöpe** Tel.: **(0 82 66)–36 09 862** (p)

Termine

Donnerstags ab 20:00 Uhr
Forth–Chat net2o `forth@bernd`

27.–30. Dezember 2015:
32C3 — 32. Chaos Communication Congress, Hamburg
`https://events.ccc.de`

12.–13. März 2016:
1. Maker Faire Ruhr
`http://www.makerfaire-ruhr.com`

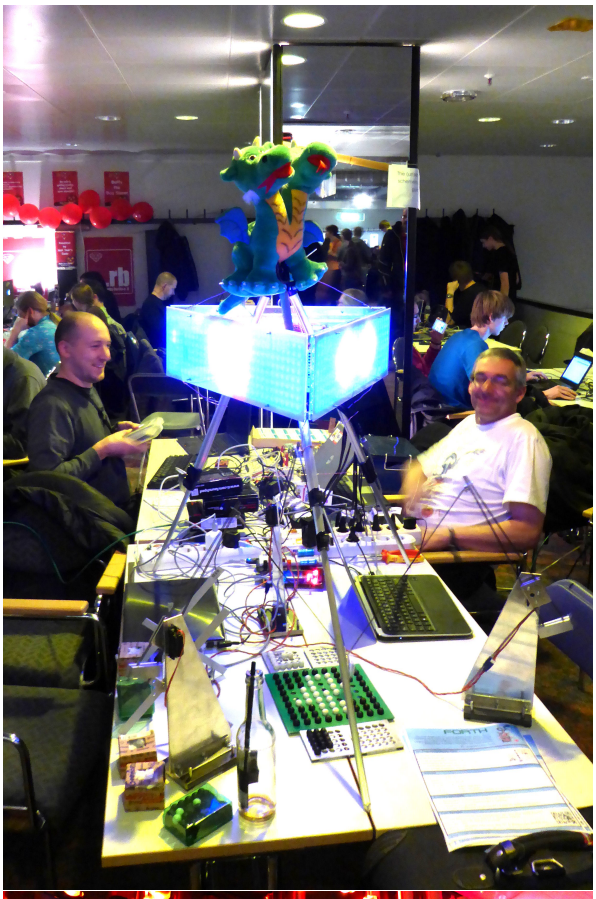


Möchten Sie gerne in Ihrer Umgebung eine lokale Forthgruppe gründen, oder einfach nur regelmäßige Treffen initiieren? Oder können Sie sich vorstellen, ratsuchenden Forthern zu Forth (oder anderen Themen) Hilfestellung zu leisten? Möchten Sie gerne Kontakte knüpfen, die über die VD und das jährliche Mitgliedertreffen 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.

32c3 vom 27. bis 30. Dezember in Hamburg

Die Forth–Gessellschaft trägt auch dieses Jahr zum Chaos Communication Congress in Hamburg bei. Für das Standpersonal durchaus anstrengend, denn Hacker kommen zwar frühestens mittags aus dem Bett, gehen aber nie vor 3 Uhr nachts schlafen — aber nur, wenn sie vorher eine komplette Einweisung in Forth bekommen haben.



Assembly

Auf unserer Assembly stellen wir die Bitkanone, den Triceps, b16, Gforth (insbesondere auf Android) und natürlich Mecrisp vor, diskutieren mit Besuchern und werben für Forth.

Vortrag "Compileroptimierungen für Forth im Microcontroller"

Matthias Koch hält einen Vortrag über seinen optimierenden Compiler.

Session #wefixthenet "net2o — make it userfriendly"

Bernd Paysan hält voraussichtlich einen net2o–Vortrag in einer selbst–organisierten Session; für genaue Details ist dieses Heft zu früh fertig geworden.

Mehr

Der CCC ist eine riesige Veranstaltung, die wahnsinnig viel Spaß macht: Von Tech–Talks über Politisches, humorvolles (Fnord–Jahresrückblick!) bis zur Kunst. Mehr auf der Congress–Homepage: `https://events.ccc.de/congress/2015/wiki/Main_Page`