Vierte Dimension

4/2018

Das Forth–Magazin

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

In dieser Ausgabe:

Cloning programmer for MSP430FR2433

Code Coverage messen mit Gforth

Clock Works 6 — Die UTC–Funkuhr

Günstiger Einstieg in die FPGA–Programmierung

Amforth wird groß

Forth–Tagung in Worms

34. Jahrgang — **Ausg. 4/2018** — 4,–€

Organ der Forth–Gesellschaft e.V.

Dienstleistungen und Produkte fördernder Mitglieder des Vereins

ServoNaut

Fahrtregler - Lichtanlagen - Soundmodule - Modellfunk

tematik GmbH
Technische
Informatik

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

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

LEGO RCX–Verleih

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

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

Inhalt:

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

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

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

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

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

Ingenieurbüro
Klaus Kohl–Schöpe

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

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

KIMA Echtzeitsysteme GmbH

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

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

FORTecH Software GmbH
Entwicklungsbüro Dr.–Ing. Egmont Woitzel

Tannenweg 22 m — D–18059 Rostock
`https://www.fortech.de/`

Wir entwickeln seit fast 20 Jahren kundenspezifische Software für industrielle Anwendungen. In dieser Zeit entstanden in Zusammenarbeit mit Kunden und Partnern Lösungen für verschiedenste Branchen, vor allem für die chemische Industrie, die Automobilindustrie und die Medizintechnik.

Cornu GmbH
Ingenieurdienstleistungen
Elektrotechnik

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

Leserbriefe und Meldungen 5

Cloning programmer for MSP430FR2433 7
Willem Ouwerkerk

Code Coverage messen mit Gforth 13
Bernd Paysan

Clock Works 6 — Die UTC–Funkuhr 17
Erich Wälde

Günstiger Einstieg in die FPGA–Programmierung 25
Klaus Kohl–Schöpe

Amforth wird groß 28
Matthias Trute

Forth–Tagung in Worms 32
Ewald und Andrea Rieger

Titelbild Cloning AUTOR mk

Quelle Teilausschnitt, www ...

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

Manuskripte und Rechte

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

Liebe Leser,

das vorliegende Heft wurde ja schon letztes Jahr aufgelegt, ist aber nun erst fertig — 2019. Doch in der Hoffnung, schon bald wieder ein Heft zu haben, bleibt es nun das vierte Heft 2018. Vorneweg gleich der Hinweis auf die letzte Seite des Heftes und unsere Website: `www.forth-ev.de` — denn dort könnt ihr euch anmelden zur Forth–Tagung 2019. Es geht in ein Weingut im Frühling. Ich bin gespannt, wie die Beteiligung sein wird. . . :-)

Den Auftakt der Beiträge macht diesmal Willem. Es war ihm ja schon gelungen, einen Cloner für das *noForth* der MSP430G2553–MCU zu schreiben. Also ein Stückchen Code, welches das komplette noForth, samt Cloner–Code, in eine weitere MCU dieser Sorte kopiert. Und zwar ohne einen PC dazwischen, das macht der Mikro selbst. Na gut, die Hardware muss schon da sein. . . Und wenn man erstmal bei so einem Thema ist, findet man so einiges. Viel Vergnügen bei der Lektüre.
`https://medium.economist.com/if-human-cloning-happened-db76888a2069`
Die Kunst des *Code Coverage* kann Programmierer in die Verzweiflung treiben. Wikipedia ist noch der Meinung, dass die „Testabdeckung“ mit Testfällen geschehen müsse und dass daher „die Bestimmung der Anzahl der möglichen Testfälle für reale Probleme oft nicht möglich“ sei. Bernd zeigt, wie es mit Forth anders und besser geht.
Das Zeit so ein Thema für sich ist, hat uns Erich ja schon einige Male näher gebracht. Und auch jetzt tickt die Uhr. . . Mir fällt dazu an dieser Stelle ein, euch allen ein gutes neues Jahr zu wünschen. Möge euer *Kairos* oft genug erholsam sein und euch dem *Chronos* entfliehen lassen.
Erfreut habe ich vernommen, dass der Traum, selbst einen Prozessor „backen“ zu können, greifbarer wird. Nicht nur theoretisch, sondern ganz praktisch, weil die Boards mit FPGAs nun erschwinglich geworden sind. Klaus hat viel damit zu tun und weiht uns in den Stand der Dinge ein.
Hat man sich an ein Forth gewöhnt, möchte man es überall haben. Das liegt in der Natur des Menschen als reviertreues Hordentier. Die Versionen erobern sich mehr und mehr MCUs — sofern die Autoren mitziehen. Die „Matthiasse“ tun es: Nach Mecrisp nun auch Amforth, es springt über auf die 32-Bitter.

Nachruf

Unser Verseinsmitglied und „unser Lektor und Mathematiker“ Prof. Dr. Fred Behringer ist am 18.12.2018 überraschend im Alter von 86 Jahren verstorben. Die Beisetzung fand am 28.12.2018 auf dem Pasinger Friedhof, München statt. Viele von uns kannten ihn gut von den Tagungen, die er früher nie ausgelassen hatte und zu denen er immer zusammen mit seiner Lebensgefährtin Elisabeth Rohrmayer, unserer Liesl, angereist war. Die Reisen zu den letzten Tagungen hingegen waren ihm aber doch zu beschwerlich geworden. Mitglied fast von Anfang an, hat er dem Verein immer die Treue gehalten, war im Drachenrat, im Vorstand und hat sich ganz besonders engagiert als unermüdlicher Korrektor unserer Zeitschrift. Er liebte den Gedankenaustausch bei den Treffen, immer getragen von seiner Wahrheitsliebe, aufrichtig und klar.

Fred, wir vermissen Dich.

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Webseite des Vereins herunterladen.
`http://fossil.forth-ev.de/vd-2018-04`

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

Ulrich Hoffmann — Kontakt: `Direktorium@Forth-ev.de`
Bernd Paysan
Carsten Strotmann

Mikrocontroller–Verleih ist umgezogen

Leser unserer Website bemerken immer mal wieder zu Recht, dass dortige Links auf den Mikrocontroller–Verleih ins Leere gehen. Das ist so, weil *mcv* ins Wiki umgezogen ist und der alte Link gelöscht wurde. Hier der neue:

`https://wiki.forth-ev.de/doku.php/mcv:mcv2`

Solltet ihr auf tote Links stoßen, meldet euch bitte. Wir werden dort die gültigen einsetzen. mk

noForth–Cloner

Während der Drucklegung hat Willem damit natürlich weiter gemacht und inzwischen auch die Version für den FR2355 fertig gestellt. So ein Printmedium ist zu träge um das nachzuhalten — aber wem erzähle ich das. Schaut euch einfach selbst um auf der noForth-Website oder fragt Willem direkt. mk

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

Die Forth–Gesellschaft auf der OpenRhein-Ruhr in Oberhausen, November 2018

„Ein Pott voll Software“ — Die Organisatoren der kleinen, aber feinen Messe rund um das Thema *Freie Software* hatten sich auch dieses Jahr wieder mächtig ins Zeug gelegt, um Besuchern und Ausstellern ein angenehmes Ambiente zu bieten. Die Veranstaltung fand in den Räumlichkeiten des Rheinischen Industriemuseums in Oberhausen statt. Knapp 30 Aussteller empfingen am ersten Novemberwochenende einige hundert Besucher, um Projekte vorzustellen und regen Erfahrungsaustausch zu betreiben. Des Weiteren wurden parallel Vorträge und Workshops angeboten. Für das leibliche Wohl war ebenfalls gesorgt.

Diese Gelegenheit konnte sich die lokale Gruppe Ruhrgebiet des Forth e.V. nicht entgehen lassen und betrieb mit 3 Mitgliedern einen Stand, um Interessenten Forth näherzubringen oder alte Erinnerungen wiederzubeleben.

Martin zeigte seinen Niedertemperatur–Stirlingmotor nebst Prüfstand und grafischer Datenauswertung auf dem PC. Carsten hatte zwei kleine Gadgets dabei. Den Ben–NanoNote, ein Kleinst–Linux–Rechner, auf dem 4thForth und auch Gforth läuft, sowie einen Wiki–Reader mit Forth in der Firmware. Ich stellte den NumWorks aus, einen Open–Source–Taschenrechner, auf dem Matthias Koch sein Mecrisp–Stellaris zum Laufen gebracht hat. Außerdem veranschaulichten wir, wie man nutzlos herumrostenden Arduinos mit Amforth neues Leben einhauchen kann. Bei durchweg fachkundigem Publikum konnte sich unser Team über mangelnde Resonanz nicht beklagen.

Rückblickend fragt man sich oft, ob der Aufwand noch lohnt, auf Messen auszustellen. Ich denke, ja! Es ist bei Werbung üblich, dass der Erfolg nicht direkt in Zahlen zu messen ist. Jemand sagte mal so passend: „Kundschaft ist träge.“ Da heißt es, am Ball zu bleiben. Wolfgang

`https://openrheinruhr.de/`
`https://en.wikipedia.org/wiki/Ben_NanoNote`
`https://en.wikipedia.org/wiki/WikiReader`
`https://www.numworks.com`
`http://amforth.sourceforge.net`
`http://mecrisp.sourceforge.net`

Abbildung 1: Die Zinkfabrik Altenberg in Oberhausen beherbergt heute das Rheinische Industriemuseum.

Embedded Forth im Linux–Kernel

Das Internet ist wie ein großer Dachboden: Beim Suchen nach verlegten Sachen findet man schnell interessante neue Forth–Artefakte. So auch geschehen in der Woche vor Weihnachten 2018, als ich auf der Suche nach in C geschriebenen Forth–Systemen für mein Unikernel–Projekt war (Artikel zu Unikernel und Forth kommt in der ersten VD im Jahr 2019).

Forth als Linux–Kernel–Modul

Auf Github fand ich ein Forth–System als Kernel–Modul.

`https://github.com/howerj/libforth/tree/linux-kernel-module`

Diese Idee hatte die Forth–Gesellschaft auf einem der Linuxtage, damals noch in Karlsruhe, vorgedacht. Richard James Howe hat diese Idee (unabhängig) umgesetzt. Das Kernel–Modul implementiert ein einfaches, in C geschriebenes Forth–System im Linux–Kernel. Das Modul, wenn geladen und aktiviert, erstellt einen Geräte–Pfad unter `/dev/forth`. Dieses Gerät kann zum Lesen und Schreiben geöffnet werden (z. B. per `screen /dev/forth`) und damit bekommt der Anwender eine Forth–Schnittstelle in den Linux–Kernel.

Mit dem Forth–System können die Datenstrukturen des Linux–Kernels erforscht werden. Es wäre sogar möglich, Gerätetreiber direkt in Forth zu schreiben. Aber Achtung: Wie so oft bei Forth gibt es keine Sicherheitsnetze. Der Anwender hat unbeschränkten Zugriff auf den Linux–Kernel, mit allen Vor- und Nachteilen. Ein unbedachter Schreibzugriff an die falsche Speicherstelle und der Kernel stürzt ab (im besten Fall) oder das Dateisystem auf der Festplatte wird korrumpiert und alle Daten sind verloren. Erste Schritte mit dem libforth–Kernel–Modul sollten daher zur Sicherheit in einer virtuellen PC–Maschine (z. B. VirtualBox oder KVM) geschehen, auf einer Installation ohne wichtige Daten auf den Datenträgern.

`https://howerj.github.io/libforth/readme.htm`

Embedded Forth–Emulator mit Meta–Compiler

Ein neues und derzeit aktiv entwickeltes Projekt von Richard J. Howe ist *embed*, ein in C geschriebener Emulator für eine einfache Forth–CPU.

`https://github.com/howerj/embed`

embed implementiert einen Emulator für idealisierte 16–Bit–Forth–CPUs (basierend auf der H2–CPU und J1–CPU) und auf dieser CPU läuft ein eForth–System. Neben dem Basis–Forth kommt embed mit einem `see`–Debugger, Block–IO und einem einfachen Block–Editor. Die Blöcke werden dabei nicht auf ein Speichermedium geschrieben, sondern leben im 64–KByte–Adressraum der CPU. Beim Start kann der Adressraum des Emulators aus einer Datei geladen und bei Beenden wieder in eine Datei geschrieben werden.

Das Forth–System hinter embed ist in der Form von *literate Programming* geschrieben. Die Dokumentation befindet sich direkt im Quellcode, kann aber auch auf der Webseite des Autors gelesen werden. Wie der Name schon andeutet, eignet sich embed, um ein kleines Forth–System in größere C–Programme als Skripting- oder Konfigurations–Sprache einzubauen. embed wird derzeit aktiv entwickelt, die API zum Emulator soll Plänen des Autors zufolge noch überarbeitet und vereinfacht werden. Cas

`https://howerj.github.io/embed/meta.htm`
`https://de.wikipedia.org/wiki/Literate_programming`

Film: All Creatures Welcome

Die Forth–Gesellschaft ist seit vielen Jahren auf dem *Chaos Communication Congress* vertreten und bringt dort Interessierten die Sprache und Philosophie der Programmiersprache *Forth* näher. Der Congress findet jedes Jahr zwischen Weihnachten und Neujahr statt, viele Jahre in Hamburg und nun zum zweiten Mal in *Leipzig*. Zusätzlich gibt es alle vier Jahre im Hochsommer eine Veranstaltung an der frischen Luft, das *Chaos Communication Camp*.

Filmemacherin Sandra Trostel hat einen Film über beide Veranstaltungen gedreht. Der Dokumentarfilm in Spielfilmlänge gibt einen Einblick in die Hackerkultur und die einzigartige Gemeinschaft auf diesen Treffen. Dank eines Crowdfunding konnte der Film zum 35C5–Congress Ende 2018 unter einer Creativ–Commons–Lizenz frei zum Download veröffentlicht werden. Im Film kommt auch ein prominentes Mitglied der Forth–Gesellschaft zu Wort — und es ist diesmal nicht, wie bei anderen Dokumentationen über den CCC, Klaus Schleisiek. Schaut euch den Film einmal an. Cas

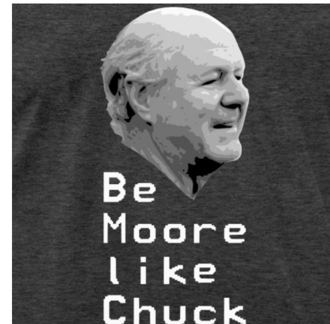
`http://sandratrostel.de/projects/allcreatureswelcome/`

QEMU–Adventskalender 2018 — Fun with Forth

Das Projekt rund um den Open–Source–System–Emulator *QEMU* bietet jeden Dezember einen elektronischen Adventskalender mit kleinen System–Abbildern, welche direkt in QEMU ausprobiert werden können. Am 4. Dezember gab es *Fun with Forth*, ein kleines Snake–Spiel in Forth, welches in Open–Firmware auf dem PPC64–Emulator läuft. Die Open–Firmware–Implementierung *OpenBIOS* ist inzwischen gut genug, um damit im QEMU–PPC MacOS 9 und MacOS X sowie Solaris 2.x im QEMU–Sparc–Emulator zu installieren. Cas

`https://www.qemu-advent-calendar.org/2018/`

To be or not to be ...



Ich bin neulich über diese T–Shirts und die Geschichte dahinter gestolpert. Viel Vergnügen bei der Lektüre. Cas

`http://sigusr2.net/be-moore-like-chuck.html`
`https://shop.spreadshirt.com/sigusr2/`

FORTH–MEETUP RUHR 2019

Monatlich trifft sich eine kleine Gruppe von Forth–Enthusiasten im Unperfekthaus in Essen. Oft gibt es einen Vortrag über eine neue Forth–Entwicklung oder generische Computer–Themen.

Weitere Kernthemen unserer Treffen: Neue Entwicklungen in der Forth-Programmierung, Erfahrungen mit Forth–Systemen auf Desktop und embedded Systemen, Erstellung von eigenen Forth–Systemen oder Portierung von Forth auf neue Architekturen, Benutzung von Forth auf ATMEL, Arduino, ARM, MIPS, RISC–V, Intel, PPC und anderen CPU–Architekturen, Beteiligung an Messen und Open–Source–Veranstaltungen (z. B. OpenRheinRuhr).

Wir treffen uns im Café (Erdgeschoss) des Unperfekthauses, als Erkennungszeichen haben wir den Swap–Drachen. Das Unperfekthaus kostet Eintritt, dafür sind die Getränke frei. Die aktuellen Termine und Themen unserer Treffen findet ihr auf unserer Meetup–Seite. Dort könnt ihr euch für die Treffen anmelden und auch Fragen stellen. Schaut mal vorbei! Cas

`https://www.forth-ev.de/staticpages/index.php/swap`
`https://www.unperfekthaus.de/restaurant/`
`https://www.meetup.com/de-DE/Essen-Forth-Meetup/`

Cloning programmer for MSP430FR2433

Willem Ouwerkerk

Most MSP430 MPU's contain one of many versions of a bootloader known as BSL. When there is a need to copy an MSP430 without much resources, then there is the possibility for using this built–in BSL programmer.

This cloning BSL programmer can be made with only a few instructions of the BSL and may be used stand–alone!

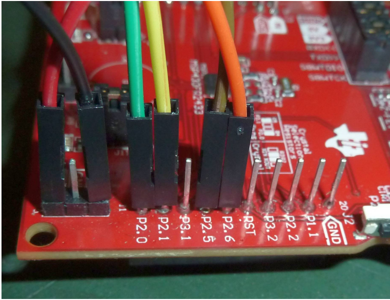


Figure 1: Wiring of cloner ...

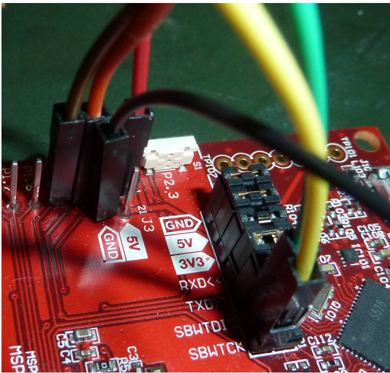


Figure 2: ... and target

This article describes two versions (just for fun). One that uses the absolute minimum of code and only 2 BSL commands. The cloning action is done within 14 seconds.

The second one uses 3 BSL commands and it does the cloning in less than 2 seconds now.

The code is separated into 4 parts: The RS232 communication using even parity, entering and leaving the BSL, communication with the BSL using the built–in CRC generator for checksums (this results in two or three BSL–commands), and finaly the cloner.

RS232 communication

- `BSL-CONFIG` — That needs baudrate data on the stack for USCI-A1.
- `BSL-KEY?` — UART data from BSL received.
- `BSL-KEY` — Read UART data from BSL.
- `BSL-EMIT` — Send data to BSL using the UART.
- `CRC-EMIT` — Same as previous, only the CRC-generator is updated too.

Entering and leaving the BSL

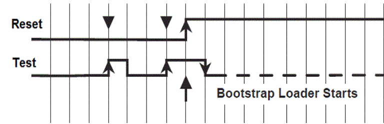


Figure 3: BSL entry sequence

- `START-BSL` — Generate the BSL entry sequence on RST & TCK.
- `RESET-TARGET` — Normal reset pulse to start target.
- `SETUP-BSL` — Initialise BSL I/O & RS232 to 9600 baud.

Communication with BSL

The command structure consists of a header 0x80 and a 2 byte core length field. The command itself (byte–code), a 3 byte address field (optional), a data record of one or more bytes (optional) and finally the checksum. The checksum is generated over the core only.[1]

Table 6. UART BSL Command

Header	Length	Length	BSL Core Command	CKL	CKH
0x80	NL	NH	See Section 4.1.5	CKL	CKH

Figure 4: Command packet

The response from the BSL starts with an ACK. This signals that the received packet is formatted ok. After that a response packet is sent. This contains the requested information or an error message. When this message is zero the command was executed without errors.

Table 7. UART BSL Response

Acknowledgment	Header	Length	Length	BSL Core Response[1]	CKL	CKH
ACK from BSL	0x80	NL	NH	See Section 4.1.4	CKL	CKH

[1] BSL Core Response is not always included.

Figure 5: Response packet

- `ANSWER?` — Wait a few millisec for a response.
- `'RESPONSE` — Buffer for BSL message packet.
- `RESPONSE` — Catch & store BSL response packets.
- `ERR?` — Check for ACK, then catch & check the BSL response packet, leave zero when all is ok.
- `PREPARE` — Check BSL command(s) to help format the correct command packets.

[1] 4.1.1 UART Peripheral Interface Wrapper; FRAM–bootloader — Texas Instruments slau550p.pdf

- `COMMAND?` — Send BSL commands with CRC-checksum and catch errors, leave zero when all is ok.
- `PASSWORD?` — Send password to BSL to get access, note that an invalid password does a mass erase!
- `WRITE-BLOCK` — Copy a block of FRAM to the target.
- `BAUDRATE` — Change the baudrate for the BSL.

The cloner

- `CLONE-FRAM` — Copy all relevant noForth FRAM to the target. These are INFO FRAM, used main FRAM and the vector table.
- `CLONE` - Initialise BSL, enter the BSL, set empty password (all 0xFF). Send password. If this results in an error the device was not empty and a mass erase is done. The password is then sent a second time. This trick always opens the BSL! Now all FRAM is copied and a reset is applied.

The target contains the exact copy of the cloner. So it is ready to clone again and again . . . When the shield is removed from the source code, the cloned copy just resets as the used noForth version, with added tools!

Link

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

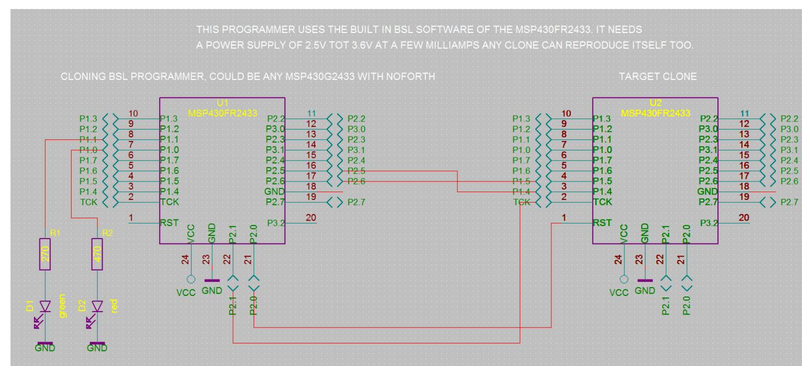


Figure 6: Schematics of the cloner

Listings

```
1   (* Smallest readable cloner for MSP430FR2433, size 718 bytes
2
3   The MSP430FR2433 port registers are memory mapped. An overview:
4   Label     P1  P2  P3  Function
5   ------------------------------------------------
6   PxIN      200 201 220 Input
7   PxOUT     202 203 222 Output
8   PxDIR     204 205 224 Direction
9   PxREN     206 207 226 Resistor enable
10  PxSEL0    20A 20B 22A Select 0
11  PxSEL1    20C 20D 22C Select 1
12  P1IV      20E 21E 22E Interrupt vector word
13  P1SELC    210 211 230 Complement selection
14  PxIES     218 219 238 Interrupt edge select
15  PxIE      21A 21B 23A Interrupt on
16  PxIFG     21C 21D 23C Interrupt flag
17
18  eUSCI_A0 : 0500h = base address
19  eUSCI_A1 : 0520h = base address
20  CRC16    : 01C0h = base address
21
22  Target BSL needs:
23  - BSL entry sequence at reset and TCK pin, followed by 50 ms delay
24  - UART on P1.4 & P1.5 at 9600 baud [8bit data, even parity bit, 1 stop bit]
25  - CRC: All bytes in the core block, beginning with cmd-byte to last data-byte
26  - Write bytes to CRCDIRB - not CRCDI - but read CRCINIRES.
27  - Wait minimal 1.2 ms between commands and check ACK-byte=00 before next command.
28
29  Cloner ------------ Clone
30  P2.0 = Reset BSL -> Reset
31  P2.1 = TCK BSL   -> TCK
32  P2.5 = Rx BSL    -> P1.4 Tx
33  P2.6 = Tx BSL    -> P1.5 Rx
34  3V3  = Vcc       -> 3V3
35  GND  = Ground    -> GND
```

```
36
37   *)
38
39   chere  hex     \ RS232 routines using even parity
40   : BSL-KEY       ( -- b )        begin  1 53C bit* until  52C c@ ;
41   : BSL-EMIT      ( b -- )        begin  2 53C bit* until  52E c! ;
42   : CRC-EMIT      ( b -- )        dup 01C2 c!  bsl-emit ; \ CRCDIRB
43
44   \ 21.4.1 UCA1CTLW0 Register: bit14=1 even, bit15=1 parity.
45   \ Add 0x01 : UCSWRST=1 in &UCA1CTLW0 To freeze UART
46   \ Add 0x80 : UCSSELx = Use SMCLK
47   \ Add 0xC000 : parity 11=even, 01=odd, 00=no
48   : BSL-CONFIG    ( baud mod -- ) \ config uart_a1
49     C081 520 !        \ UCA1CTLW0  Even parity & block uart
50     ( 4911 ) 528 !    \ UCA1MCTLW  Modulation control
51     ( 34 ) 526 !      \ UCA1BRW    9600 baud default
52     60 20B *bis       \ P2SEL0 bit5+6  P2.5 & P2.6 are RS232
53     01 520 *bic ;     \ UCA1CTLW0  clear UCSWRST bit to restart UART
54
55
56   \ Cloning BSL programmer for MSP430FR2433
57
58   : START-BSL     ( -- )
59       1 203 *bic              \ Reset low
60       2 203 *bis  2 203 *bic  \ TCK pulse 1
61       2 203 *bis  1 203 *bis  \ TCK pulse 2 & BSL reset
62       2 203 *bic  30 ms ;     \ Release TCK & wait for BSL
63
64   : RESET-TARGET  ( -- )      \ Give target a reset pulse & set TCK low
65       3 203 *bic  1 203 *bis ;
66
67   : SETUP-BSL     ( -- )
68       03 205 *bis             \ P2.0 is Reset & P2.1 is TCK
69       reset-target            \ Target MPU runs
70       34 4911 bsl-config ;    \ Baudrate at 9600 baud
71
72
73   (* ACK and error packages with the FRAM BSL
74
75   Errors: 00 = Ack
76           51 = Header incorrect
77           52 = Checksum incorrect
78           53 = Packet size zero
79           54 = Packet size to large
80           55 = Something is wrong
81           56 = Unknown baud rate change
82   *)
83
84   : ANSWER?       ( -- f )    \ Wait about 4 ms for BSL response
85       120 0 do                \ Note this is a software timed loop!
86           1 53C bit* ?dup if  unloop exit  then \ BSL key received?
87       loop  false ;
88
89   create 'RESPONSE  08 allot  \ Save BSL response packets
90   : RESPONSE      ( -- )
91       3B 'response 3 + !    \ Init. response...
92       0C 0 do  answer? if  leave  then  loop  \ Catch response
93       08 0 do             \ Receive max. 8 bytes
94           answer? if      \ Wait until answer present?
95               bsl-key     \ Yes, read answer,
96               'response i + c! \ and store it
97           then
98       loop ;
99
100  \ Response is zero when a command was a success
101  : ERR?          ( -- 0|b )
102      answer? if  bsl-key 0<>  else  true  then \ Read Ack or nothing?
103      response [ 'response 3 + ] literal  \ Catch response if any and when
104      count 3B =  swap c@ and  or ;       \ BSL message is zero ( = no error )
105
106
107  \ Prepare data to help building a correct command record
108  value ADDR?     \ Build address field for this BSL-command Yes/No
109  : PREPARE       ( a u1 cmd -- a u1 u2 )
110     11 <>  to addr?      \ Command with or without address field
111     dup 1+               \ Command and to core length makes u2
```

```
112      addr? 3 and  +  ;    \ Add address field optional to u2
113
114   \ Leave word x into byte swapped & original version
115   : SHAKE     ( x -- 'x x )   dup ><  swap ;
116
117   \ BSL command with data block
118   : COMMAND?      ( a u command -- flag )
119       2 ms  80 bsl-emit  >r        \ Sync & send header
120       r@ prepare  shake            \ Prepare length of data
121       bsl-emit  bsl-emit           \ Send core length
122       FFFF 1C4 !  r> crc-emit      \ CRCINIRES  Restart CRC, send command
123       addr? if                     \ Send address?
124           over shake               \ Yes, now prepare start address
125           crc-emit  crc-emit       \ Send start address
126           0 crc-emit               \ Page address fixed to zero (for now)
127       then
128       bounds ?do                   \ Send data block ( a u1 )
129           i c@ crc-emit
130       loop
131       1C4 @ shake bsl-emit bsl-emit \ CRCINIRES  Finally send checksum
132       err? ;                       \ Ack or/and error message received?
133
134   : PASSWORD?     ( -- f )    fhere 20   11 command? ;
135   : WRITE-BLOCK   ( a +n -- ) ch * emit  10 command? ?abort ;
136
137   : CLONE-FRAM    ( -- )
138       frozen 200 bounds                \ FR2433 uses all info space too
139       do   i 100 write-block  100 +loop  \ noForth on FR2433 uses all info space
140       chere origin ?do  i 100 write-block  100 +loop \ Copy used main FRAM
141       ivecs dup abs write-block ;     \ Copy FR2433 vector table
142
143   \ Write used part of FRAM, the vector table and the whole info FRAM
144   \ Only two instructions are needed. When a MPU has code in it, an empty
145   \ password triggers a mass erase, after that the password is known.
146   \ It's all FF so just type CLONE and the cloning is done.
147   \ Does the cloning action within 15 seconds!
148   : CLONE         ( -- )
149       setup-bsl   start-bsl       \ Start BSL on target at 9600 baud,
150       fhere 20 FF fill            \ Empty password, all FF at FHERE
151       password? if  password? ?abort  then \ Send password, the first may trigger a mass erase
152       clone-fram  reset-target ;  \ Clone yourself and start clone MPU
153
154   \ ' clone  to app     \ Make cloner turnkey, each reset then does a cloning attempt
155   shield CLONER\  freeze
156   chere swap - dm u.
157
158   \ End
```

```
 1   (* Fastest cloner for MSP430FR2433, size 890 bytes
 2
 3   The MSP430FR2433 port registers are memory mapped. An overview:
 4   Label    P1  P2  P3  Function
 5   ------------------------------------------------
 6   PxIN     200 201 220 Input
 7   PxOUT    202 203 222 Output
 8   PxDIR    204 205 224 Direction
 9   PxREN    206 207 226 Resistor enable
10   PxSEL0   20A 20B 22A Select 0
11   PxSEL1   20C 20D 22C Select 1
12   P1IV     20E 21E 22E Interrupt vector word
13   P1SELC   210 211 230 Complement selection
14   PxIES    218 219 238 Interrupt edge select
15   PxIE     21A 21B 23A Interrupt on
16   PxIFG    21C 21D 23C Interrupt flag
17
18   eUSCI_A0 : 0500h = base address
19   eUSCI_A1 : 0520h = base address
20   CRC16    : 01C0h = base address
21
22   Target BSL needs:
23   - BSL entry sequence at reset and TCK pin, followed by 50 ms delay
24   - UART on P1.4 & P1.5 at 9600 baud [8bit data, even parity bit, 1 stop bit]
25   - CRC: All bytes in the core block, beginning with cmd-byte to last data-byte
26   - Write bytes to CRCDIRB - not CRCDI - but read CRCINIRES.
```

```
27   - Wait minimal 1.2 ms between commands and check ACK-byte=00 before next command.
28
29   Cloner ------------ Clone
30   P2.0 = Reset BSL -> Reset
31   P2.1 = TCK BSL   -> TCK/Test
32   P2.5 = Rx BSL    -> P1.4 Tx
33   P2.6 = Tx BSL    -> P1.5 Rx
34   3V3  = Vcc       -> 3V3
35   GND  = Ground    -> GND
36
37   *)
38
39   chere  hex     \ RS232 routines using even parity
40   code BSL-KEY?   ( -- 0|b )
41   \   tos sp -) mov  53C & tos .b mov  #1 tos .b bia  next
42       8324 ,  4784 ,  0 ,  4257 ,  53C ,  F357 ,  4F00 ,
43   end-code
44   : BSL-KEY       ( -- b )        begin  bsl-key? until  52C c@ ;
45   : BSL-EMIT      ( b -- )        begin  2 53C bit* until  52E c! ;
46   : CRC-EMIT      ( b -- )        dup 01C2 c!  bsl-emit ; \ CRCDIRB
47
48   \ 21.4.1 UCA1CTLW0 Register: bit14=1 even, bit15=1 parity.
49   \ Add 0x01 : UCSWRST=1 in &UCA1CTLW0 To freeze UART
50   \ Add 0x80 : UCSSELx = Use SMCLK
51   \ Add 0xC000 : parity 11=even, 01=odd, 00=no
52   : BSL-CONFIG    ( baud mod -- ) \ config uart_a1
53     C081 520 !        \ UCA1CTLW0  Even parity, block uart
54     ( 4911 ) 528 !    \ UCA1MCTLW  Modulation control
55     ( 34 ) 526 !      \ UCA1BRW    9600 baud default
56     60 20B *bis       \ P2SEL0 bit5+6  P2.5 & P2.6 are RS232
57     01 520 *bic ;     \ UCA1CTLW0  clear UCSWRST bit to restart UART
58
59
60   \ Cloning BSL programmer for MSP430FR2433
61
62   : START-BSL     ( -- )
63       1 203 *bic             \ Reset low
64       2 203 *bis  2 203 *bic  \ TCK pulse 1
65       2 203 *bis  1 203 *bis  \ TCK pulse 2 & BSL reset
66       2 203 *bic  30 ms ;     \ Release TCK & wait for BSL
67
68   : RESET-TARGET  ( -- )      \ Give target a reset pulse & set TCK low
69       3 203 *bic  1 203 *bis ;
70
71   : SETUP-BSL     ( -- )
72       3 204 *bis      \ Leds are outputs
73       3 202 *bic      \ Leds off
74       3 205 *bis      \ P2.0 is Reset & P2.1 is TCK
75       reset-target    \ Target MPU runs
76       34 4911 bsl-config ; \ Baudrate at 9600 baud
77
78
79   (* ACK and error packages with the FRAM BSL
80
81   Errors: 00 = Ack
82           51 = Header incorrect
83           52 = Checksum incorrect
84           53 = Packet size zero
85           54 = Packet size to large
86           55 = Something is wrong
87           56 = Unknown baud rate change
88   *)
89
90   : ANSWER?       ( -- f )    \ Wait about 4 ms for BSL response
91       180 0 do                \ Note this is a software timed loop!
92           bsl-key? ?dup if  unloop exit  then \ BSL key received?
93       loop  false ;
94
95   create 'RESPONSE  08 allot  \ Save BSL response packets
96   : RESPONSE      ( -- )
97
98   : RESPONSE      ( -- )
99       3B 'response 3 + !    \ Init. response...
100      0C 0 do  answer? if  leave  then  loop  \ Catch response
101      08 0 do          \ Receive max. 8 bytes
102          answer? if   \ Key present?
```

```
103             bsl-key  \ Yes, to next location & read data,
104             'response i + c! \ store it & incr count
105         then
106     loop ;
107
108 \ Response is zero & the green led on, when a command was a success
109 : ERR?          ( -- 0|b )
110     answer? if  bsl-key 0<>  else  true  then   \ Read Ack or nothing?
111     3 202 *bic                  \ Leds off
112     response [ 'response 3 + ] literal \ Catch response if any and when
113     count 3B =  swap c@ and  or \ BSL message is zero ( = no error )
114     dup if  1  else  2  then    \ Red is on when tos = true, otherwise green
115     202 *bis ;                  \ Activate led
116
117
118 \ Prepare data to help building a correct command record
119 value ADDR?     \ Build address field for this BSL-command Yes/No
120 : PREPARE       ( a u1 cmd -- a u1 u2 )
121     dup 11 =  swap 52 = or 0= to addr? \ Command with or without address field
122     dup 1+              \ Command and to core length makes u2
123     addr? 3 and  + ;    \ Add address field optional to u2
124
125 \ Leave word into byte swapped & original version
126 : SHAKE     ( x -- 'x x )   dup ><  swap ;
127
128 \ BSL command with data block
129 : COMMAND?      ( a u command -- flag )
130     2 ms  80 bsl-emit  >r   \ Sync & send header
131     r@ prepare  shake       \ Prepare length of data
132     bsl-emit  bsl-emit      \ Send core length
133     -1 01C4 !  r> crc-emit  \ CRCINIRES Restart CRC, send command
134     addr? if                \ Send address?
135         over shake          \ Yes, now prepare start address
136         crc-emit  crc-emit  \ Send start address
137         0 crc-emit          \ Page address fixed to zero (for now)
138     then
139     bounds ?do              \ Send data block ( a u1 )
140         i c@ crc-emit
141     loop
142     01C4 @ shake bsl-emit bsl-emit \ CRCINIRES Finally checksum
143     err? ;                  \ Ack or/and error message received?
144
145 create PASS 20 allot        \ Hold BSL password
146 : EMPTY-PASS    ( -- )          pass 20 FF fill ; \ PW for empty MPU
147
148 : PASSWORD?     ( -- f )    pass 20  11 command? ;
149 : BAUDRATE      ( +n -- )   pass c!  pass 1  52 command? ?abort ; \ 3 = 19K2
150 : WRITE-BLOCK   ( a +n -- ) ch * emit  10 command? ?abort ;
151
152 : CLONE-FRAM    ( -- )
153     frozen 200 bounds               \ FR2433 uses all info space too
154     do  i 100 write-block  100 +loop  \ Copy noForth FRAM info block
155     chere origin ?do  i 100 write-block  100 +loop \ Copy main FRAM
156     ivecs dup abs write-block ;     \ Copy FR2433 vector table
157
158 \ Write used part of FRAM, the vector table and the whole info FRAM
159 \ Only two instructions are needed. When a MPU has code in it, an empty
160 \ password triggers a mass erase, after that the password is known.
161 \ It's all FF so just type CLONE and the work is done.
162 \ Does the cloning action within 3 seconds!
163 : CLONE         ( -- )
164     setup-bsl  start-bsl  6 baudrate    \ Start at 9600B, then set BSL at 115K2
165     4 5551 bsl-config  empty-pass       \ Adapt cloner to 115k2, password = FF
166     password? if password? ?abort then  \ Send password, the first may trigger a mass erase
167     clone-fram  reset-target ;          \ Clone yourself and start clone MPU
168
169 : .CLONE      cr ." Fast cloner vsn 1.0 " ;
170
171 \ ' clone   to app  \ Make cloner turnkey, each reset then does a cloning attempt
172   ' .clone  to app
173
174 shield CLONER\  freeze
175 chere swap - dm u.
176
177 \ End
```

Code Coverage messen mit Gforth

Bernd Paysan

Wer Software testet, möchte auch wissen, ob seine Tests alle Code–Pfade durchlaufen. Dazu benutzt man Code–Coverage–Tools (eine Form des Profilings). Gforth hatte da bislang noch nichts, auch wenn Anton Ertl *vor 14 Jahren mal angefangen hat, eines zu schreiben; aufgrund fehlender Hilfsmittel aber wieder aufgegeben hat. Jetzt sind alle nötigen Bestandteile da, also ging die Implementierung schnell (in der aktuellen Entwicklungs–Version Gforth 0.7.9).*

Was will man erreichen?

Code, der nicht getestet wurde, läuft erfahrungsgemäß auch nicht. Code, der getestet wurde, läuft für die Testdaten auf den Pfaden, die beim Test durchlaufen wurden — der Rest ist ungetestet, läuft also nicht. Hat man wirklich alles getestet? Man müsste zumindest nachgucken können, welcher Teil des Codes denn durchlaufen wurde. Das Nachgucken kann der Computer für uns erledigen, er muss einfach kleine Schnipsel in den Code einstreuen, die ihm sagen: „Ich war hier.“ Also irgendwie das Äquivalent zu dem da:

```
         \|||/
         (o o)
--------ooO-(_)-Ooo--------
```

Diese Informationen will man auf zwei Arten auswerten: Zum einen als Prozentzahl der Coverage (soll 100 % erreichen), zum anderen als Annotation des Quelltextes, damit man gucken kann, welche Teile des Codes nicht erreicht wurden. Und wie oft. Grün für erreicht, Rot für nicht erreicht.

Was will man nicht erreichen?

Code–Coverage ersetzt kein Nachdenken über komplettes funktionales Testen. Der Code, der erreicht wurde, muss auch tatsächlich tun. Tests sind keineswegs komplett, wenn die Code–Coverage 100 % ist. Ist die Code–Coverage unter 100 %, ist der Test aber sicher unvollständig oder es gibt eben Code, der nie erreicht werden kann. Manchmal ist das sinnvoll, etwa, wenn man Fehlerabfragen einbaut und der Fehlerfall nicht eintreten können soll. Solche Teile des Codes könnte man mit einer umgekehrten Code–Coverage versehen: Grün für nicht erreicht, Rot für erreicht.

Coverage messen

Das Kern–Element zur Messung ist einfach das Inkrementieren eines Zählers, also `1 addr +!`. Um Platz zu sparen (das hier sind zwei Literals und ein Primitive, also fünf Zellen gefädelter Code) und um etwas mehr Tempo zu bekommen, wird das als Primitive `inc#` implementiert (holt sich die Adresse aus dem nächsten Wort im Code). Damit sind es nur noch zwei Zellen und schneller geht es auch noch, weil das Primitive den Stack überhaupt nicht anfassen muss.

Neben dem Zähler müssen wir auch noch abspeichern, wo genau die Stelle im Quelltext ist. Dafür gibt es in Gforth 0.7.9 die `current-sourceview`, die die Position im Quelltext als eine Zelle codiert. Da gibt es auch Tools für `locate`, mit denen man von der Position direkt in den Quelltext kommt und den anzeigen kann.

Wo muss der Messzähler–Code hin?

Man muss nicht überall messen. Code, der linear durchlaufen wird, wird allenfalls durch Exceptions unterbrochen. Diese sollte man beim Testen genau beobachten. Zwingend erforderlich sind Messzähler am Anfang eines Wortes sowie hinter jeder Kontrollflussänderung. Der Einfachheit halber hat man gern zusätzlich noch am Anfang jeder Zeile einen Messzähler; das hilft dann auch, Exceptions einzugrenzen. Ein `throw` z. B. packt man dann an das Ende der Zeile, wird die nächste erreicht, hat alles geklappt.

Gforth hat dafür fast alle Hooks schon bereit gestellt, da sie für Locals gebraucht werden. Nur nach dem `IF` fehlte ein Hook. Manchmal ist das Ergebnis etwas unintuitiv. So werden am Ende eines `?DO ... LEAVE ... LOOP` drei Zähler compiliert. Das hat aber seine Berechtigung: Der erste wird nur erreicht, wenn das `LOOP` das Schleifenende erreicht hat. Der zweite, wenn man über `LEAVE` aus der Schleife ausgestiegen ist. Und der dritte, wenn `?DO` über die Schleife gesprungen ist. Beim Regexp–Compiler von Gforth, der ja Backtracking macht, kann es am Ende der Regexp sehr viele solche Stellen geben, die aus unterschiedlichen Punkten innerhalb der Regexp erreicht werden; ebenso bei Case–Statements.

Wohin mit dem Zähler selbst?

Der Zähler soll ja nicht zwischendrin im Code herumhängen, da stört er nur. Anton hat mit seinen Sections aber auch hier schon Vorarbeit geleistet: Alle diese Zähler kommen zusammen mit den Sourceviews in eine eigene Section.

Coverage anzeigen

In Prozent

Das ist die einfache Variante: Hier muss man nur alle Zähler angucken und je nachdem, ob der Zähler 0 ist oder nicht, eine Variable hochzählen. Die mal 1000 geteilt durch die Anzahl der Zähler insgesamt ergibt dann die Prozent mit einer Nachkommastelle.

Als farbig annotierter Quelltext

Hier muss man jede geladene Datei durchgehen: Ist sie überhaupt im Coverage–Modus compiliert? Wenn nein, braucht man sie nicht auszugeben. Falls doch, geht man sowohl sequentiell durch die Zähler, als auch zeilenweise durch die Datei und setzt je nach Zählerstand die Farbe auf rot oder grün bzw. neutral, wenn es keine Coverage–Information für diese Zeile gibt. Die Zählerstände selbst werden invertiert ausgegeben.

Diese so annotierten Quelltexte kann man mit `annotate-cov` auch als Datei abspeichern und am Terminal angucken (farbig, versteht sich).

Im Editor ist Farbe nicht so einfach. Hier gibt es neben dem `color-cover` auch einen Modus `bw-cover`, in dem die Zählerstände in Klammern (also als Forth–Kommentare) ausgegeben werden. Diese so annotierten Dateien kann man als Quelltext verwenden; bei der nächsten Annotation werden die alten Zählerstände entfernt. Auch im Editor kann man am Ende der Arbeit die Zählerstände schnell löschen mit globalem Suchen und Ersetzen nach der Regexp `␣(␣[0-9]*)␣`.

Coverage speichern und laden

Wenn man verschiedene Tests laufen lässt, die jeweils einen Neustart des Programms erfordern, ist es sinnvoll, die Zählerstände mit `save-cov` abzuspeichern und später mit `load-cov` wieder zu laden. Dabei sollten sich verschiedene Programme nicht in die Quere kommen, und die Coverage verworfen werden, wenn sich an den Quellen etwas geändert hat. Dazu berechne ich den Hash über alle Sourceviews und speichere die Datei unter diesem Namen (als Hex–Ziffern) ab. Beim Laden wird erst der Hash berechnet, und dann, wenn vorhanden, die passende Datei geladen. Dabei ersetzt sie die vorhandenen Zählerstände; d. h. ausgeführte Programmteile, die während des Ladens bereits ausgeführt werden, werden nur einmal gezählt. Das sollte immer das Gleiche sein.

Nutzung

Wie nutzt man das jetzt? Ich habe mein Datums–Programm aus dem Forth–eV–Wiki mit ein paar Tests versehen. Dazu lädt man zunächst `test/ttester.fs` und `coverage.fs` und dann das eigentliche Programm. Vier Tests reichen aus, um alle Teile des Codes abzudecken; ausgeführt wird dieser Teil, wenn die Test–Tools auch zur Verfügung stehen:

```
[defined] t{ [defined] cov% and [IF]
     t{ 0 3 1 ymd2day dup day2dow
        -> 0 1 }t cov% cr
     t{ 1582 10 15 ymd2day 1- day2ymd
        -> 1582 10 4 }t cov% cr
     t{ 1400 3 1 ymd2day 1- day2ymd
        -> 1400 2 29 }t cov% cr
     t{ 2018 1 1 ymd2day 1- day2ymd
        -> 2017 12 31 }t cov% coverage
[THEN]
```

Danach ist alles grün, und die Coverage auf 100 %. Die Ausgabe der einzelnen `cov%` sieht so aus:

```
34.7% coverage
82.6% coverage
86.9% coverage
100.0% coverage
```

Das heißt noch lange nicht, dass der Code funktioniert. Ja, der Nullpunkt dieser Datumsrechnung ist am 1. März im Jahre 0 (Astronomen zählen so; auch wenn die Römer 153 v.Chr. den Monatsanfang auf den 1. Januar verlegt haben, ist nur mit dem 1. März als Nullpunkt des Jahres die Zahl der Monate einfach zu berechnen), das war ein Montag. Es gibt den Sprung vom julianischen auf den gregorianischen Kalender. 1400 war ein Schaltjahr und geht man vom 1.1. einen Tag zurück, landet man am 31.12. des Vorjahres.

Was hier überhaupt nicht getestet wird, ist, ob es wirklich nur 7 Wochentage gibt, ob die Anzahl der Tage pro Monat stimmen, ob die richtigen Jahre Schaltjahre sind oder nicht: Das sind alles funktionale Tests, sie ändern am Kontrollfluss nichts. Der ganze Quelltext in diesem Beispiel hat gerade mal fünf `IF`s.

Listings

`coverage.fs`

```
1   \ Code coverage tool
2
3   \ Copyright (C) 2018 Free Software Foundation, Inc.
4
5   \ This file is part of Gforth.
6
7   \ Gforth is free software; you can redistribute it
8   \ and/or modify it under the terms of the
9   \ GNU General Public License
10  \ as published by the Free Software Foundation,
11  \ either version 3 of the License,
12  \ or (at your option) any later version.
13
14  \ This program is distributed in the hope that it will
15  \ be useful, but WITHOUT ANY WARRANTY;
16  \ without even the implied warranty of
17  \ MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
18  \ See the GNU General Public License for more details.
19
20  \ You should have received a copy of the
21  \ GNU General Public License along with this program.
22  \ If not, see http://www.gnu.org/licenses/.
23
24  require sections.fs
25
26  unused extra-section coverage
27
28  ' Create coverage cover-start
29
30  : cover-end ( -- addr ) ['] here coverage ;
31  : cover, ( n -- ) ['] , coverage ;
32  : cover-end! ( addr -- )  [: dp ! ;] coverage ;
33
34  [IFUNDEF] coverage?
35      0 Value coverage? ( -- flag ) \ gforth-exp
36      \G Coverage check on/off
```

```
37  [THEN]
38  0 value dead-cov?
39
40  : cov+, ( -- )
41      coverage?  dead-code @ 0= and  loadfilename# @
42      0>= and  IF
43          current-sourceview input-lexeme @ + cover,
44          postpone inc# cover-end , 0 cover,
45      THEN
46      false to dead-cov? ;
47
48  : cov+ ( -- ) \ gforth-exp
49      \G add a coverage tag here
50      dead-cov? 0= state @ and  IF  cov+,  THEN
51      false to dead-cov? ; immediate compile-only
52  : ?cov+ ( flag -- flag ) \ gforth-exp
53      ]] dup IF ELSE THEN [[ ; immediate compile-only
54
55  :noname defers :-hook
56      cov+, ; is :-hook
57  :noname defers if-like
58      postpone cov+ ; is if-like
59  :noname defers until-like
60      postpone cov+ ; is until-like
61  :noname defers basic-block-end
62      postpone cov+ ; is basic-block-end
63  :noname defers exit-like
64      true to dead-cov? ; is exit-like
65  :noname defers before-line
66      postpone cov+ ; is before-line
67
68  : cov% ( -- ) \ gforth-exp
69      \G print the coverage percentage
70      0 cover-end cover-start U+DO
71          I cell+ @ 0<> -
72      2 cells +LOOP
73      #2000 cells cover-end cover-start - */
74      0 <# '%' hold # '.' hold #s #> type
75      ."  coverage" ;
76
77  : .cover-raw ( -- ) \ gforth-exp
78      \G print all raw coverage data
79      cover-end cover-start U+DO
80          I @ .sourceview ." : " I cell+ ? cr
81      2 cells +LOOP ;
82
83  Defer .cov#
84
85  : .ansi-cov# ( n -- )
86      >r info-color error-color r@ select
87      dup Invers or attr! space r> 0 .r  attr! ;
88  : .paren-cov# ( n -- ) ."  ( " 0 .r ." ) " ;
89
90  : color-cover ( -- ) ['] .ansi-cov#  is .cov# ;
91  \G print coverage with colors
92  : bw-cover    ( -- ) ['] .paren-cov# is .cov# ;
93  \G print coverage with parents
94  \G (source-code compatible)
95  color-cover
96
97  : ?del-cover ( addr u -- n )
98      \G remove coverage comment
99      2dup s"  ( " string-prefix?  IF
100         3 dup >r /string
101         BEGIN  over c@ digit?
102       WHILE  drop 1 /string r> 1+ >r  REPEAT
103           s" ) " string-prefix?
104       IF  r> 2 +  ELSE  rdrop  0  THEN
105     ELSE  2drop  0  THEN ;
106
107 : .cover-file { fn -- } \ gforth-exp
108     \G pretty print coverage in a file
109     fn included-buffer 0 locate-line 0
110     { d: buf lpos d: line cpos }
111     cover-end cover-start U+DO
112         I @ view>filename# fn = IF
113             buf lpos
114             BEGIN  dup I @ view>line u<  WHILE
115                     line cpos safe/string type cr
116             default-color attr!
117                     locate-line  to line  0 to cpos
118             REPEAT  to lpos  to buf
119             line cpos safe/string
120             over I @ view>char cpos - tuck type
121         +to cpos  2drop
122             I cell+ @ .cov#
123             line cpos safe/string ?del-cover +to cpos
124         THEN
125     2 cells +LOOP
126     line cpos safe/string type cr
127     default-color attr!  buf type ;
128
129 : covered? ( fn -- flag ) \ gforth-exp
130 \G check file number @var{fn} has coverage information
131     false cover-end cover-start U+DO
132         over I @ view>filename# = or
133     2 cells +LOOP  nip ;
134
135 : .coverage ( -- ) \ gforth-exp
136     \G pretty print coverage
137     cr included-files $[]# 0 ?DO
138         I covered? IF
139             I [: included-files $[]@ type ':' emit cr ;]
140             warning-color color-execute
141             I .cover-file
142         THEN
143     LOOP ;
144
145 : annotate-cov ( -- ) \ gforth-exp
146     \G annotate files with coverage information
147     included-files $[]# 0 ?DO
148         I covered? IF
149           I [: included-files $[]@ type ." .cov" ;] $tmp
150           r/w create-file dup 0= IF
151       drop { fd }
152           I ['] .cover-file
153       fd outfile-execute
154       fd close-file throw
155           ELSE
156           I [: included-files $[]@ type space
157           .error-string cr ;] warning-color color-execute
158           drop  THEN \ ignore write errors
159         THEN
160     LOOP ;
161
162 \ load and save coverage
163
164 $10 buffer: cover-hash
165
166 : hash-cover ( -- addr u ) \ gforth-exp
167     cover-hash $10 erase
168     cover-end cover-start U+DO
169         I cell false cover-hash hashkey2
170     2 cells +LOOP
171     cover-hash $10 ;
172
173 : cover-filename ( -- addr u ) \ gforth-exp
174     "~/.cache/gforth/" 2dup $1ff mkdir-parents drop
175     [: type
176         hash-cover bounds ?DO  I c@ 0 <# # # #> type LOOP
177     ." .covbin" ;]
178     ['] $tmp $10 base-execute ;
179
180 : save-cov ( -- ) \ gforth-exp
181     \G save coverage counters
182     cover-filename r/w create-file throw >r
183     cover-start cover-end over - r@ write-file throw
184     r> close-file throw ;
185
186 : load-cov ( -- ) \ gforth-exp
187     \G load coverage counters
188     cover-filename r/o open-file dup #-514 = IF
```

```
189            2drop true [: ." no saved coverage found" cr ;]
190        ?warning
191            EXIT  THEN  throw  >r
192        cover-start r@ file-size throw
193        drop r@ read-file throw
194        cover-start + cover-end!
195        r> close-file throw ;
196
197    true to coverage?
198
199    \ coverage tests
200
201    [defined] test-it [IF]
202        : test1 ( n -- )  0 ?DO  I 3 > ?LEAVE I . LOOP ;
203        : yes ." yes" ;
204        : no  ." no" ;
205        : test2 ( flag -- ) IF  yes  ELSE  no  THEN ;
206    [THEN]
```

date.fs

```
1    \ convert day since 0-3-1 to ymd
2    \ public domain
3
4    : /mod3 ( n1 n2 -- r q )
5        dup >r /mod dup 4 = IF  drop r@ + 3  THEN
6        rdrop ;
7
8    : day2dow ( day -- dow )  1+ 7 mod ;
9
10   \ julian calendar
11
12   : j-day2ymd ( day -- y m d )
13       1461 /mod 4 * swap
14       365 /mod3 rot + swap
15       31 + 5 153 */mod swap 5 / >r
16       2 + dup 12 > IF  12 - swap 1+ swap  THEN
17       r> 1+ ;
18
19   : (ymd2day) ( y m d -- day year/4 )
20       1- -rot
21       2 - dup 0<= IF  12 + swap 1- swap  THEN
22       153 5 */mod swap 0= >r
23       31 - swap 4 /mod swap 365 * swap
24       >r + + r> swap r> + 1+ swap ;
25
26   : j-ymd2day ( y m d -- day )  (ymd2day) 1461 * + ;
27
28   \ gregorian calendar
29
30   1582 10 15 (ymd2day) 1 0 d+ 2Constant gregorian.
31   1582 10 5 j-ymd2day Constant gregorian
32
33   : day2ymd ( day -- y m d )
34       dup gregorian >= IF
35           2 - 146097 /mod 400 * swap
36           36524 /mod3 100 * rot + swap
37           j-day2ymd 2>r + 2r>
38       ELSE
39           j-day2ymd
40       THEN ;
41
42   : ymd2day ( y m d -- day )
43       (ymd2day)
44       over 1+ over gregorian. d< 0= IF
45           25 /mod swap 1461 * swap
46           4 /mod swap 36524 * swap
47           146097 * + + + 2 +
48       ELSE
49           1461 * +
50       THEN ;
51
52   [defined] t{ [defined] cov% and [IF]
53       t{ 0 3 1 ymd2day dup day2dow
54           -> 0 1 }t cov% cr
55       t{ 1582 10 15 ymd2day 1- day2ymd
56           -> 1582 10 4 }t cov% cr
57       t{ 1400 3 1 ymd2day 1- day2ymd
58           -> 1400 2 29 }t cov% cr
59       t{ 2018 1 1 ymd2day 1- day2ymd
60           -> 2017 12 31 }t cov% .coverage
61     \ The tests up to here are sufficient
62     \ for a full code coverage.
63     \ They are not sufficient to ensure functionality.
64       t{ 1900 3 1 ymd2day 1- day2ymd
65           -> 1900 2 28 }t cov% cr
66       t{ 1582 10 4 ymd2day 1+ day2ymd
67           -> 1582 10 15 }t cov% cr
68       13 1 [DO] t{ 2018 [I] 13 ymd2day day2ymd
69           -> 2018 [I] 13 }t [LOOP] cov% cr
70       32 1 [DO] t{ 2018 12 [I] ymd2day day2ymd
71           -> 2018 12 [I] }t [LOOP] cov% cr
72       t{ 2018 2 1 ymd2day 1- day2ymd
73           -> 2018 1 31 }t cov% cr
74       t{ 2018 3 1 ymd2day 1- day2ymd
75           -> 2018 2 28 }t cov% cr
76       t{ 2018 4 1 ymd2day 1- day2ymd
77           -> 2018 3 31 }t cov% cr
78       t{ 2018 5 1 ymd2day 1- day2ymd
79           -> 2018 4 30 }t cov% cr
80       t{ 2018 6 1 ymd2day 1- day2ymd
81           -> 2018 5 31 }t cov% cr
82       t{ 2018 7 1 ymd2day 1- day2ymd
83           -> 2018 6 30 }t cov% cr
84       t{ 2018 8 1 ymd2day 1- day2ymd
85           -> 2018 7 31 }t cov% cr
86       t{ 2018 9 1 ymd2day 1- day2ymd
87           -> 2018 8 31 }t cov% cr
88       t{ 2018 10 1 ymd2day 1- day2ymd
89           -> 2018 9 30 }t cov% cr
90       t{ 2018 11 1 ymd2day 1- day2ymd
91           -> 2018 10 31 }t cov% cr
92       t{ 2018 12 1 ymd2day 1- day2ymd
93           -> 2018 11 30 }t cov% cr
94     2100 1904 [DO]
95       t{ [I] 3 1 ymd2day 1- day2ymd
96           -> [I] 2 29 }t
97     4 [+LOOP]
98     2000 1700 [DO]
99       t{ [I] 3 1 ymd2day 1- day2ymd
100          -> [I] 2 28 }t
101    100 [+LOOP] cov% cr
102    1620 1560 [DO]
103      t{ [I] 1 3 ymd2day day2ymd
104          -> [I] 1 3 }t
105    [LOOP] cov% cr
106    7  0 [DO]
107      t{ 1896 [I] + 12 13 ymd2day day2dow
108          -> [I] }t
109    [lOOP]    cov% cr
110    2000 1 1 ymd2day 1461 bounds [DO]
111      t{ [I] day2ymd ymd2day
112          -> [I] }t
113    [LOOP] cov% cr
114    1580 1 1 ymd2day 1461 bounds [DO]
115      t{ [I] day2ymd ymd2day
116          -> [I] }t
117    [LOOP] cov% cr
118      .coverage
119      #ERRORS @ [IF]
120      error-color attr!  ." had " #ERRORS ? ." errors"
121      [ELSE]
122        info-color attr!  ." passed successful"
123      [THEN]
124        default-color attr! cr cov% cr
125  [THEN]
```

Clock Works 6 — Die UTC–Funkuhr

Erich Wälde

Pepe sagte sinngemäß: „Ich hätte gerne eine Uhr mit einer großen Anzeige, die sich selbst stellt und die UTC anzeigt — und zwar ohne Schluckauf bei der Sommerzeit–Umstellung.“ *Das war die ursprüngliche Aufgabe, die ich vor ein paar Jahren aufgegriffen habe. Nach allerhand Umwegen und Ablenkungen habe ich eine solche Uhr jetzt tatsächlich selbst gebaut.*

Im sechsten Teil dieser Artikelreihe ([1], [2], [3], [4], [5]) stelle ich vor, wie ich die Abtastung des DCF77–Signals gelöst habe. Aus dem Signal wird die Uhrzeit gewonnen und mit der Uhrzeit des Mikrocontrollers verglichen.

Wie man die aktuelle Uhrzeit aus dem DCF77–Signal gewinnen kann, habe ich vor langer Zeit (2007) schon einmal vorgestellt [12], damals allerdings auf dem Renesas R8C13–Controller programmiert in gforth–ec.

Der Plan

In dem alten Artikel zum Thema [12] hatte ich schon eine Menge Entscheidungen getroffen, wie ich mit der Auswertung des DCF77–Signals verfahren wollte. Diese sind mit kleinen Änderungen geblieben. Es gilt weiterhin die Annahme, dass das DCF77–Signal jederzeit stark gestört sein kann oder ganz ausbleibt. Um in diesem doch recht umfangreichen Projekt dem Leser eine reelle Chance zu geben, kann ein Überblick in Stichpunkten nicht schaden.

- Es gibt mehrere (Software–)Uhren: die `MasterClock`, die `dcfClock` und die externe DS3231–RTC
- Die externe (batteriegepufferte) DS3231–RTC spendiert das Gedächtnis der Zeit über einen Stromausfall hinweg (angebunden via I^{2}C)
- Die externe DS3231–RTC (mit TCXO) spendiert ein stabiles 32768–Hz–Signal, aus dem der Mikrocontroller via timer/counter0 den Tick (128/s) gewinnt
- Das Controller–Programm ist auf 2 Tasks verteilt: Task1 bedient die serielle Schnittstelle; Task2 betreibt die Uhr/Uhren und die zugehörigen periodischen Jobs (Buchhaltung, Anzeige)
- Die `MasterClock` läuft in UTC, nicht in Lokalzeit
- Die Buchhaltung der `MasterClock`–Zeit wird in der Hauptschleife von Task2 (`run-masterclock`) durchgeführt. Wenn ein Tick vergangen ist, wird `job.tick` aufgerufen. Wenn eine Sekunde vergangen ist, wird `MasterClock timeup` aufgerufen (darin werden die Zähler der Uhr weitergezählt), sowie nacheinander ggf. die Jobs von `job.sec` bis `job.year` aufgerufen.
- Aus rein ästhetischen Gründen habe ich die Innereien von `run-masterclock` auf Halbsekunden–Intervalle umgebaut. Ich wollte eine LED nach der halben Sekunde ausschalten, ohne in jedem Tick zu prüfen, ob es denn schon so weit ist.
- Den Phasenakkumulator habe ich ausgebaut — der wird nur benötigt, wenn man die Frequenz der Ticks rechnerisch korrigieren will. Mit der DS3231–RTC als frequenzgebende Instanz sehe ich im Moment keine Notwendigkeit dafür.

Jetzt kommt das Thema DCF77–Zeit dazu.

- Es gibt eine zusätzliche Software–Uhr: `dcfClock`. Es gibt damit alle Zähler der Uhr (Sekunden, Minuten etc.) zweifach: einmal in der `MasterClock` und einmal in der `dcfClock`.
- Die (absolute) Zeit der `dcfClock` wird (gelegentlich, z. B. beim Start des Programms) aus dem dekodierten DCF77–Signal übernommen.
- Das DCF77–Signal transportiert in jeder Sekunde ein Bit an Information. Ein komplettes Datentelegramm beinhaltet Zeit/Zeitzone/Datum gültig für den Beginn der nächsten Minute.
- Die `dcfClock` wird, wie die `MasterClock`, aus dem o. g. Tick angetrieben, beide laufen gleich schnell.
- Die Buchhaltung der `dcfClock` wird über `job.tick` regelmäßig aufgerufen (Funktion `dcf.tick`).
- Bei jedem Aufruf von `dcf.tick` wird das DCF77–Signal abgetastet. Es werden keine Interrupts verwendet, die von den Flanken des Signals ausgelöst würden.
- Wird eine führende Flanke des DCF77–Signals (genauer des aktuellen Bit–Wertes) erkannt, dann wird der Tick–Zähler von `dcfClock` so angepasst, dass bei dieser Flanke eben eine DCF–Sekunde beginnt.
- Der Beginn der Sekunde in der `MasterClock` und der Beginn der Sekunde im DCF77–Signal werden normalerweise nicht zusammenfallen — zumindest nicht im unsynchronisierten Zustand.
- Deswegen wird die Funktion `dcf.sec` nicht in `job.sec` aufgerufen, sondern nach 128 DCF–Ticks.

Wir wissen jetzt, wann im DCF77–Signal eine Sekunde startet.

- Die Abtastung des DCF77–Signals findet 128 Mal pro Sekunde statt. Wird eine 1 gelesen, inkrementiere ich den Zähler `dcfPulse`
- Am Ende einer Sekunde, genauer nach 128 DCF–Ticks, wird der Wert von `dcfPulse` mit den erwarteten Werten verglichen:
  0,1: es ist kein Bit empfangen worden, das zeigt normalerweise den Beginn einer neuen Minute an.
  9..12: (100 ms Puls) es wurde eine Null empfangen
  20..24: (200 ms Puls) es wurde eine Eins empfangen
  sonst: Fehler
- Ist das Bit gültig, dann wird es bewertet und ggf. in den zugehörigen Zähler (DCF–Minuten, DCF–Stunden etc.) aufgenommen.

- Jedes empfangene Bit hat eine Bedeutung, die mit einer entsprechenden Funktion ausgewertet werden soll. Um welches Bit es sich handelt, steckt im Wert der aktuellen DCF–Sekunde. Man könnte nun eine Funktion mit einem gigantischen `case`–Block schreiben, aber das hat mir nicht gefallen. Alternativ kann man die XTs der nötigen Funktionen in einer Sprungtabelle im Flash–Speicher vorrätig halten.
- Die aktuelle DCF–Sekunde wird als Index in diese Tabelle verwendet. Darin findet sich das *execution token* XT einer Funktion, die die notwendigen Handgriffe tut. Diese Funktionen wurden alle mit `noname:` definiert, sind also nur über die Tabelle erreichbar.
- Die Sprungtabelle wird zur *Compile*–Zeit im RAM aufgebaut und danach ins Flash übertragen. Der verwendete Bereich im RAM wird danach wieder freigegeben.
- In diesen Funktionen wird auch die Auswertung der Paritäts–Bits und der übrigen Bits (Sommerzeit?) vorgenommen.
- Um festzustellen, ob ein Telegramm fehlerfrei ist, werden die relevanten Prüfschritte gezählt. Nur wenn alle Prüfschritte bestanden wurden, gilt das Telegramm als fehlerfrei.

Jetzt haben wir den Anfang der DCF–Sekunden, den Anfang der DCF–Minute und ein fehlerfreies Telegramm. In diesem Fall wird auf Wunsch die Information aus dem Telegramm in die Uhr `dcfClock` übertragen. Diese Zeit wird außerdem in die `MasterClock` und die DS3231–RTC übertragen. Damit hat sich die Uhr gestellt.

- Bei der Übernahme der Zeit ist die Zeitzone und die Sommerzeit im DCF–Telegramm zu berücksichtigen.
- Wurde die `MasterClock` gestellt, dann ist auch der zugehörige Wert in Epochensekunden neu zu berechnen. Aus diesem wird die Zeit für die Anzeige gewonnen.
- Für die Anzeige der Zeit auf dem Display werden die Pegel an zwei Pins ausgewertet, die über die gewünschte Zeitzone der Anzeige entscheiden. Die Anzeige wird aus der Zeit in Epochensekunden gewonnen, für die die Korrektur der lokalen Zeitzone sehr einfach ausfällt (Offset addieren).
- Die Anzeige der Lokalzeit kennt keine Sommerzeit. Diese ist durch Auswahl der entsprechenden Zeitzone zu realisieren.

Diese Übersicht ist schon länglich, der expandierte Programmtext (alle `include`–Anweisungen aufgelöst) mit Kommentaren beträgt etwa 2200 Zeilen — das ist keine Kleinigkeit, die man mal an einem halben Nachmittag geschwind hinprogrammiert.

Das Programm

Code 1: Anschluss des DCF–Empfängers

Der DCF77–Empfänger ist ein Modul von Conrad (641138), welches die 5–V–Spannungsversorgung verkraftet. Das invertierte Signal liegt an Pin D.7 an. Außerdem ist die LED an Pin B.2 zum Anzeigen des Signals vorgesehen.

```
PORTB 2 portpin: led.0
: led_dcf   [: led.0 ;] execute ;

PORTD 7 portpin: _dcf
```

Code 2: Die Zähler der DCF–Uhr

Die Zähler der Uhr `dcfClock`, die nach dem DCF–Signal gestellt wird, werden in gleicher Weise bereitgestellt wie die `MasterClock`. Dabei hatte ich mich im letzten Artikel noch gefragt, ob ich `clock:` so schnell noch einmal brauchen würde.

```
s" DCF" clock: dcfClock
#include ramtable_to_flash.fs
#include dcf_01.fs
```

Funktionen wie `timeup` funktionieren in gleicher Weise für diesen neuen Satz von Zählern.

Code 3: `dcf.tick`: Abtastung des DCF77–Signals

Für eine Zusammenfassung des Übertragungsprotokolls sei auf Wikipedia [10] verwiesen. Die Abtastung wird via `job.tick` an die Funktion `dcf.tick` weitergereicht. Diese ist eine *Deferred*–Funktion, welche den Aufruf an `(dcf.tick)` weiterreicht. Diese Funktion ist eher kurz, benötigt aber eine Reihe kleiner Hilfsfunktionen:

```
variable dcfTick              \ separate tick for dcf
: dcfTick++/mod ( -- )
  dcfTick @ 1+ ticks/sec mod dcfTick !
;
variable dcfEdge              \ edge detector
#4  constant dcfEdge:leading.bits
1 dcfEdge:leading.bits lshift 1-
    constant dcfEdge:leading.mask
: dcfEdge<<1     dcfEdge @ 1 lshift dcfEdge ! ;
: dcfEdge+1      1 dcfEdge +! ;
: dcfEdge?       dcfEdge @ dcfEdge:leading.mask = ;
: dcfTick.set    dcfEdge:leading.bits dcfTick ! ;

: (dcf.tick)
    dcfClock  1 tick +!       \ let dcfClock tick
    dcfTick++/mod             \ .
    dcfEdge<<1                \ . edge detection

    _dcf pin_low? if          \ dcf "active"
        dcfPulse++
        dcfEdge+1
        -1 dcf.led
    else
        0 dcf.led             \ dcf "idle"
    then

    dcfEdge? if               \ keep dcfClock in
        dcfTick.set           \ sync with radio
    then                      \ signal (brute force)

    \ dcfTick: 0 .. 127
    \ 127 indicates the last tick in this second
    dcfTick @ ticks/sec 1- = if
        dcf.sec               \ "one second over"
    then
;
```

In den Zählern der `dcfClock` wird `tick` erhöht. Dann wird der Wert an Pin `_dcf` gelesen. Der Bitwert wird in die Variable `dcfEdge` hineinrotiert. Außerdem wird die Variable `dcfPulse` erhöht. Das Ausgeben des Signalwerts auf der LED ist eine ästhetische Sache, man kann auch bei abgesetztem Empfänger den Empfang *sehen*.

Die Variable `dcfEdge` wird danach noch einmal ausgewertet: Beträgt der Wert `$000F`, dann wurde nach 12–mal dem Wert 0 jetzt 4–mal der Wert 1 gesehen — das interpretiere ich als führende Flanke eines Bits des DCF–Signals und setze dreist den Wert der dcfTicks auf 4. Dieser Schritt synchronisiert die Sekunden der `dcfClock` mit der Bitfolge des empfangenen Signals. Zumindest ist es der Wunsch.

Nach 128 dcfTicks (0...127) wird außerdem die Funktion `dcf.sec` aufgerufen. Diese wertet das empfangene Bit aus. Dieser Aufruf ist bewusst nicht in den `job.sec` der `MasterClock` eingebunden, weil diese beiden Uhren nicht synchron gehen werden — auch wenn das der Wunsch ist.

Streng genommen bräuchte es `dcfTick` vielleicht nicht. Die Zähler von `dcfClock` enthalten ja ebenfalls den Zähler `tick`, und der wird auch gepflegt. Andererseits ist `dcfTick` so etwas wie ein Schleifenindex, der keine anderen Aufgaben beinhalten soll und der möglicherweise zur Unzeit verdreht wird.

Code 4: `dcf.sec`: Auswertung des DCF–Bits

Am Ende der DCF–Sekunde wird also die Auswertung des gerade empfangenen Bits durchgeführt. Auch `dcf.sec` ist ein Funktionszeiger, er zeigt gewöhnlich auf `(dcf.sec)`. Diese Funktion ist länger. Zunächst wird die Buchhaltung der Zeit durchgeführt, schließlich ist eine Sekunde vergangen:

```
-2 constant dcf.bit:sync59
-1 constant dcf.bit:error
 0 constant dcf.bit:0
 1 constant dcf.bit:1
: dcf.pulse>bit ( pulse -- bit|error )
  dup  #2 < if                dcf.bit:sync59 else
  dup  #9 < if f.dcf.err fset dcf.bit:error  else
  dup #13 < if                dcf.bit:0      else
  dup #20 < if f.dcf.err fset dcf.bit:error  else
  dup #26 < if                dcf.bit:1      else
               f.dcf.err fset dcf.bit:error
  then then then then then
  swap drop
;

Rdefer dcf.sec
: (dcf.sec)
  dcfClock timeup
  0 tick !
```

Danach wird der Zähler `dcfPulse` bewertet:

```
  dcfPulse @ dcf.pulse>bit  dup dcfBit !
```

In der Funktion `dcf.pulse>bit` wird festgestellt, in welchen Wertebereich der Zählerstand von `dcfPulse` fällt. Die Rückgabewerte bedeuten:

`dcf.bit:1` Es wurde eine Eins empfangen
`dcf.bit:0` Es wurde eine Null empfangen
`dcf.bit:error` Fehler: Der Zählerwert liegt außerhalb der für gut befundenen Bereiche
`dcf.bit:sync59` Es wurde die DCF–Sekunde 59 erkannt, der Minutenwechsel steht an

Normalerweise (kein Minutenwechsel) wird das empfangene Bit ausgewertet. Diese Auswertung führt der Aufruf der Funktion `pos.cmd` durch. Auf dem Stack werden der Bitwert (`0` oder `1`) sowie die Position im DCF–Telegramm (aktuelle Sekunde) (`dcfPos @`) erwartet. `dcfPos` wird als Index in eine Sprungtabelle verwendet, das dort gefundene `XT` wird aufgerufen und konsumiert den Bitwert. Das verbirgt sich hinter der Design–Entscheidung *"Die Auswertung der aufgenommenen Bits erfolgt fortlaufend jede Sekunde"*.

```
  ( bit ) case

    dcf.bit:0      of
      0 dcfPos @ pos.cmd   dcf.dbg.sec
    endof

    dcf.bit:1      of
      1 dcfPos @ pos.cmd   dcf.dbg.sec
    endof

    dcf.bit:error  of
      f.dcf.err fset       dcf.dbg.sec
    endof
```

Beim Minutenwechsel lauert Mehrarbeit, die sich hinter dem Aufruf von `dcf.min` und etwas Aufräumen verbirgt.

```
    dcf.bit:sync59 of
      dcf.min
      dcf.tmp.counter.reset
      dcfErrCnt.set
    endof

  endcase
```

Abschließend wird `dcfPulse` gelöscht und die aktuelle Sekunde in `dcfPos` erhöht. Auch dieser Zähler ist eine Art Schleifenindex und den wollte ich nicht mit dem Zähler `sec` in `dcfClock` gleichsetzen.

```
  dcfPulse.clr
  dcfPos++
;
```

Code 5: Die Auswertungsfunktionen

Die Auswertung des empfangenen Bits wird an die zugehörige Funktion übertragen. Diese Funktionen erwarten den Wert des Bits auf dem Stapel. Dieser Wert kann verworfen oder ausgewertet werden. Alle Auswertungsfunktionen haben den gleichen Aufbau. Ihre Verwaltung wird im nächsten Abschnitt verdeutlicht.

```
:noname   ( 0|1 -- )
    drop ( always ) ...
;                               #idx >rt
:noname   ( 0|1 -- )
    if   ( bit:1  ) ...
```

```
    else ( bit:0  ) ...
    then ( always ) ...
;                               #idx >rt
```

Die Verarbeitung eines jeden Bits über eine eigene Funktion mag möglicherweise sehr überkandidelt erscheinen. Aber eine einzige Funktion, die das komplette Wissen über das DCF–Telegramm in sich trägt, wird auf keinen Fall kürzer. Dabei muss man die Bitpositionen in den Zählern bei jedem Durchgang buchhalterisch ebenso mitführen, wie die Umrechnung von BCD in Dezimal. Habe ich in meinen allerersten Versuchen (in PIC–Assembler!) so gemacht — das ist nicht übersichtlich geworden. Forth bietet deutlich ausdrucksstärkere Sprachelemente, also soll man die auch nutzen!

Die hier vorgestellten einzelnen, namenlosen Funktionen belegen vielleicht mehr Platz im Programmspeicher — jede einzelne ist aber so klein, dass ihre Aufgabe m. E. schneller verstanden werden kann. Ich konnte so auch mit einem überschaubaren Anteil an Funktionen anfangen und den Rest unbesetzt lassen. Und am Ende finde ich meinen Programmtext so besser lesbar — auch wenn die Zeilen leider recht breit ausfallen.

Code 6: Die Verwaltung der Bewertungsfunktionen in einer persistenten (Sprung–)Tabelle

Die Geschichte mit der Sprungtabelle sieht auf den ersten Blick etwas unübersichtlich aus, ist aber keinesfalls neu (s. [16], [13]). Am Beispiel von Bit 25, dem ersten (niedrigsten) Bit der Minuten–Zehner sei das verdeutlicht. Die zugehörige Funktion benötigt keinen Namen, wird also mit `:noname` definiert.

```
:noname
  if ( bit:1 )  #10 dcfTMin   +!  dcfPar++
  then
;                              #25 >rt \ minute tens
```

Das `if` konsumiert den Bitwert (`0` oder `1`) vom Stapel. Hat das Bit den Wert `1`, dann wird seinem Wert entsprechend `#10` zum Minutenzähler `dcfTMin` addiert. Bemerke: dezimal–10, nicht hexadezimal. Damit wird nebenbei auch die Umwandlung von BCD–kodierten Zahlen (aus dem DCF–Telegramm) in Dezimalzahlen (Variablen im Programm) vorgenommen. Außerdem wird der Wert, der die Parität aufsammelt, erhöht. Der `else`–Zweig ist in dieser Funktion leer. Damit diese Funktion jemals wiedergefunden wird, kopiert `>rt` das von `noname:` auf dem Stapel hinterlassene `XT` (*execution token*) unter dem Index `#25` in eine Tabelle im RAM.

`(dcf.sec)` findet also in der Tabelle unter dem Index 25 diese Funktion wieder. Damit das auch nach einem Stromausfall klappt, wird die Tabelle zur *Compile*–Zeit ins Flash kopiert:

```
\ create temporary RAM table
\ only needed during compile time
variable dcf.tmp.table #60 cells allot

\ fill RAM table with ' drop
' drop dcf.tmp.table #60 ramtable.init

: >rt ( xt idx -- )  dcf.tmp.table >ramtable ;

\ define all functions ...
:noname ( bit -- )  ... ; #index >rt
...

\ copy RAM table to FLASH
dcf.tmp.table #60 >flashtable pos_cmd_map
\ release RAM
dcf.tmp.table to here

: pos.cmd ( index -- )
  dup 0 #60 within if
    ( position ) pos_cmd_map +  @i execute
  else
    drop
  then
;
```

Sind alle `XT`s erzeugt und im RAM abgelegt, kopiert `>flashtable` die Werte ins Flash, wo sie unter der Anfangsadresse `pos_cmd_map` zugänglich sind. Damit das schöne RAM — immerhin 60 Zellen — nicht unnütz vergeigt wird, merken wir uns vorher den Anfang und kopieren diesen Wert am Ende zurück in den Zeiger `here`. Solange zwischen `variable dcf.tmp.table` und `dcf.tmp.table to here` keine weiteren Variablen definiert werden, ist alles gut — auch wenn es vielleicht schlimm aussieht. Die Variable `dcf.tmp.table` existiert zwar weiterhin, sollte aber nicht mehr benutzt werden, weil ihr Speicher im RAM neu vergeben wird.

Code 7: dcf.min

Die weitere Verarbeitung des Telegramms fällt also nur einmal am Ende der Minute an. Das Telegramm enthält die Daten, die zum Beginn der nächsten Minute gültig sind. Diese Verarbeitung wird durch den harmlos aussehenden Aufruf von `dcf.min` in der oben gezeigten Funktion `(dcf.sec)` ausgeführt.

Wir sind am Ende der aktuellen Minute und am Ende des aktuellen Telegramms angekommen. Hier wird also die abschließende Bewertung desselben vorgenommen. Stimmt die Position im Telegramm? Wenn ja, sind alle Tests erfolgreich gewesen und der Zähler `dcfErrCnt` enthält eine Null? Dann ist das Telegramm gut.

```
: (dcf.min) ( -- )
  dcfPos @ #59 =  if dcfErrCnt-- then
  dcfErrCnt @ 0=  if f.dcf.err fclr then
```

Wenn alles gut ist und wenn die Übernahme der Zeit erwünscht ist, dann soll das auch geschehen.

```
  \ this block runs at the *end* of second "59" i.e.
  \ at the start of second "00", thus copy counters
  f.dcf.err fclr? if
    f.dcf.commit fset? if
      dcfTemp>dcfClock
      dcfClock>MasterClock
      space [char] C emit
      f.dcf.insync fset \ dcfClock is in sync!
```

```
      f.dcf.commit fclr
    then
  then
```

Am Ende werden weitere Debug–Ausgaben gemacht. Das Flag, welches die Debug–Ausgaben erlaubt, wird gelöscht falls die Übernahme der Uhrzeit stattgefunden hat.

```
  dcf.dbg.sec
  dcf.dbg.min
  f.dcf.commit fclr? if
    \ clear debug flags --- demo only
    f.dcf.dbg fclr
  then
;
```

Die Übernahme der Zeit wird in *zwei separaten Stufen* vorgenommen? Das klingt zuerst nach doppelt gemoppelt. Bis man einen Blick in die Innereien riskiert.

dcfTemp>dcfClock

Die fortlaufende Auswertung des Telegramms speichert die Ergebnisse in separaten Variablen:

```
dcfTMin dcfTHour dcfTDay dcfTWday dcfTMonth dcfTYear
```

Diese Variablen ergeben aber keine *Software–Uhr*, deren Zähler regelmäßige Pflege erfahren. Eine solche Uhr wurde anders definiert (was ich beim Schreiben des Artikels leider schon vergessen hatte):

```
s" DCF" clock: dcfClock
```

Um die Zeit aus dem Telegramm in die Software–Uhr zu übernehmen, sind ein paar Anpassungen nötig: `tick` und `sec` werden auf Null gesetzt — das Ganze passiert am Anfang der neuen Minute. Der Wochentag wird ignoriert, obwohl er im Telegramm ja tatsächlich enthalten ist. Die Zähler von Tag und Monat werden um `1` erniedrigt; so werden sie gepflegt. Das Jahrhundert wird hier ebenfalls addiert (2000). Und ja, wir haben damit ein Jahr–2100–Problem! Allerdings hoffentlich einfach zu lösen durch Ändern der Konstante `Century`.

```
: dcfTemp>dcfClock
  dcfClock
  0                       tick  !
  0                       sec   !
  dcfTMin   @             min   !
  dcfTHour  @             hour  !
  dcfTDay   @  1-         day   !
\ dcfTWday  @          ( ignored )
  dcfTMonth @  1-         month !
  dcfTYear  @  Century +  year  !
;
```

Ob `tick` wirklich sinnvoll gesetzt ist, das müsste man mal untersuchen. Vielleicht ist `dcfTick` hier der bessere Wert?

dcfClock>MasterClock

Zwar muss man die Zähler jetzt noch von einer Uhr zur anderen kopieren, allerdings befinden die sich in verschiedenen Zeitzonen.

Zunächst einmal ist die Funktion ein `Rdefer`. Das ist der Organisation des Quelltextes geschuldet.

```
Rdefer dcfClock>MasterClock

: (dcfClock>MasterClock)
  dcfClock
  sec  @     min   @     hour @
  day  @ 1+  month @ 1+  year @
  MasterClock
                  \ -- sec min hour day month year
```

Der Anfang ist völlig plausibel: Die Zähler aus der Uhr `dcfClock` werden auf den Stapel gelegt, die Offsets bei Tag und Monat werden korrigiert. Im nächsten Schritt wird die "lokale" Zeit in Epochensekunden umgerechnet und die Zeitzonen korrigiert.

```
  \ convert "local time" to epoch seconds
  ut>s.short        \ -- T/sec
  \ adjust local time zone
  f.dcf.CEDT fset? if
    #3600. d-
  then
  #3600. d-
  2dup Esec 2! \ copy to Epoch seconds!
```

Der ebenfalls vorhandene Zähler der Epochensekunden wird gestellt. Danach wird die Zeit wieder in das konventionelle Format in der Zeitzone UTC zurückgerechnet und in die `MasterClock` übernommen.

```
  \ convert back to "UTC date time" format
  s>dt.short        \ -- sec min hour day month year

  MasterClock
  year !  1- month !  1- day   !
  hour !  min      !  sec      !
```

Allerdings wollte ich die ticks jetzt noch *verbessern*. Ob das wirklich hilft, erscheint mir im Moment aber zweifelhaft.

```
  \ fixme: might cause wreaking havoc?
  dcfClock tick @
  dup MasterClock tick !
  ct.ticks.follow !
;
```

An dieser Stelle habe ich noch Zweifel, ob das alles so toll gelöst ist. Denn ich beobachte, dass die übernommene Zeit vom Blinken der DCF–LED oft um ca. eine Sekunde abweicht. Das ist für die meisten Belange unerheblich, deutet aber auf Unwissen hin. Womöglich müsste man die Aufrufe der Auswertungsfunktionen in die Mitte der Sekunde verlegen und bei der Übernahme der DCF–Zeit tatsächlich auf die eintreffende Flanke warten. Aber auch das hat so seine Spezialitäten.

Code 8: Umrechnung Epochensekunden

Die Umrechnung der Uhrzeiten in Epochensekunden und zurück ist zwar nur eine Fleißarbeit, allerdings auch nicht ganz kurz in der Berechnung [14]. Und da es recht dumm ist, die (fast) gleiche Berechnung ständig zu wiederholen, habe ich der Umrechnung eine Abkürzung spendiert. Wenn man die Berechnung nicht am 1.1.1970 anfängt, sondern vielleicht am 1.1.2017, dann sollte sich die Berechnung doch ordentlich verkürzen, oder? Immerhin spart man 47 Runden durch die Jahre. Also habe ich zwei Variablen spendiert, die den geänderten Startwert enthalten. Die Werte werden im EEPROM abgelegt und beim Start des Programms ins RAM übernommen.

```
           variable   _last_epoch
          2variable   _last_esec

#2017         Evalue   EE_last_epoch
#1483228800. 2Evalue   EE_last_esec

#include epochseconds.fs
```

Aus den beiden Umrechnungsfunktionen sei nur so viel gezeigt, dass diese Variablen tatsächlich benutzt werden.

```
: s>dt.short
   ( d:EpochSeconds -- sec min hour day month year )
    _last_esec 2@ d-
    ...
;
: ut>s.short
            ( sec min hour day month year -- T/sec )
    \ add start value T=0
    0 over
    \ -- sec min hour day month year T=0 year
    _last_epoch @
    \ -- sec min hour day month year T year Epoch
    ...
    _last_esec 2@ d+    \ -- T/sec
;
```

Es sei darauf hingewiesen, dass die Funktionen zur Umrechnung keinerlei Prüfung ihrer Argumente vornehmen. Falls der Startwert von der Anzeigezeit aus gesehen in der Zukunft liegt (Zeitzonen westlich von UTC), dann wird die Berechnung wahrscheinlich schieflaufen.

An dieser Stelle mehren sich gelegentlich auch die Zweifel, ob es schlau war, die `MasterClock` in UTC zu betreiben, und die Anzeige jedesmal neu zu berechnen, falls die Zeit in einer anderen Zeitzone angezeigt werden soll. Nichts ist umsonst. Und bestimmt gibt es noch Raum für Verbesserung. Andererseits hat der Mikrocontroller ja wirklich fast nie was zu tun…

Code 9: Debug–Ausgaben?

Oh ja, beim Entwickeln eines so umfangreichen Programms sind fortlaufende Ausgaben zum Mitlesen unverzichtbar. Das hat zwar ebenfalls seine Spezialitäten: z. B. musste ich lernen, dass die Ausgabe für Sekunde 1 halt erst am Ende dieser Sekunde kommt, für „klein Menschlein“ ununterscheidbar vom Beginn der nächsten Sekunde. Aber besser einen zeitlichen Versatz als gar keine Meldungen.

Die Ausgaben sind abschaltbar. Der Bitwert wird am Ende der Sekunde als ein Zeichen ausgegeben. Am Beginn der neuen Minute wird zusätzlich der Inhalt des Telegramms ausgegeben.

```
: +DD  f.dcf.dbg fset ;
: -DD  f.dcf.dbg fclr ;
: .DD
  ." dcf.error:  "
         f.dcf.err     fset? if 1 else 0 then . cr
  ." dcf.parity: "
         f.dcf.parerr  fset? if 1 else 0 then . cr
  ." dcf.commit: "
         f.dcf.commit  fset? if 1 else 0 then . cr
  ." dcf.debug:  "
         f.dcf.dbg     fset? if 1 else 0 then . cr
  ." dcf.dbg.rx: "
         f.dcf.dbg.rx  fset? if 1 else 0 then . cr
;
: +DRX  f.dcf.dbg.rx fset ;
: -DRX  f.dcf.dbg.rx fclr ;

: dcf.dbg.second.over ( -- )
  f.dcf.dbg.rx fset? if
    dcfBit @ case
      dcf.bit:0      of [char] . emit endof
      dcf.bit:1      of [char] + emit endof
      dcf.bit:error  of [char] / emit endof
      dcf.bit:sync59 of [char] S emit endof
    endcase
  then
;
: dcf.dbg.minute.over ( -- )
  space
  f.dcf.err fclr? if
    [char] O emit
  else
    [char] F emit
  then
  cr
;
: dcf.dbg.sec ( -- )
  f.dcf.dbg fset? if dcf.dbg.second.over then ;
: dcf.dbg.min ( -- )
  f.dcf.dbg fset? if dcf.dbg.minute.over then ;
```

Das sieht dann etwa so aus (von Hand umgebrochen, damit es in die Spalte passt):

```
> run-turnkey
amforth 6.7 ATmega644P ok
~6F> +.+++++.+..++++......++....S F
.+++..++.....++...+.+.++.+.+......+++++.+..++++.....
.++.... CS O
```

Die Umrechnerei am Ende der alten bzw. am Beginn der neuen Minute und v. a. die Debug–Ausgaben führen zu einer Verspätung der Uhr `MasterClock`. So per Augenmaß (auf die LEDs gucken) ist das so im Bereich einer Sekunde. Für den normalen Gebrauch reicht das. Für höhere Ansprüche an die absolute Genauigkeit muss man wohl andere Wege gehen.

Code 10: Öfter mal neu synchronisieren

Die Uhr der frei laufenden `MasterClock` soll gelegentlich die Zeit aus der `dcfClock` übernehmen. Käufliche Wecker machen das z. B. jede Stunde. Auch einmal pro Nacht ist denkbar. Das lässt sich beispielsweise in den Minuten–Job einbauen, der immer bei Minute 58 eine Synchronisation anfordert:

```
: dcf.resync ( -- )
  f.dcf.insync fclr \ resync dcf time
  f.dcf.commit fset \ sync dcf -> MasterClock wanted
;

: job.min
  ...
  \ update display
  _tz.set cd.localtime
  \ once per hour request re-sync
  MasterClock min @  #58 = if dcf.resync then ;
;
```

Dieses Vorgehen würde auch bei einer Uhr, deren Geschwindigkeit deutlich abweicht oder schwankt, zu einer brauchbaren Anzeige führen, solange das DCF–Signal einigermaßen zuverlässig gelesen werden kann.

Code 11: Einschalten!

Um die DCF–Funktion der Uhr zu aktivieren, sind noch ein paar wenige Handgriffe zu tun. Der Rest des Programms ist einigermaßen identisch zu der Version aus der letzten Folge.

```
: +dcf
  _dcf pin_input
  [’] (dcf.tick) to dcf.tick
  [’] (dcf.sec)  to dcf.sec
  [’] (dcf.min)  to dcf.min

  0 dcfPos !        \ assume we are at 0
  f.dcf.err fset    \ error unless proven otherwise
  f.dcf.commit fset \ request to set dcfClock
  f.dcf.insync fclr \ not in sync yet!
  dcfErrCnt.set
;
: -dcf
  [’] noop to dcf.tick
  [’] noop to dcf.sec
;
```

Schluss

Auch an dieser Uhr kann man noch eine Menge Sachen dazubauen, wenn man will. Eine kleine, spontane Auswahl:

- eine generellere Auswahl der lokalen Zeitzone
- das Datum ebenfalls anzeigen, temporär, auf Wunsch
- eine Menü–Funktion, um die Uhr stellen zu können
- mit der gleichen Menü–Funktion eine Alarmzeit stellen
- einen Wecker realisieren
- den Wochentag aus DCF77 korrekt übernehmen
- die Ankündigung der Sommerzeit–Umschaltung korrekt aus dem Telegramm übernehmen und anzeigen.
- die Ankündigung von Schaltsekunden korrekt aus dem Telegramm übernehmen, anzeigen und behandeln — man will die 60 in den Sekunden vielleicht wirklich sehen!
- einen 32–kHz–Uhrenquarz an timer2 betreiben, dessen Abweichung (Phasenakkumulator) messen und die Korrektur ausrechnen
- andere/zusätzliche Anzeigen (Epochensekunden)
- zusätzlich Sternzeit–Uhr, wahlweise Anzeige
- einen HKS–Chip spendieren, um MeteoTime–Wetterinformation zu extrahieren und anzuzeigen.

So Sachen

1 Die Empfängermodule sind möglicherweise so verschieden, dass man die Vergleichswerte für Bit:0 und Bit:1 anpassen können sollte. a) durch eine Debug-Ausgabe, b) durch EE–cached–values für die Grenzwerte.

2 Die Stromversorgung hat eine Sendeantenne namens Zuleitung. Kann schon vorkommen, dass in dem Krach das DCF77–Signal komplett absäuft. Ordentliches Industrie–(Schalt–)Netzteil? Zusätzlich Entstörung? Gehäuse? Abstand?

3 Der aktuelle Stand hat noch etliche *Phänomene*. Beispielsweise führt ein externer Reset dazu, dass das Programm nicht mehr starten will. Die Initialisierung findet statt, aber dann hängt sich das alles irgendwie lustig auf. Kann ich leider nicht gutheißen. Nachtrag: das Phänomen habe ich gefunden: es war eine Verwendung von uninitialisierten Variablen. Aber es gibt noch andere.

4 Es gibt beim Funkamateur einen Bausatz, der eine 10–MHz–Referenzfrequenz realisiert, welche über die recht genaue Trägerfrequenz von DCF77 diszipliniert wird. Die Anzeige des Telegramms fällt dabei quasi mit ab [18].

5 Es gibt bei AATiS einen Bausatz, mit dem man ebenfalls eine DCF–Uhr realisieren kann [17]. Ich habe aber nicht versucht, mein Programm zu portieren. Die Anzeige wird über zeitliches Multiplexen der Ziffern gemacht, was mir bekanntlich nicht gefällt.

Verweise

1. VD 2016-04, S.15ff — E. Wälde, Clockworks 1 Die kleine Uhr
2. VD 2017-02, S.12ff — E. Wälde, Clockworks 2 Anzeige à la Abakus
3. VD 2017-03, S.12ff — E. Wälde, Clockworks 3 Auf der Suche nach der verlorenen Zeit
4. VD 2017-03, S.23f — E. Wälde, Clockworks 4 Des Rätsels Lösung
5. VD 2017-04, S.6ff — E. Wälde, Clockworks 5 Die UTC Wanduhr
6. `https://wiki.forth-ev.de/doku.php/projects:clockworks:clockworks`

7. http://amforth.sourceforge.net/Projects/index.html
8. Forth Tagung 2017 (Kalkar), http://wiki.forth-ev.de/doku.php/events:tagung-2017, Video und Folien
9. https://de.wikipedia.org/wiki/UTC%C2%B10
10. https://de.wikipedia.org/wiki/DCF77
11. Unix–Zeit, Epochensekunden https://de.wikipedia.org/wiki/Unixzeit
12. VD-2007-01, S.10ff — E. Wälde, Adventures 4: Eine Funkuhr
13. VD-2015-0304, S.24f — E. Wälde, Permanente Tabellen im AmForth–Flash ablegen
14. VD-2015-0304, S.34ff — E. Wälde, Von Universal Time zu Epochensekunden und zurück
15. VD-2014-01, S.22ff — R. Deliano, DCF77–Funkuhr
16. VD-2012-03, S.25ff — E. Wälde, Morse 5: Eine deklarative Version
17. Uhrenbausatz von AATiS https://www.aatis.de/content/bausatz/AS324_Multiclock
18. DCF77-gesteuertes 10–MHz–Frequenznormal http://www.box73.de/product_info.php?products_id=1964

Listings

Das Programm ist viel zu lang, um es hier komplett abzudrucken. Auch nicht nur die beiden wichtigen Dateien `main-dcf.fs` und `ewlib/dcf_01.fs`. Der komplette Programmtext ist im Forth–Wiki [6] zu finden. Eine in englischer Sprache kommentierte Version findet sich auf der Webseite von AmForth unter *Commented Projects* [7].

Nachtrag: Lektüre zum DCF77

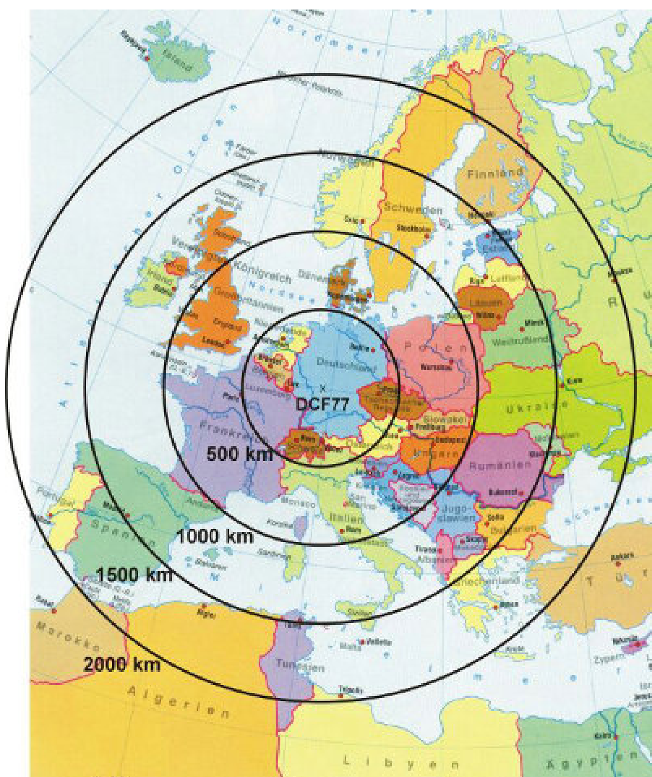


Abbildung 1: DCF77 Reichweite

So beim Stöbern zu dem Thema fand ich die Bayern–Online–EDV–Administration, Abteilung „Historische Fernmeldetechnik und Virtuelles Fernmeldemuseum“[1]. Spannend.

> Die Übertragung des Zeitsignals von einer Quelle bis zum Nutzer erfolgte anfänglich auf unterschiedlichen Wegen. Beispielsweise konnte man beim mathematisch–physikalischen Salon in Dresden die Uhrzeit abonnieren, die dann in regelmäßigen Abständen von einem Boten mit einer tragbaren Uhr überbracht wurde. [Aus: `www.bayern-online.com`; Geschichtliche Entwicklung der Zeitübertragung per Funk.] ...
>
> Alle wichtigen Zeitsignaldienste sind mit höchster Präzision vernetzt, sodass sie weltweit (von der Signallaufzeit abgesehen) im Bereich weit unter Nanosekunden übereinstimmen – siehe die Zeitsysteme UTC („Weltzeit“), TAI, TD und die Koordination durch den internationalen Erdrotations–Dienst IERS. Daher ist auch auf Nutzerseite eine hohe Genauigkeit garantiert. [ebenda]

Auf elektronischem Wege sind Genauigkeiten bis zu Nanosekunden keine Ausnahme; bei den heute in der Geodäsie verwendeten GPS–Empfängern sind bereits Systeme zur Zeitanalyse auf mindestens 0,1 ns eingebaut, was bei der Lichtgeschwindigkeit von 299792 km/s nur 3 cm ausmacht.

Der reitende Bote galoppiert nun durch den Äther, stürzt manchmal auch ab und es ist nicht immer so klar, was er meint. Das hat Erich mir hier klar gemacht. Diese Übermittlungsfehler zu überbrücken, ist die eigentliche Kunst dabei. mk

[1] http://www.bayern-online.com/v2261/artikel.cfm/203/Geschichtliche-Entwicklung-der-Zeituebertragung-per-Funk.html

Günstiger Einstieg in die FPGA–Programmierung

Klaus Kohl–Schöpe

Als Field Application Engineer (FAE) bei dem weltweit größten Bauteiledistributor ARROW *empfehle ich meistens Mikrocontroller für alle Arten von Steuerungen. Es gibt aber auch Fälle, bei denen FPGAs besser die Anforderungen für schnellen Datentransfer und Auswertung abdecken oder — gerade für Forth–Programmierer interessant — auch wieder ältere Prozessorarchitekturen bzw. Forth–Prozessoren zum Leben erwecken können. Für den schnellen Einstieg hat Arrow zusammen mit Trenz Electronic diverse günstige FPGA–Boards entwickelt, die ich hier vorstellen möchte.*

Vorgeschichte

Entwicklungsboards für FPGAs gibt es viele, weil die Hersteller selbst die Hardware für ihre Chips und Software brauchen. Jedoch haben diese oft den Nachteil des höheren Preises und dienen nur der Evaluierung, was industrielle Temperaturbereiche und eine Garantie für den Serieneinsatz ausschließt. In einigen Fällen ist auch kein Debugger bzw. Programmer integriert und muss teuer dazugekauft werden. In der FPGA–Community gibt es auch viele Boards, die aber meist für bestimmte Anwendungen gedacht und deshalb bezüglich Speicher- und Schnittstellenausstattung limitiert sind. Auch diese dürfen normalerweise nicht für industrielle Produkte eingesetzt werden.

Deshalb hat sich das ARROW ENGINEERING TEAM zusammen mit TRENZ überlegt, was man für einen günstigen Einstieg in die FPGA–Programmierung benötigt und wie man dieses Tool danach auch für Projekte in kleinen Serien nutzen kann. Da ARROW sowohl Altera[1]–, als auch Lattice– und Microsemi[2]–FPGAs vertreibt, sollte daraus eine Familie von Boards werden.

Gewünschte Features waren:

- Günstiger Preis (< 50 €)
- Integrierter Debugger mit serieller Schnittstelle
- Kleiner Formfaktor (Arduino–Nano)
- Tasten und LEDs zur Bedienung und Statusanzeige
- Zusätzlicher Programmspeicher
- Zusätzliches RAM
- Pmod–Konnektor

Innerhalb der letzten zwei Jahre wurden 4 Boards realisiert, welche bei Preisen unter 50 € die gewünschten Features realisiert haben. Da alle Hersteller für die kleineren FPGA–Derivate kostenlose Software und auch entsprechende IPs[3] für die Schnittstellen und sogar Prozessoren — NIOS II, LatticeMICO8/32 und jetzt auch RISC–V — zur Verfügung stellen, ist dies alles, was man investieren muss.

Diese Boards werden über den Micro–USB–Konnektor versorgt und programmiert. In dem damit verbundenen, auch separat verfügbaren, Arrow–USB–Programmer–2 ist auch eine USB–UART–Bridge integriert und damit immer eine serielle Schnittstelle für die Prozessor–IP verfügbar. So ein Board kann auch über zwei Taster bedient werden und hat 8 LEDs als Anzeige. Die zusätzlich bestückbaren Arduino–Nano– und Pmod–Konnektoren erlauben den Anschluss externer Hardware[4].

MAX1000 (TEI0001)

- Intel MAX10 mit 8000 LE[5] (integrierter Flash und 1–MSPS–ADC)
- Arrow–USB–Programmer–2 mit USB–UART–Bridge
- 8 MByte QuadSPI
- 8 MByte SDRAM
- 2 Buttons & 8 LEDs
- 3–Achsen–Beschleunigungsaufnehmer

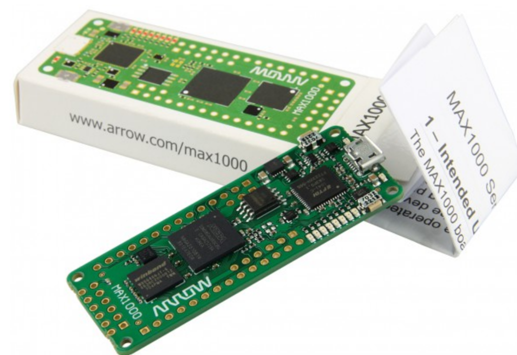


Abbildung 1: MAX1000

Für ca. 26 € — bei Arrow aktuell mit kostenlosem Versand — erhalten Sie die 8–kLE–Version. Aber auch größere Versionen, z. B. mit 16 kLE bzw. 32 kLE und 32 MByte

[1] Jetzt Intel.

[2] Jetzt Microchip.

[3] Intellectual Property — vorgefertigter Funktionsblock eines Chipdesigns; `https://de.wikipedia.org/wiki/IP-Core`

[4] Pmod interface (or Peripheral Module interface) [Wikipedia]; hier ist ein 12–Pin–Konnektor eingebaut, definiert von Digilent für I/O, SPI, UART usw.
`https://reference.digilentinc.com/_media/reference/pmod/pmod-interface-specification-1_2_0.pdf`

[5] Logic Elements — Logikblock meist mit einer (4–Bit–)Lookup–Tabelle (LUT) und einem D–FlipFlop; kLE sind 1000 LE.

[6] Million Samples Per Second = 1 Mikrosekunde/Messung.

SDRAM sind verfügbar. Die MAX10–Serie hat ein integriertes Flash und einen 1–MSPS–12–Bit–A/D–Wandler[6]. Der Chip ist deshalb sofort nach dem Start ohne zusätzlichen Download aktiv, kann aber auch das externe SDRAM als Programmspeicher nutzen und dafür auch große Programme aus dem hier verfügbaren QuadSPI laden. Über die Trenz–Webseite sind alle Informationen wie User–Guide, Schematics und Beispielprogramme, meist für den auf MAX10 üblichen NIOS–II–Prozessor, verfügbar.

CYC1000 (TEI0003)

- Intel Cyclone 10 LP (25 kLE)
- Arrow–USB–Programmer–2 mit USB–UART–Bridge
- 2 MByte Flash
- 8 MByte SDRAM
- 2 Buttons & 8 LEDs
- 3–Achsen–Beschleunigungsaufnehmer

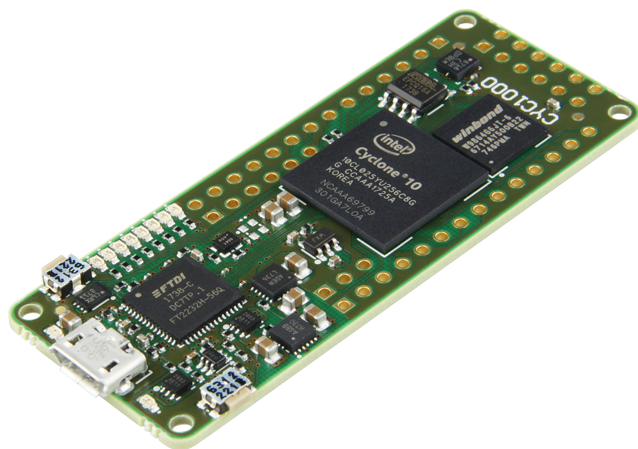


Abbildung 2: CYC1000

Ca. 35 € muss man investieren, wenn man die neuere Cyclone–10–LP–Familie testen will. Da mehr LUTs verfügbar sind, können auch größere IPs geladen werden. Ansonsten ist das Board ähnlich dem MAX1000.

SMF2000 (TEM0001)

- Microchip SmartFusion2 (integrierter Flash, Cortex–M3 und 12 kLE)
- Arrow–USB–Programmer–2 mit USB–UART–Bridge
- 8 MByte QuadSPI
- 8 MByte SDRAM
- 2 Buttons
- 8 LEDs

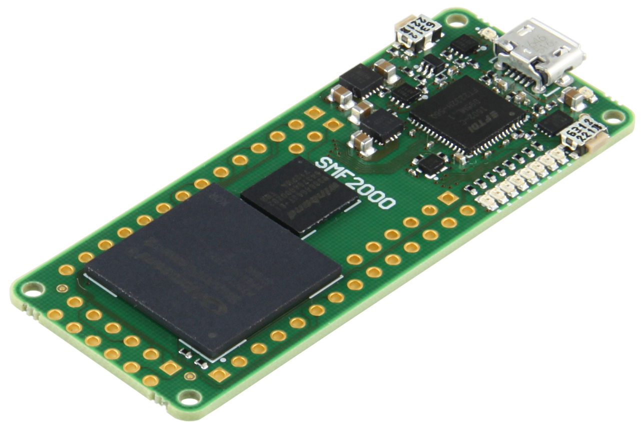


Abbildung 3: SMF2000

Mit 42 € schon etwas teurer ist der auf Microsemi SmartFusion2 basierende SMF2000. Dafür erhält man eine schon im Chip integrierte 166–MHz–Cortex–M3–Hard–IP, 256 KByte Flash, 80 KByte SRAM, sowie einen 12–kLUT–FPGA–Teil. Dies wird dann extern um 8 MByte QuadSPI–Flash, 8 MByte SDRAM und den üblichen Tasten und LEDs ergänzt. Dadurch könnte man eine eigene Prozessor–IP im FPGA durch den Cortex–M3 bedienen und Features wie Programm–Download, Debugging und Single–Step relativ einfach realisieren. Auch hier sind wieder alle Dokumente und Beispielprogramme auf der Trenz–Webseite zu finden.

LXO2000 (TEL0001)

- Lattice MachXO2 (4 kLE)
- Arrow–USB–Programmer–2 mit USB–UART–Bridge

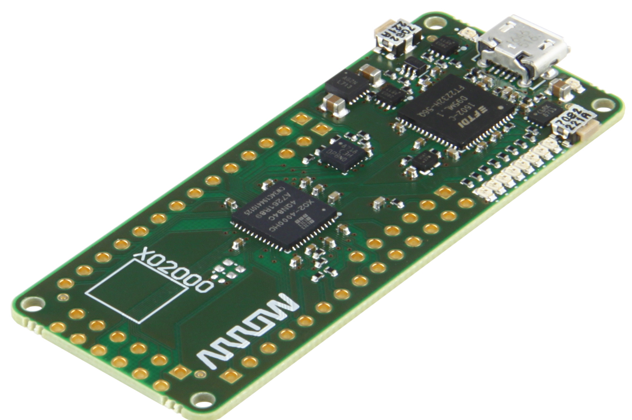


Abbildung 4: LXO2000

Eigentlich nur der Vollständigkeit halber hier das vierte Board basierend auf einem Lattice MACHXO2. Mit nur 4 kLE ist er fast zu klein für eine Prozessor–IP, aber selbst damit kann man 16–Befehle–Cores oder Sequenzer programmieren. Dafür kann man auch die 200 kBit internes RAM nutzen, die auf dem Chip verteilt sind.

Prozessor–IPs

Hier bewege ich mich auf Glatteis, da die Programmierung von FPGAs nicht meine Kernkompetenz ist und ich deshalb wenig Erfahrung mit den diversen IPs habe. Trotzdem hier einige Hinweise auf verfügbare Cores.

Ich denke, dass OpenCores die wichtigste Quelle für freie IPs ist. Unter anderem werden dort ca. 200 Prozessoren wie 6502, 6809, 68HC11, Z80, 8051 oder sogar ein 32–Bit–Forth–Prozessor mit Java–Compiler gelistet.

Es gibt eine lange Liste von Forth–Cores für FPGA. Hier möchte ich nur die vermutlich bekanntesten erwähnen:

- The J1 (H2) Forth CPU (siehe Gameduino)
- C. H. Tings EP16/EP32
- Don Goldings FP1
- Richard E. Haskells FP16 Forth Core
- Bernd Paysans B16
- MPEs RTXcore (RTX–2000 für FPGA)
- Klaus Schleisieks MicroCore

Da man mit den meisten Entwicklungsumgebungen sowohl VHDL als auch Verilog verwenden kann, gibt es bei allen Forth–Cores nur die Limitierungen durch den FPGA oder den angehängten externen Speicher. Viele der Implementierungen sind auf Xilinx–FPGA–Boards getestet, aber vermutlich leicht auf Intel–, Lattice– oder Microchip–FPGAs übertragbar. Da man kaum um die Entwicklungsumgebung der Hersteller herumkommt, muss man sich zuerst einige 100 MByte bis 5 GByte aus dem Internet laden und installieren. Für die Einarbeitung helfen natürlich die vielen Einführungen der Hersteller im Web. Für die erwähnten Boards gibt es jeweils Beispiele, welche die Schnittstellen initialisieren oder sogar schon eine Prozessor–IP implementieren.

Schlusswort

Ich weiß, dass sich dieser Artikel eher nach Werbung für unsere Produkte anhört. Aber ich hoffe, dass es eine Anregung ist, um mit möglichst kleiner Investition die alten Forth–Mikrocontroller wie 6502, 6809, 68HC11 oder Z80 wieder zum Leben zu erwecken oder selbst einen Forth–Prozessor zu implementieren. Die 8 kLE bis 25 kLE des MAX10, Cyclone–10–LP oder SmartFusion2 sind dafür ausreichend und bieten wegen des zusätzlichen externen Flashs und dem SDRAM genügend Speicher für die Emulation der meisten Systeme. Über die vielen Pins sind dazu auch Anbindungen an (PS2–) Tastatur und Display möglich, falls ein UART über die Debug–Schnittstelle nicht ausreicht. Leider kann ich euch die Verwendung der Hersteller–Tools nicht abnehmen, da es nur wenige freie FPGA–Entwicklungsumgebungen gibt und die entsprechenden Compiler am besten auf die Chips angepasst sind.

Ich würde mich über Interesse freuen und, wie auch bei meinen Mikrocontrollern, soweit möglich auch diese Tools für Projekte zur Verfügung stellen. Dazu bitte eine E–Mail an `klaus.kohl-schoepe@arrow.com` (möglichst mit Informationen über die geplante Anwendung).

Links

`https://www.arrow.de/products/max1000/arrow-development-tools`

`https://www.arrow.de/products/cyc1000/arrow-development-tools`

`https://www.arrow.de/products/smf2000/trenz-electronic-gmbh`

`https://www.arrow.de/products/lxo2000/trenz-electronic-gmbh`

`https://shop.trenz-electronic.de/de/Produkte/Trenz-Electronic/MAX1000-Intel-MAX10/`

`https://shop.trenz-electronic.de/de/Produkte/Trenz-Electronic/CYC1000-Intel-Cyclone-10/`

`https://shop.trenz-electronic.de/de/Produkte/Trenz-Electronic/SMF2000-Microsemi-SF2/`

`https://shop.trenz-electronic.de/de/Produkte/Trenz-Electronic/LXO2000-Lattice-XO2-4000/`

`http://opencores.org`

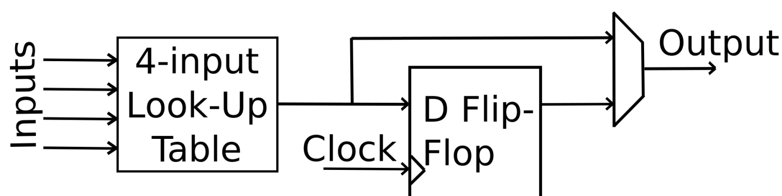


Abbildung 5: Logikblock eines FPGA mit 4-Bit-LUT und optionalem Ausgangs-Flipflop

Amforth wird groß

Matthias Trute

Amforth ist als 16–Bit–Forth auf einem 8–Bit–Controller entstanden. Später kam der MSP430 als 16–Bit–Controller hinzu. Jetzt ist Amforth auf dem Sprung in die 32–Bit–Welt.

32–Bit–Controller

Die Welt bleibt nicht stehen. Daher ist es nicht erstaunlich, wenn die lang geliebten kleinen Controller sich so langsam ins Grau der Geschichte verabschieden. Daher sollte man einen Blick über den Tellerrand wagen und schauen, was es sonst noch so gibt. Hier ist die Forth–Gesellschaft ein unglaublich hilfreicher Partner. Michael hatte mir schon vor Jahren ein Launchpad mit einem LM4F120XL zugeschickt, mutmaßlich ein Fehlgriff. Wir sprachen damals über die MSP430, die als 4–EURO–Forth Furore machen sollten. Trotzdem hat er kein bisschen geschimpft, dass ich den einfach ignoriert habe. Mehr noch: Auf der Forth–Tagung in Essen haben wir kurz über einen neuen Stern am CPU–Himmel namens RISC–V diskutiert und Wolfgang hat es sich nicht nehmen lassen, ein paar Platinen zu bestellen und zu verteilen. Eine davon kam zu mir.

Damit hatte ich zwei 32–Bit–Controller auf Platinen, beide spannend und keine Ahnung, wie man die Dinger zum Leben erweckt. Wie immer hat man den größten Überblick, wenn man auf Schultern von Riesen steht. In diesem Fall ist der Riese Matthias Koch, der schon seit langer Zeit sein Mecrisp auf diversen Controllern implementiert hat. Da er freundlicherweise seinen Code unter der GNU–Lizenz veröffentlicht hat, stand einem ausgiebigen Studium und anschließender Zweitverwertung nichts im Weg. Mecrisp selbst war irgendwie keine Option, warum nur?

Wortbirnen

Der Code von Matthias ist ungewöhnlich. Genau wie sein Sprachwitz. Aber zuerst die Hürden des Handwerks: Wie übersetzt man den Quellcode und wie kommt er zum Controller? Die Übersetzung macht der GNU–Assembler. Der ist mittlerweile in den gängigen Linuxdistributionen für die hier betrachteten CPUs enthalten. Das war vor einigen Jahren noch anders. Die Tools für den Upload waren schon weniger einfach zu bekommen. Die ARM–Controller sind wie Diven, jede will ihr eigenes Tool haben. Da war es ein Glück, dass ich nur einen davon hatte und das Tool `lm4flash` zum Standardlieferumfang von Ubuntu zählt.

Der RISC–V war sehr viel komplizierter. Der hätte gerne Spezialversionen von gleich zwei Tools, dem GNU–Debugger `gbd` und dem `openocd`. Die Uploadroutine von Mecrisp war sehr komplex und vor allem nicht automatisiert. Man musste die beiden Prozesse parallel starten, die dann über Netzwerk (ja, richtiges IP) miteinander reden mussten und dann hoffen, dass alles gut geht. Ein verwertbarer Vorteil, dass die Prozesse für mehrere Zyklen zur Verfügung stehen, war allerdings nicht ersichtlich. Am Ende half ausgiebiges Suchen und Experimentieren, um das Ganze in einem Makefile automatisiert durchführen zu können.

Als das so weit organisiert und es ausreichend war, ein `make && make upload` abzufeuern, um die neueste Version zu testen, fing die Arbeit an, die Spaß macht. Wie macht man aus einem Forth, das nativen Code generiert, dabei eine Vielzahl von Optimierungen durchführt und obendrein auf unbekannten CPUs läuft, ein Forth, das man auch nach Feierabend noch verstehen kann? Man lässt einfach alles weg, was unverständlich ist und hofft, dass es immer noch klappt. Klingt naiv, hat aber fast bis ins Ziel geführt. Die Lernkurve hat dabei irgendwann die Streichkurve erreicht und seitdem gibt es den einen oder anderen Optimierungsfall.

Der erste Schritt war der innere Interpreter. Jedes Indirect–Threaded–Forth (ITC) hat so einen, Mecrisp ist aber keines. Das sind ja nur zwei Register, die push und pop spielen und ansonsten nur als Adresse für das jeweils andere dienen. Sollte man meinen.

Beim ARM sieht das noch einfach aus. Da kommt man eher ins Grübeln, ob das wirklich noch ein RISC–Prozessor ist (ARM stand mal für Acorn–RISC–Machine). Diese Frage kommt auch an anderen Stellen auf.

```
DOCOLON:
    push {FORTHIP}
    mov FORTHIP, FORTHW
DO_NEXT:
    ldr FORTHW, [FORTHIP], #4
DO_EXECUTE:
    ldr r0, [FORTHW], #4
    mov pc, r0
```

Der RISC–V–Assembler kennt leider keine Register–Alias und auch sonst ist dieses System dem RISC–Gedanken deutlich näher, was sich in einem *etwas* komplexeren inneren Interpreter niederschlägt:

```
DOCOLON:
    addi sp, sp, -4
    sw x16, 0(sp)  # push IP
    mv x16,x17     # W->IP
DO_NEXT:
    lw x17, 0(x16) # @IP -> W
    addi x16,x16,4 # INC IP
DO_EXECUTE:
    lw x10, 0(x17)  # @W, address of executable code
    addi x17,x17,4  # INC W, points now to PFA
    jalr zero,x10,0 # jump to code
```

Alle Worte im Mecrisp sind Codeworte. Die komplexeren sehen eher wie eine Folge von Unterprogrammaufrufen aus, durchsetzt von kurzen Codeabschnitten. Genau diese Kategorie Worte sollte das existierende Amforth ohnehin beisteuern, daher mussten sie nur grob gesichtet werden. Verbleiben die Worte, die einfach nur eine kleine Aufgabe erfüllen und sonst nichts machen. Diese Worte waren oft genug nur eine Instruktion lang, eine Beobachtung, die auch schon beim MSP430 zutraf.

Damit war der grobe Weg vorgezeichnet: Von Mecrisp die einfachen Worte (sorry Matthias), von Amforth die komplizierten. Die Entscheidung, wie welches Wort codiert werden sollte, war einfach: Alle Worte, die andere Worte aufrufen, wurden COLON–Worte, der Rest CODE–Worte.

Jede Wortdefinition in Forth hat einen Anfang, den Header. Hier kommen die Wortbirnen, Matthias fängt jede Wortdefinition so an. Ein Schelm, wer bei den Früchten an Köpfe denkt.

```
  Wortbirne Flag_foldable_1|Flag_inline, "1-" @
    ( u -- u-1 )
  subs tos, #1
  bx lr
```

Daraus wurde:

```
CODEWORD "1-", 1MINUS @ ( u -- u-1 )
  subs tos, #1
NEXT
```

Die RISC–V–Variante ist der für den ARM sehr ähnlich

```
CODEWORD "1-", 1MINUS @ ( u -- u-1 )
  addi x3, x3, -1
NEXT
```

Da sich diese Ähnlichkeit durch den ganzen Code zieht, war es vergleichsweise einfach, die Portierung für beide CPUs parallel anzugehen und es trotzdem zum Laufen zu bringen.

Konstantenfaltung und Inline–Definitionen sind nicht mehr. Zu kompliziert. Ein Wort ist einfach nur noch sichtbar, dann heißt es `COLON` bzw. `CODEWORD` oder es ist Immediate (`IMMED`) und damit qua Definition auch sichtbar oder schlicht unsichtbar: `NONAME` oder `HEADLESS`. Das sind alles Macros, die den Code vereinfachen.

Die komplizierten COLON–Worte stehen aus der 16–Bit–Welt schon bereit:

```
HEADER(XT_2DROP,5,"2drop",DOCOLON)
    .dw XT_DROP,XT_DROP,XT_EXIT
```

Daraus wird:

```
COLON "2drop", 2DROP
    .word XT_DROP,XT_DROP,XT_EXIT
```

Damit haben vier CPU–Architekturen fast automatisch den gleichen Code, die Unterschiede beschränken sich auf syntaktische Feinheiten. Damit sind auch Bugfixes für alle vier gleichermaßen einfach umzusetzen. Die Codeworte der Basis sind für sich meist hinreichend einfach, Division und anspruchsvolle Doppelzell–Arithmetik mal außen vor gelassen.

Unterschiede

Auch wenn die Plattformen den gesamten höheren Code teilen, heißt dies nicht, dass sie identisch sind. Offensichtlich ist, dass eine Zelle jetzt doppelt so groß ist. Das ist überraschenderweise nur bei der Länge der Adressen wichtig, die lesen und tippen sich weniger schnell und einfach.

Einstweilen ist der Flashspeicher für das Dictionary read–only. Also richtig read–only und nicht so „einmal–schreiben–erlaubt“, wie bei den MSP430. Das hat einige Konsequenzen. Zunächst gibt es jetzt eine `RAM-WORDLIST`, die die neuen Worte aufnimmt. Daneben gibt es eine Vielzahl von Daten, die man zum Systemstart gerne auf bestimmte Werte setzen möchte, aber später ändern will. Bislang haben eingebaute EEPROMs oder spezielle, per SAVE speicherbare Bereiche diesen Dienst erbracht. Mecrisp hat einen Mechanismus, der beim Start das Dictionary durchgeht und alle Worte herausfindet, die ihre Daten im RAM erwarten und dabei feste Startwerte nutzen. Der laufende Code hat dann keine Kenntnis von den Flash–Originalen und arbeitet direkt mit der RAM–Kopie. Diese Idee habe ich übernommen, die Startverzögerung ist nicht merkbar. Damit kann man sein System auch nicht mehr zerstören, spätestens nach einem `COLD` (oder Reset, wenn die Kommandozeile nicht mehr funktioniert) ist alles zurückgesetzt. Für eigene Turnkey–Aktionen ist das natürlich keine Lösung.

Außerdem leiden die Neuen unter den üblichen Beschwerden der Jugend: Fehler und mangelndes Feintuning. Es knirscht und knackt an noch recht vielen Stellen, wo der AVR–Code einfach nur so drüber hinweglächelt. Er ist ja auch schon Teenager und die haben bekanntlich andere Sorgen als die Krabbelgruppe. . .

Dies und Das

Neue Prozessoren sind ein Abenteuer, der RISC–V ist hier keine Ausnahme. Die Toolchain wurde ja schon erwähnt. Der erste Code war für die binutils 2.28 geschrieben, Ubuntu liefert aber die Version 2.30 aus, und damit ging gar nichts. Die Lösung war, dass die alte Version einige Sprungdistanzen selbst angepasst hat, was die Neue nicht mehr macht. Also musste der relevante Code manuell an die richtige Stelle im Quelltext platziert werden.

ARMs sind grässlich. Die haben so viele Varianten, dass man es eigentlich nicht mehr so genau wissen will. Die Dokumentation ist so umfangreich, dass man sie nicht mehr sehen mag. Also findet man auch nichts, wenn man ein Problem hat. Lange Zeit hat eine Hardware–Exception mit der Nummer 3 den Fortschritt gehemmt. Der innere Interpreter funktionierte dann doch für einige wenige Zyklen, nur um wieder mit einer Exception Nummer 3 auszusteigen. Das ist wohl die ARM–Variante von *Es ist ein Fehler aufgetreten*, denn andere Exceptions traten bislang nicht in Erscheinung, trotz einer beeindruckend langen Liste von möglichen Exceptions. Für das Debugging standen eine RGB–LED und der serielle Port mit

einem einzigen Zeichen zur Verfügung. Schon etwas wenig, aber Bernd hatte die Idee, jedem Wort sein eigenes Zeichen bei einer Terminalausgabe zu geben und so zu schauen, wo es schief läuft. Wirklich wichtig war, dass das Macro, welches das Einzelzeichen ausgab, auch gewartet hat, bis das vorherige Zeichen weg war. Die großen CPUs sind doch deutlich schneller als eine kleine serielle Leitung senden kann. Daher soll es hier verewigt werden:

```
.macro SERIAL_EMIT register
   push {r0}
0: ldr r0, =UARTFR
     ldr r0, [r0]
     ands r0, #TXFF
   bne 0b

   ldr r0, =UARTDR
   str \register, [r0]
   pop {r0}
   .ltorg
.endm
```

Der RISC–V hatte diese Sorgen nicht. Er hat einfach so funktioniert. Schon eine tolle Erfindung.

Dass es hinterher gar nicht so einfach war, den Debug–Code wieder los zu werden, sei nur am Rande erwähnt. Manchmal fragt man sich wirklich, ob Debug–Code wirklich so schlimm ist, dass er auch wieder raus muss.

Dabei haben die ARMs einige sehr interessante Features. So kann man bei praktisch jeder Anweisung im Code angeben, ob man die CPU–Flags (Carry, Overflow, Zero etc.) gesetzt haben möchte oder nicht. Auch sind viele Befehle in der Lage, nur bei Vorliegen von bestimmten Flags ausgeführt zu werden. Spart den Sprung drumherum, wenn es nicht sein soll. Dafür sind CALL und RET nicht auffindbar. Die sehen vollkommen ungewohnt aus:

```
... bl sub ...
sub:     push {lr}  ...  pop {pc}
```

Der Branch–And–Link–Befehl `bl` kopiert die der aktuellen Codeadresse nachfolgende Adresse in ein spezielles Register namens LR, das natürlich auf einem Stack gesichert werden muss, wenn man verschachtelte Aufrufe haben möchte (muss man aber nicht machen, wenn man weiß, was man tut). Das `pop pc` macht dann das, was `ret` bei anderen Systemen tut. Dass `push` und `pop` nicht auf Einzelregistern, sondern auf Listen von Registern agieren, ist so ein „ist das noch RISC?“ Verständnisproblem.

Zum Glück ist das alles in Amforth nicht mehr vorhanden. Worte, die andere Worte aufrufen, sind COLON–Worte und diese anderen hören mit NEXT auf.

Die RISC–V–CPU kennt kein push/pop von Registerinhalten. Da muss man selbst basteln:

```
.macro push register
   addi sp, sp, -4
   sw \register, 0(sp)
 .endm

 .macro pop register
   lw \register, 0(sp)
   addi sp, sp, 4
 .endm
```

Dafür kann man sich das Stackpointer–Register frei wählen und, für den Fall von mehreren Registern, die Stackpointer–Arithmetik optimiert auf einmal machen. Alles in bester RISC–Tradition.

```
.macro pushdouble register1 register2
   addi sp, sp, -8
   sw \register1, 4(sp)
   sw \register2, 0(sp)
.endm
```

Allerdings ist es sehr eigentümlich, dass die CPU keine Flags wie Carry oder Zero kennt. Wie addiert man da zwei doppeltgenaue Zahlen? Die Antwort besteht darin, dass man ein Register nimmt und das zusammen mit einigen bedingten Sprüngen, die direkt Register als Vergleichsparameter nutzen, als Carry–Bit nutzt. Hier merkt man, dass die 32–Bit–Welt wenig Bedarf an doppelt genauen Zahlen hat.

Bei beiden Systemen ist das direkte Laden von Zahlenwerten in Register seltsam, wenngleich nachvollziehbar: Wenn der Opcode nur 32–Bits lang sein darf, kann da kein 32–Bit–Wert zusätzlich hineincodiert werden. Der eine (ARM) legt den Zahlenwert in der Nähe ab (!) und liest relativ zum Program–Counter von dieser Stelle, speichert also nur den Offset zwischen den beiden Adressen ab. Der andere (RISC–V) generiert mehrere Instruktionen, bei denen der Zahlenwert dann ziemlich gewagt zusammengeshiftet wird. Beides macht das Lesen der Listings etwas anspruchsvoller als es bei den kleinen AVRs noch war.

Natürlich darf die Standarderkenntnis nicht fehlen: Faul sein lohnt sich nicht. Eine provisorische Definition zu erstellen, nur damit der Assembler erst mal durchläuft, ist eine schlechte Idee. Man vergisst das sehr schnell und wundert sich später, warum ein Programm es nicht mehr tut. Kann es auch nicht, wenn der Basiscode dann nicht mehr vorhanden ist. Nerven, Zeit oder Aufwand spart man so eher nicht.

Wie geht es weiter?

Was auf jeden Fall interessant werden könnte, ist eine Linuxportierung, also ein ganz normaler Linuxprozess. Es gibt nicht viele Programme unter Linux, die komplett in Assembler geschrieben werden. Jetzt, wo die Hürde des GNU–Linkers, wenngleich noch nicht komplett, genommen ist, aber machbar wirkt, erscheint das nicht mehr ganz so abwegig. Ein Raspberry–Pi ist dem Cortex–M4 schon recht ähnlich. Matthias Koch hat da schon was vorbereitet. ;)

Verweise

1. `http://amforth.sf.net/` Amforth Webseite
2. `http://mecrisp.sf.net/` Mecrisp Webseite
3. `http://forth-ev.de/filemgmt/singlefile.php?lid=602` ARM Monografie der VD

Forth–Gruppen regional

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

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

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

Ruhrgebiet
Carsten Strotmann
`ruhrpott-forth@strotmann.de`
Treffen alle 1–2 Monate im Unperfekthaus Essen
`http://unperfekthaus.de`.
Termine unter: `https://meetup.com/de-DE/Essen-Forth-Meetup`

Gruppengründungen, Kontakte

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

μP–Controller Verleih

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

Spezielle Fachgebiete

Forth-Hardware in VHDL microcore (uCore)	**Klaus Schleisiek** Tel.: **(0 75 45) – 94 97 59 3**$_p$ kschleisiek@freenet.de
KI, Object Oriented Forth, Sicherheitskritische Systeme	**Ulrich Hoffmann** Tel.: **(0 41 03) – 80 48 41** `uho@forth-ev.de`
Forth–Vertrieb volksFORTH ultraFORTH RTX / FG / Super8 KK–FORTH	Ingenieurbüro **Klaus Kohl–Schöpe** Tel.: **(0 82 66)–36 09 862**$_p$

Termine

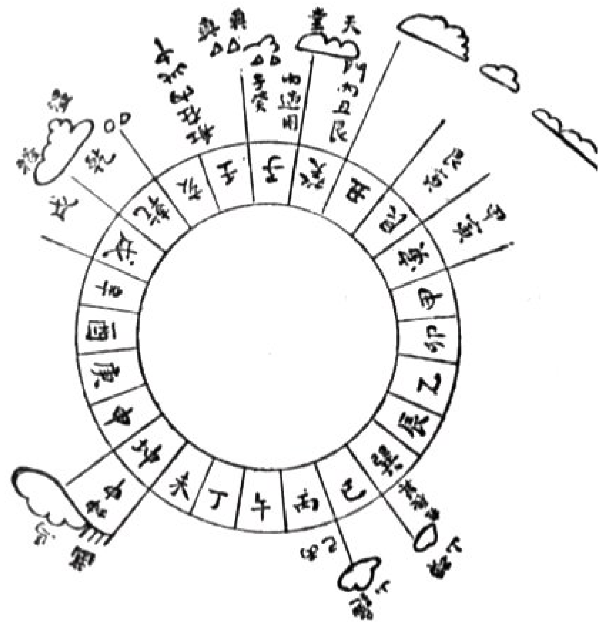
Donnerstags ab 20:00 Uhr
Forth–Chat net2o `forth@bernd` mit dem Key `keysearch kQusJ`, voller Key:
`kQusJzA;7*?t=uy@X}1GWr!+0qqp_Cn176t4(dQ*`

2.–3.2.2019
FOSDEM in Brüssel
`https://fosdem.org/2019/`

11.–14.4.2019
Forth–Tagung in Worms
`https://tagung.forth-ev.de`

27.–28.4.2019
VCFe in München
`https://vcfe.org`

Details zu den Terminen unter `http://forth-ev.de`



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

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

Forth–Tagung in Worms

Ewald und Andrea Rieger

Die Tagung findet vom 11. bis 14. April 2019 im Hotel–Weingut Sandwiese *in der Nibelungen-, Dom-, Luther- und Weinstadt Worms statt. Das Hotel Sandwiese — Fahrweg 19, 67550 Worms–Herrnsheim — liegt am Rande von Worms–Herrnsheim. In unmittelbarer Nähe befindet sich das Herrnsheimer Schloss mit seinem Park im Stil eines englischen Landschaftsgartens.*

Anreise

Worms ist gut über die A 61, Abfahrt Worms–Mitte erreichbar. Bahnreisende erreichen den HBF Worms über Mainz oder Mannheim kommend und steigen dort in den Bus nach Worms–Herrnsheim um.

Anmeldung

Unter dem Link `https://tagung.forth-ev.de/` könnt ihr euch anmelden.