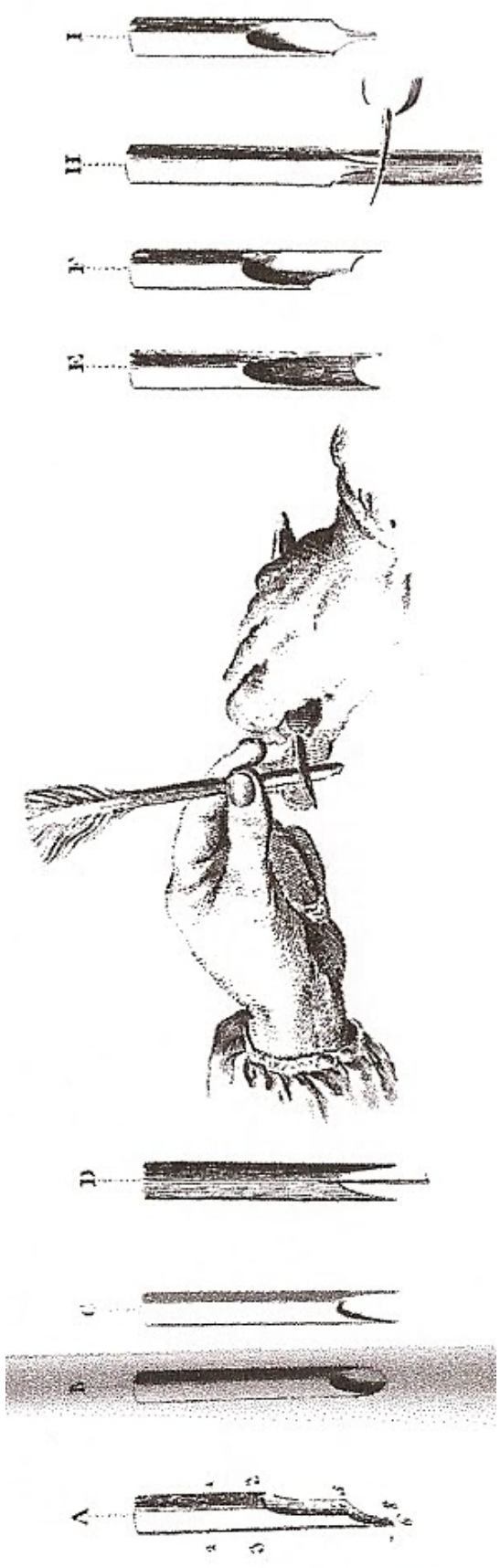
Vierte Dimension

1/2023

Das Forth–Magazin

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



In dieser Ausgabe:

Titelbild: Federkiele schnitzen AUTOR: Skizze bearbeitet von mk

Quelle: Aus dem Internet gefischt

Impressum

Name der Zeitschrift

Vierte Dimension

Herausgeberin

Forth–Gesellschaft e. V.
Postfach 1030
48481 Neuenkirchen
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 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,

unser alljährliches Forthtreffen steht an! Bald ist es endlich wieder soweit. Auf der Rückseite ist erklärt, wie ihr teilnehmen könnt. Ihr habt es bestimmt bemerkt: Der vor Weihnachten im letzten Heft noch anvisierte Termin hat sich verschoben; die Tagung ist in den Mai gerückt.

Reisen ist nicht nötig, wir bleiben beim Videokonferenzformat. Warum? Weil wir es jetzt können. Nach dem Federkiel und reitenden Boten kamen Buchdruck, Briefpost, Schreibmaschine, Keyboard und nun Mikrofon und Kamera — das Bildtelefon hatte im Film von Fritz Langs *Metropolis* seinen wohl ersten Auftritt in der Filmgeschichte, nun machen wir Videokonferenzen.

Die simplifizierten Buchstaben der anfänglichen Terminals sind schon wieder *retro*. Schreiben als Kunst hat sich durchgesetzt. Computerschriften und Handschriften nähern sich wieder an. Und auch die Werkzeuge um Forth herum bleiben nicht stehen. Wirkt Forth in seiner kleinsten Form auf MCUs bisweilen wie ein Federkiel, sind die großen Forth–Systeme weit vorn im Fortschritt dabei. Aber die Tools für die kleinsten holen auf.

Auch Ingolf Pohl hat sich dieses Thema vorgenommen. Wie findet ihr seinen *FEE*? Ist natürlich auf seine Projekte zugeschnitten — „Embedded“ mit Mecrisp. Bin gespannt auf die Resonanz. Und damit man das Mecrisp auch ins Target kriegt, macht das Tool das gleich mit.

Andrew Read führt uns in die Welt der Astronomie und ihre tollen, modernen Möglichkeiten. Ob Charles Moore damals beim Entwickeln von Forth hat kommen sehen, dass man heute von zuhause aus solche professionellen Geräte benutzen kann für die Himmelserforschung?

Natürlich braucht man für solche Teleskop–Ansteuerungen eine handfeste Hardware. Rafael Deliano lässt uns wieder teilhaben an solchen Entwicklungen, die robust sind und angepasst auf den Zweck. Wichtig dabei ist natürlich, dass die Komponenten auch zu haben sind. Hoffentlich bleibt das so. Das letzte Jahr war da nicht so gut — ihr erinnert euch: Lieferengpässe.

Bernd Paysan schließlich beschreibt, wie weit die Darstellung ganz verschiedener Schriften in seinem MINΩΣ2 bereits gediehen ist. Und knüpft damit an seinen Bericht aus dem letzten Heft an. Hier sind wir endgültig weg vom Federkiel — obschon, wer weiß? Vielleicht erlebe ich es noch, dass ein Gforth–Humanoid mir einen handschriftlichen Brief schickt, geschrieben mit selbstgeschnitzter Schwanenflügelfeder.

Und sogar ich hab diesmal etwas Forthiges beigetragen können, denn es ergab sich, SPI erforschen zu dürfen. Dirk Brühl war mein Sponsor. Er hat die Bauteile und Platine bereitgestellt. Vielen Dank dafür.

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
http://fossil.forth-ev.de/vd-2023-01

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

Ulrich Hoffmann Kontakt: Direktorium@Forth-ev.de
Bernd Paysan
Gerald Wodni

Abbildung 1: Chris Hay

Why Forth Language is Important

Chris Hay (Abb. 1) stellte neulich ein zunächst unscheinbar wirkendes Video auf YouTube mit dem Titel: „Introduction to Forth Programming Language — Tutorial for Beginners!“ Doch er lässt Forth nicht so an und für sich im Raum stehen, wie es sonst so oft geschieht. Er zeigt den Nutzen solcher Konzepte wie Forth eindrücklich am Beispiel der Bitcoins.

Daher sei seine Einleitung hier komplett wiedergegeben. Gut gemacht, finde ich.

> „An introduction tutorial to the FORTH programming language and why it's important. If you truly want to understand stack machines, WebAssembly, bitcoin, blockchain or smart contracts then you really need to learn Forth programming.
>
> Although the Forth language is an old programming language, it will truly help you understand stack machines, stack programming and fundamental computer science concepts. It even helps you understand how modern concepts such as bitcoin and smart contracts work. Bitcoin Script which is the smart contract language that powers the bitcoin blockchain is completely built upon on the Forth language. Satoshi Nakamoto was no doubt a Forth programmer with a Forth programming background.
>
> In this video, Chris gives you an introduction to Forth using Gforth and shows you how to quickly get started with the language. We look at arithmetic, stack operations, words and give you enough of an introduction to the language itself.
>
> We then compare the Forth language to Bitcoin Script by looking at the source code of the current version and 0.5.0 release of bitcoin written by Satoshi Nakamoto, getting into the mind of the creator and theorizing on Satoshi Nakamoto's background based on actual code.
>
> We finally look at how close Gforth is to assembly by looking at an old Forth interpreters source code and looking at how close the language is to asssembly.“

Es gibt eine Einteilung in Kapitel, die man gezielt aufrufen kann. So muss man nicht von vorn bis hinten durch den 44–Minuten–Clip. Such dir das raus, was dich besonders interessiert. cas/mk

00:00	why Forth language is important
01:45	installing Gforth
02:18	getting started with Forth programming with Gforth
02:35	introduction to stack machines
03:00	programming stacks with Forth
06:08	performing arithmetic with Forth using postfix
12:08	stack operations (dup, drop, rot, over, nip, swap)
20:48	explaining stack diagrams
23:08	words or functions with Forth
27:53	comparing Bitcoin Script to Forth
32:21	2drop, 2dup, 2swap etc.
34:39	Satoshi Nakamoto was a Forth programmer
36:31	Bitcoin Script commands that were disabled
37:32	comparing Forth to assembly by dissecting eForth source code
39:15	Forth extends itself
40:35	conclusion

`https://www.youtube.com/watch?v=i7Vz6r6p1o4`

Mon bon Forth 0.5.4 (Atari ST)

> „It's a FORTH *Editor*, Interpteter and Compiler for the ATARI computers. Version 68030 (Falcon, TT) and 68000 (STe) . . .“

Die Freunde des Vintage–Computing dürften sich freuen. Guillaume Tello veröffentlichte Ende Januar diesen Jahres die Version 0.5.4 auf seiner Webseite. Sein ganz auf diese Maschinen spezialisiertes Forth kann nun noch mehr:

- M&E modules management (load/save images + effects) + AUDIO modules
- 2D and 3D curves and surfaces
- Dialog with M_PLAYER/MP_STE for videos
- In–line help with help/guide and ST–Guide or HypView
- Allows multitasking with separate pages and stacks

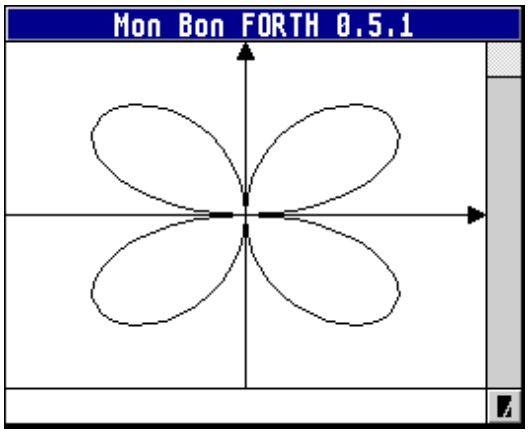


Abbildung 2: Polar curve r(t)

Seine Website zeigt etliche Forth–Beispiele, z. B. die 2D–Maths:
`gr_window` — to define the limits of the axes
`gr_axes` — to draw axes
`gr_grid` — to display a grid
`gr_y(t)`, `gr_xy(t)`, `gr_r(t)` — for the different types of curves (Abb. 2)

Eine 3D–Math ist übrigens auch schon dabei.

Sein Forth ist komfortabel, eben nicht auf fremde Editoren angewiesen und hat viele Hilfen, die den Umgang damit ermöglichen.

Das Forth–Manual, ein PDF mit prallen 219 Seiten, deckt alles ab, zum Forth und zum Betriebssystem der Maschinen selbst auch.

Gratuliere, toll gemacht! cas/mk

`https://gtello.pagesperso-orange.fr/forth_e.htm`

Von Gforth zu Lichen

Wo wir gerade beim Thema „Editieren und Ansehnliches schaffen“ sind:

> „Lichen is the simplest possible CMS[1] for the web that is friendly enough for non–technical users. Comprised of just a few Forth CGI scripts, it is extremely lightweight . . .“

Und was braucht man für Lichen alles?

- A POSIX–compatible environment
- A CGI–capable web server
- Gforth v0.7.3+

Die Lichen–Website selbst ist in Lichen verfasst, also ein Beispiel für die Leichtigkeit, so etwas zu erschaffen — sehenswert elegant, finde ich. cas/mk

[1] Content Management System

Abbildung 3: Screenshot des Lichen Quellcode–Editors

`https://lichen.sensorstation.co/`

busy? oder ready?

Wie sollte man die Logik drehen? Diese Frage stellte sich jüngst, als ich ein externes Flash via SPI erkundete. Wolfgang Strauss, der meinen SPI–Beitrag (weiter hinten in diesem Magazin) sah, meinte, dass die Phrase

```
begin busy? until
```

auf Deutsch übersetzt keinen Sinn machen würde, auch wenn es funktioniere. Was war da passiert in meinem Forthcode? Drehen wir das mal zurück auf den Anfang: Wie war das doch gleich mit dem BUSY–Flag?

Im Datenblatt zum Winbond SPI–Flash steht, was die Bits in dessen *Status Register* bedeuten, besonders das Bit S0:

> 9.1.1 BUSY
>
> BUSY is a read only bit in the status register (S0) that is set to a 1 state when the device is executing a Page Program, Sector Erase, Block Erase, Chip Erase or Write Status Register instruction. During this time the device will ignore further instructions except for the Read Status Register instruction . . . When the Program, Erase or Write Status Register instruction has completed, the BUSY bit will be cleared to a 0 state indicating the device is *ready* for further instructions.

Status	S0
busy	1
ready	0

Tabelle 1: Flash Status–Bit S0

Die höheren Bits haben auch Bedeutungen, aber die blenden wir mal aus.

Bevor man also mit dem Flash was anfangen kann, hole man seinen Status.

```
: status ( -- u ) \ read status register
  csdown 05 >spi spi> csup ;
```

Da liegt der nun auf dem Stack.

Solange S0 gesetzt ist, muss man warten. Deutet man das gesetzte Bit S0 als ein `true flag`, müsste man in Pseudocode formulieren:

```
flag while --> restart loop if flag is true.
```

Also los, das `flag` aus dem Status–Byte extrahiert:

```
: busy? ( -- S0 ) status 1 and 0<> ;
```

Weil `1` nicht unbedingt gleich `true` ist im Forth, testet man auf „verschieden von null“ und bekommt sein `true`.

Die Warteschleife lautet dann in Forth:

```
begin busy? while repeat
```

Oder man testet `0=` und bekommt die umgedrehte Logik. Nun ist S0 = 0 `true` und der Pseudocode lautet:

```
flag until -> restart loop if flag is false.
```

```
: ready? ( -- S0 ) status 1 and 0= ;
```

Mir ist die „positive“ Warteschleife irgendwie sympathischer, oder?

```
begin ready? until
```

Wie der Klassiker:

```
begin key? until ...
```

mk

Schlaglicht: Speicherzugriffe mit verschiedener Wortbreite

Die Welt der 16–Bit–Forth–Systeme war einfach. Um auf Bytes oder Zeichen zuzugreifen, gab es `C@` und `C!`, um Zellen — damals 16 Bit — zu bearbeiten, `@` und `!`. Aber die Zeit der 16–Bit–Systeme ist vorbei und längst gibt es 32– und sogar 64–Bit–Systeme. Wie geht man hier mit den 16–Bit– und 32–Bit–Worten um? Eine Möglichkeit ist die Definition zahlreicher weiterer Speicher–Operatoren. Zumindest auf passende Namen sollte man sich einigen können. Hier eine Übersicht der bisherigen Namensgebungen.

Speicheroperatoren in 32–Bit–Systemen

Auf einem 32–Bit–System hat man typisch drei Wortbreiten zu bearbeiten: Bytes (8 Bit), Halbworte (16 Bit) und Langworte (32 Bit).

8–Bit–Bytes

Dabei erfolgt der klassische Zugriff mit `c@` und `c!` und die *Bytes* werden als vorzeichenlos angesehen, daher findet keine Vorzeichenerweiterung statt.[2]

16–Bit–(Halb–)Worte

Solche Zugriffe sind uneinheitlich benannt, aber es hat sich eingebürgert, diese Adressbitbreite *Word* zu nennen und dafür Operatoren mit `W` als Prefix zu verwenden. (Halb–)Worte werden als vorzeichenlos oder vorzeichenbehaftet angesehen. Es gibt daher lesende Zugriffe mit oder ohne Vorzeichenerweiterung; Tab. 2 zeigt einige Beispiele. Andere machen es anders.

Forth	unsigned fetch	signed fetch	store
Open Boot	w@	<w@	w!
Gforth	uw@	sw@	w!
Swiftforth	w@	w@s	w!
VFX–Forth	w@	w@s	w!

Tabelle 2: Einige 32–Bit–Systeme und ihre Halbwort–Speicheroperatoren

32–Bit–(Lang–)Worte

64–Bit–Systeme bieten auch 32–Bit–Zugriffe an, aber auch diese Zugriffe sind uneinheitlich benannt. Doch es hat sich eingebürgert, sie als *Langwort* zu bezeichnen und bei Operatoren für 32 Bit das L–Prefix zu verwenden. Langworte werden als vorzeichenlos oder vorzeichenbehaftet angesehen.

Forth	unsigned fetch	singned fetch	store
Open Boot	l@	l@	l!
Gforth	ul@	sl@	l!
Swiftforth	l@	l@s	l!
VFX–Forth	@	(kein Symbol)	!

Tabelle 3: 32–Bit– und 64–Bit–Systeme und ihre 32–Bit–Speicheroperatoren

64–Bit–Worte

Die 64–Bit–Systeme greifen natürlich auch 64–bittig auf den Speicher zu. Diese Operatoren haben das Prefix X. Doch Speicherzellen der (Daten–)Stack–Breite werden typisch nur mit `@` und `!` *ohne* Prefix bearbeitet. Diese prefixlosen *fetch* und *store* sind dann gleichzeitig die passenden Aliase zu `w@ w!` (16–Bit–System), `l@ l!` (32–Bit–System) oder `x@ x!` (64–Bit–System).

[2] VfX [2] hat allerdings den Operator `c@s` für Vorzeichenerweiterung.

Ausblick

Die Speicheroperatoren sind in der aktuellen Standardisierungs–Diskussion.

Diesen Zoo von Operatoren zu verwenden, ist höchst unerfreulich.

In seedForth (32 Bit auf PCs) gibt es keine Halb–Wort–Unterstützung. Es gibt nur `c@ c!` (vorzeichenlos) und `@ !` (32 Bit vorzeichenlos oder behaftet, aber irrelevant). Da muss man aber auch keinen Speicher sparen.

Die Forth–Community geht in die Richtung *Values* und *Value Flavoured Structs* (Strukturen, deren Felder wie Values sind, siehe unten). Für kleinere Speichergrößen hat man dann `cValue` und `uwValue swValue`, `ulValue uxValue`, usw., wenn überhaupt.

Werte werden vom `value` passend (mit oder ohne Vorzeichenerweiterung) auf den Stack gelegt. Schreibender Zugriff erfolgt immer über `TO`, das natürlich intern den passendem Operator verwendet.

Der Anwendungsprogrammierer wird dadurch entlastet und muss sich nicht dauernd die richtigen Operatoren heraussuchen. Die Les– und Wartbarkeit wird erhöht.

In einer prototypischen Implementierung von Value Flavoured Structs würde ein Beispiel–Struct so definiert werden:

```
0
int8: u8
sint8: s8
int16: u16
sint16: s16
int32: u32
Constant myStruct
Create S myStruct allot
```

`int8:` usw. sind definierende Worte für Felder passender Größe (und passendem Typs).

Die Feldnamen (hier `u8 s8` ...) führen dann die Offset–Berechnung und das Lesen des Wertes durch: z. B. liest

```
S s16
```

das Feld `s16` mit Vorzeichenerweiterung (weil vom Typ `sint16`) aus und legt den Wert auf den Stack.

Gesetzt werden kann es z. B. mit

```
-42 S TO s16
```

Dabei werden die höherwertigen Bits oberhalb Bit 15 abgeschnitten; bei allen anderen Feldern analog.

Und gibt es auch `SEX`?[3]

Das muss dann eine 8–Bit– und 16–Bit–Vorzeichenerweiterung machen können, also `cSEX` und `wSEX`, oder wie? Oder man ignoriert den Speicherbedarf und speichert gleich alles in 32 Bit. Und ist das für ein kleines System tragbar?

Solche Fragen lösen sich erst in Luft auf, wenn man objekt–orientiert arbeitet. *Values* und *Value Flavoured Structs* sind ein Zwischenweg, brauchen aber auch schon ein polymorphes `TO`. Das wäre aber vergleichbar einfach zu realisieren.

Bei Value Flavoured Structs gibt es sonst keine weiteren Methoden (vielleicht `+TO` und `ADDR`, wenn man unbedingt möchte).

Objekte sind die konsequente Weiterführung.

Doch für eine Standardisierung bräuchten wir die von der Forth–Community getragene *Best Practice*. Die ist aber nicht zu sehen. Wie kann Sie aussehen?

Ulrich Hoffmann

ESP32forthStation

Zoom, 10. Dezember 2022 — dort hab ich die *ESP32forthStation* vorgestellt. Es ist ein eigenständiger Singleboard–Forth–Computer mit WLAN–Netzwerkfähigkeiten. Das Gerät verfügt über Anschlüsse für eine Tastatur (PS/2), einen Videomonitor (VGA), kommuniziert über USB oder WLAN und kann als vollständige Entwicklungsplattform für Experimente genutzt werden.

Basierend auf Open–Source–Projekten beherbergt das kleine *TTGO VGA32 Board* von *LilyGo* die komplette ESP32forthStation. Und das ESP32forth darin arbeitet nun mit den *FABGL*– und *ESP32forth–Bibliotheken* von Fabrizio Di Vittorio und Brad Nelson zusammen. Der Code ist unter den Bedingungen der GNU GENERAL PUBLIC LICENSE lizenziert.

Die ESP32forthStation kann über die *Arduino IDE* geflasht werden.

Ulrich Hoffmann

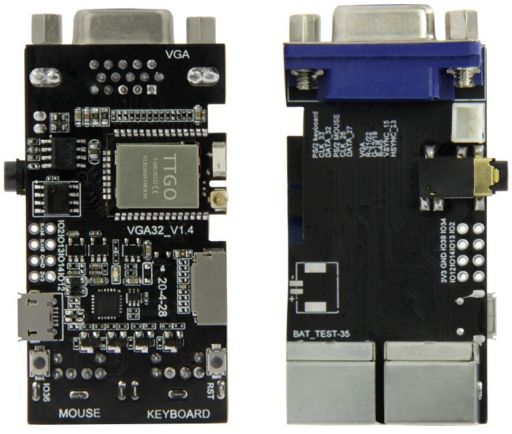


Abbildung 4: TTGO VGA32 Board von LilyGo

Weitere Informationen sowie Kontakt zum Autor findet ihr auf der Webseite seines Projekts.

`https://github.com/uho/ESP32forthStation`

[3] SignExtend (vom 6809 übernommen).

FEE — Forth Enhanced Editor

Ingolf Pohl

In der letzten Ausgabe (4d2022–04) hat Wolfgang Strauss *über das Projekt Feuerstein berichtet und dort auch meinen Editor FEE (Forth Enhanced Editor) erwähnt. Ich möchte hier beschreiben, wie es zu der Idee kam, ein eigenes Werkzeug für die Kommunikation mit einem Mikrocontroller–Target zu entwickeln und wie sich mit dem Tool produktiv arbeiten lässt.*

Es begann mit meinen Fragen an die Forther

Ich fing an, Forth zu programmieren, weil es interaktives Programmieren eines Zielsystems ermöglicht. Man kommuniziert mit dem Compiler im Zielsystem über ein Terminal — Moment mal, über ein Terminal? Was ist mit dem Code, den man gerade zum Compiler geschickt hat, wo ist der hin? Der ist weg? Das kann nicht sein! Ich fragte ein paar Forth–Programmierer im Projekt Feuerstein und bekam Antworten, wie: „Was ist dein Problem? Man schreibt doch in seinem Lieblingseditor Emacs und kopiert dann den Code ins Terminalfenster.“ Kopieren? Vom Editor ins Terminal? Wohlmöglich auch noch zeilenweise, weil das Timing nicht stimmt? Nein, das hätte ich nicht erwartet ...

Vorgeschichte

Ich bin schon durch eine andere stackorientierte und interaktive Programmiersprache bzw. Programmierumgebung verwöhnt — Fifth. Ja, man soll es nicht glauben, in den 80ern hat ein alter Professor von mir eine sehr Forth–ähnliche Programmiersprache entwickelt.[1] Sie diente auch zur Programmierung von heterogenen Multiprozessorsystemen (Controller, DSPs, Transputer etc.) für Fächer–Echolote. Das Ding lief auf dem PC unter DOS; ein System für X86, C167 und ADSP2181 war nicht mal 160 KB groß und via seriellem Interface ans Zielsystem angeschlossen. Besonders war, dass der Crosscompiler durch seinen integrierten Editor gesteuert wurde. Das Programmieren war ganz einfach und interaktiv:

1. Ein Stück Source–Code schreiben.
2. Zum Ausprobieren auf den Probieren–Knopf drücken: Der Compiler compiliert den Code, überträgt ihn an das Zielsystem und führt ihn dort aus, Ausgaben werden im Editorfenster angezeigt.
3. Der Source–Code bleibt im Editorfenster erhalten, man kann ihn verbessern, ändern, ablegen oder man entscheidet, er ist gut so.
4. Ist der Sourcecode fertig, drückt man den Compilieren–Knopf: Der Compiler compiliert, legt den Code im Zielsystem ab und ergänzt das Wörterbuch im Compiler. Die neuen Wörter können nun von neuem Code oder vom PC aufgerufen werden. Der Sourcecode wird auf den Stapel (worauf auch sonst?) mit compiliertem Sourcecode gelegt.

Auch die Sourcecodeverwaltung war eigentlich ganz einfach — einzig das eine kleine DOS–Fenster war hinderlich. Es gab drei Textbereiche, von denen man leider immer nur einen sah:

Bereich A: Fertiger Sourcecode — was man compiliert hatte und gut war

Bereich B: Sourcecode in Arbeit — was man gerade schreibt und ausprobiert

Bereich C: Zukünftiger Code — noch nicht compilierbarer Code, Testcode

So etwas hätte ich auch gerne für Forth ...

Ein interaktiver Editor muss her

Ich hätte gerne so einen Editor für Forth–Quellcode, mit mindestens drei Textbereichen, die auch in einem normalen Editor bearbeitbar sind:

```
--- Textanfang ---------------------
  reifer, schon compilierter Code
--- Textgrenze ---------------------
  gerade zu bearbeitender Code
--- Textgrenze ---------------------
  noch nicht zu compilierender Code
--- Textende -----------------------
```

- Einen Knopf zum Probieren: Aktueller Text wird zum Zielsystem gesendet
- Einen Knopf zum Compilieren: Aktueller Text wird zum Zielsystem gesendet und anschließend in den oberen Bereich geschoben

Der Sourcecode soll in einem Editorfenster erstellt und von dort sehr einfach in das Zielsystem gesendet werden können. Ein– und Ausgaben zur Steuerung des Programms sollen in einem Terminalfenster erfolgen. Es soll ein Editor mit zusätzlichem Terminalfenster sein, nicht ein Terminal mit Editorfenster!

Schnell mal was in Python frickeln

Ich bin nicht der PC–affine Programmierer. Es muss also was her, was mir den PC und seine Fenster einfach erschließt. Toll wäre auch, wenn jedes Feuerstein–Mitglied den Editor unter seiner gerade laufenden Umgebung benutzen könnte. So kam ich zu Python 3 mit `tkinter` und

[1] Fritz Mayer–Lindenberg, `https://www.researchgate.net/profile/Fritz-Mayer-Lindenberg`

`pyserial`. Ein Textfenster mit fertigen Editierfunktionen ist schnell aufgespannt. Die serielle Schnittstelle direkt, via USB oder gar über Telnet kann man so auch leicht ansprechen. Ich hätte es ja gerne in Forth gemacht, habe aber kein für mich passendes System gefunden.

Der Editor funktioniert nach dem oben beschriebenen Prinzip. Vom Terminal aus wird der Editor aufgerufen und erscheint in einem eigenen Fenster. Im Editorfenster kann Text, der zwischen zwei Textgrenzen (Stoppern) liegt, auf die gewünschten zwei Arten an das Zielsystem gesendet werden:

- `ctrl+enter` zum Probieren
- `alt+enter` zum finalen Compilieren

Als zu versendender Quellcode gilt derjenige Textbereich zwischen zwei Stoppern, in dem der Cursor gerade platziert ist.

Natürlich kann man mit nur einem oder auch beliebig vielen Textbereichen arbeiten, Textanfang und Textende sind dabei auch Textgrenzen.

Als Terminalfenster für die Ein–/Ausgabe des Zielsystems funktioniert das Terminalfenster, aus dem der Editor gestartet wurde.

Arbeiten mit dem Editor FEE

Ich persönlich arbeite mit FEE und Mecrisp auf einem GD32VF103–Board wie folgt und teile mir den Sourcecode per Stopper in mindestens 4 Bereiche, eher mehr. Dabei nutze ich die Möglichkeit von Mecrisp, wahlweise ins Flash oder ins RAM zu compilieren:

Bereich 1: Kommentar und `eraseflash` auf Vorrat

Bereich 2: Code großen Vertrauens — für's Flash

Bereich 3: Code geringen Vertrauens — für's RAM

Bereich 4: Code, an dem gearbeitet wird — ins RAM

Bereiche >4: Ideen, Testfunktionen, Entwürfe ...

Bei einem neuen System oder bei total vermurkstem Flash führe ich einfach die Bereiche 1 bis 3 nacheinander durch „Probieren“ aus und habe schnell meinen Entwicklungszustand wiederhergestellt.

Das `eraseflash` steht im Bereich 1, weil Mecrisp dabei über Reset geht und kein „ok.“ liefert. Sonst könnte man alles in einem Rutsch ausführen.

Im Bereich 2 mit dem „Code großen Vertrauens“ stehen oft nur ein paar `include`–Anweisungen, die lang Erprobtes aus ausgelagerten Dateien ins Flash laden.

Der Bereich 3 mit dem „Code geringen Vertrauens“ lädt ins RAM.[2] Dieser Code ist entweder noch nicht reif für das Flash oder wird nur für die Entwicklung benutzt.

Im Bereich 4 liegt bei mir der aktuelle Arbeitsbereich — der kann natürlich auch in einem anderen Bereich sein, je nach Entwicklerdisziplin. Der Code in diesem Block wird editiert, probiert, weiter editiert, erneut probiert ... bis er fertig ist. „Probieren“ geht idealerweise einfach per Hotkey `ctrl+enter`. Und wenn der Code dann fertig ist, verschiebe ich ihn mit einer finalen Compilierung per Hotkey `alt+enter` in den Textbereich darüber.

Beim Probieren benutze ich hier gerne zusätzlich `forget` oder `del` von Matthias Koch. Im Beispiel benutze ich die `del`–Methode, dabei wird das zuletzt compilierte Wort aus dem Dictionary gelöscht und so verhindert, dass beim wiederholten „Probieren“ das Dictionary immer größer wird und schließlich überläuft.

Höhere Bereiche benutze ich als Lager für Testcode, Ideen, alternative Ansätze, Kommentare. Oft entwerfe ich dort schon den Top–Down–Code, während im Bereich 4 dessen Gegenstücke Bottom–Up wachsen.

Wenn ich fertig bin, habe ich einen großen Textblock, der in einem Durchgang in die finalen Speicherbereiche des Zielsystems compiliert wird. Den kann ich dann als Quelltext zusammen mit den `include`–Dateien ablegen und aufheben.

Abb. 1 zeigt ein kurzes Beispiel mit ca. 40 Zeilen Code, wie er tatsächlich bei mir während der Arbeit aussehen kann.

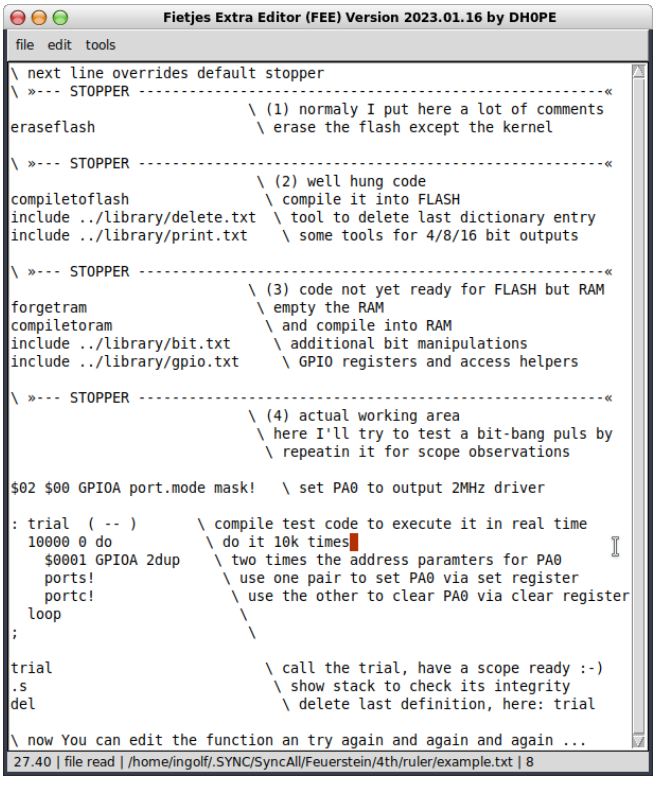


Abbildung 1: Beispiel für alle Aspekte des interaktiven Arbeitens

Die Bereiche sind zur Verdeutlichung mit `(1)` bis `(4)` markiert. Ich tüddel im Sourcecode des Wortes `trial` und probiere es mit `ctrl+enter`. Das mache ich solange, bis ich zufrieden bin. Durch `del` bleibt das Dictionary beim

[2] Wahlweise ins Flash oder ins RAM compilieren zu können, ist eine besondere Eigenschaft vom Mecrisp–Forth.

wiederholten Probieren ohne Überlauf. Zum finalen Compilieren werden die Hilfszeilen mit den Vorbereitungen entfernt — hier das Konfigurieren des Ports, der Funktionsaufruf, die Stackinspektion und die `del`–Anweisung.

Was kann der Editor sonst noch?

In kurzer Zeit habe ich das System mit ein paar nützlichen Funktionen ausgestattet.

- Es gibt ein `include`, um externen Sourcecode zu laden.
- Beim Übertragen können optional Kommentare und leere Zeilen ausgelassen werden.
- Text–Stopper lassen sich per Hotkey einfügen.
- Ein Hardware–Reset über die DTR–Leitung[3] kann per Hotkey ausgelöst werden.
- Übertragung einer einzelnen Zeile kann per Hotkey initiiert werden.
- Die Übertragung ist auf das Mecrisp abgestimmt.
- Es gibt eine magische Textblock–zu–Datei–Konvertierung mit `include`–Einfügen.
- „Leere“ GD32VF103 oder STM32F103, die noch kein Mecrisp enthalten, können über die serielle Schnittstelle das Forth aufgespielt bekommen.
- Die Baudrate zur Kommunikation kann durch ein Sonderwort im Kommentar während des Compilierens umgeschaltet werden.
- Der Editor speichert bei kritischen Operationen eine Kopie mit Zeitstempel in das Unterverzeichnis `./sav/`
- Optional kann man die Ausgaben des Zielsystems in ein Logfile schreiben lassen.
- Neuerdings gibt es auch eine Undo/Redo–Funktion.

Im Anhang findet ihr eine Liste der Funktionen und zugeordneten Hotkeys. Die funktionieren übrigens nur im aktiven Editorfenster und stören daher anderswo nicht.

Wofür steht denn FEE überhaupt?

Ursprünglich steht das TLA (Three Letter Acronym) FEE für „Forth Enhanced Editor“. Da steckt aber kein Spaß drin, das ist viel zu förmlich. Bei Feuerstein dachten wir, dass der oder die Fietje so ein Feuersteinchen ist, welches sich mit dem Forth befasst. Deshalb kann das F auch für Fietjes oder Feuersteins stehen. Das zweite E steht dann für erweitert, exzellent, einzigartig, eigenartig, elegant, erstaunlich, elaboriert usw. Das letzte E bleibt für Editor — Environment würde ich es ohne eingebauten Compiler nicht nennen.

Bezugsquelle

Die Projektseite des/der FEE findet ihr auf
`https://forth-ev.de/wiki/projects:fee:start`

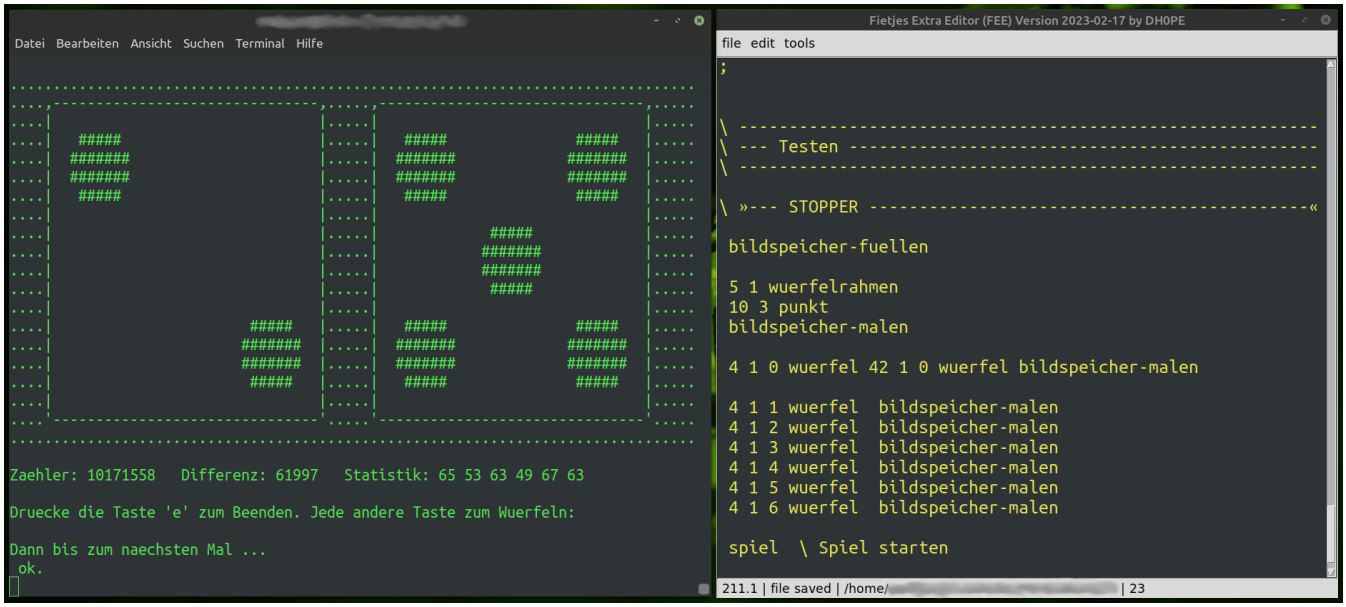


Abbildung 2: FEE in Aktion, Farbschema „Dark Mode“. Seite an Seite das Terminal–Fenster (links) und das Editor–Fenster (rechts). Im Editor ist der Testcode für ein Würfelspiel zu sehen. Jede Zeile kann einzeln mit `shift+enter` an das Forth–System gesendet werden. Übrigens: FEE kann durch Aufrufparameter individualisiert werden. Ein Muster–Shell–Skript findet ihr in Anhang E. Wer hier die Papierausgabe liest: Schau mal in die PDF–Version des Magazins, die ist auch innen farbig. :-)

[3] Data Terminal Ready (Handshake–Signal der seriellen Schnittstelle)

Anhänge

Anhang A: Dateifunktionen, auch im Menü „File“

`ctrl+o` = Datei öffnen

`ctrl+r` = Datei nachladen, Editorinhalt überschreiben

`ctrl+s` = Datei speichern

`ctrl+shift+s` = Datei speichern als

`ctrl+shift+w` = Log–Datei an–/ausschalten

`ctrl+shift+q` = Editor verlassen

Anhang B: Editorfunktionen, auch im Menü „Edit“

`ctrl+z` = Undo

`ctrl+shift+z` = Redo

`ctrl+x` = Auschneiden

`ctrl+c` = Kopieren

`ctrl+v` = Einfügen

`ctrl+l` = Zeile ausschneiden

`alt+l` = Zeile kopieren

`ctrl+v` = Zeile einfügen

`shift+enter` = Zeile probieren — aktuelle Zeile an das Zielsystem senden

`ctrl+enter` = Block probieren — aktuellen Textblock ans Zielsystem senden

`alt+enter` = Block abschicken — aktuellen Textblock ans Zielsystem senden und schieben

`ctrl+j` = Eine Stopper–Zeile einfügen

`alt+j` = Aktuellen Textblock auslagern, durch `include` ersetzen

`ctrl+shift+m` = Leerzeichen anzeigen/verbergen

Anhang C: Hilfsfunktionen, auch im Menü „Tools“

`ctrl+q` = Terminalfenster leeren

`alt+q` = Hardware–Reset des Zielsystems

`ohne hotkey` : Zielsystem mit Binärfile flashen

Anhang D: Einige wichtige Hotkeys von tkinter

`ctrl+x` = Löschen/verschieben, im Clipboard speichern

`ctrl+c` = Kopieren

`ctrl+v` = Einfügen

`ctrl+y` = Einfügen

`ctrl+b` = Cursor ein Zeichen zurückbewegen

`ctrl+f` = Cursor ein Zeichen vorbewegen

`ctrl+p` = Cursor eine Zeile zurückbewegen

`ctrl+n` = Cursor eine Zeile vorbewegen

`ctrl+a` = Cursor an Zeilenanfang bewegen

`ctrl+e` = Cursor ans Zeilenende bewegen

`ctrl+i` = Tabulator einfügen

`ctrl+d` = Löschen an Cursorposition (Delete)

`ctrl+h` = Löschen links vom Cursor (Backspace)

`ctrl+k` = Rest der Zeile löschen

`ctrl+t` = Tauschen der zwei Zeichen links vom Cursor

Anhang E: Shell–Skript für den Aufruf von FEE

```
#!/bin/bash

args=( # Path of FEE Python file
       ~/Feuerstein/work/FEE/FEEv20.py
       # The next two parameters are mandantory
       /dev/ttyUSB0      # Serial device
       115200            # Baudrate
       # The next parameters are optional. If given,
       # they override the defaults.
       # At the end of this file there is a list of
       # supported parameters
       fn='Ubuntu Mono'  # A monospaced font
       fs=16             # Fontsize
       fc=#e3e355        # Font color (amber)
       bc=#2e3436        # Background color (dark grey)
       ac=#ff5500        # Active color (orange)
       cc=#ff5500        # Cursor color (orange)
     )
python3 "${args[@]}"

# Complete list of additional command line parameters,
# which can be added after the 2nd parameter:
#
#   DTRN           : inverts the DTR-Reset from
#                      active low (default) to high
#   RTSN           : inverts the RTS-Boot0 from
#                      active high (default) to low
#   RTSCTS         : activates HW handshake via RTS/CTS
#   NOSAVE         : switches off backup copy
#   FASTER         : skip comments after a backslash
#                      on transfer to target
#   fn='font name' : replaces the default font 'mono'
#                      by 'font name'
#   fs=xx          : set font size, for example:
#                      fs=10 is default
#   fc=#rrggbb     : set font color, default is
#                      #d5d2c8 (light grey)
#   bc=#rrggbb     : set background color, default is
#                      #343d46 (dark grey)
#   ac=#rrggbb     : set active color, default is
#                      #b43104 (red)
#   cc=#rrggbb     : set cursor color, default is
#                      #b43104 (red)
#   cw=xx          : set cursor width, default is
#                       0 for block cursor
#   co=xxxx        : set cursor on time in ms,
#                       default is 1000
#   cf=xxxx        : set cursor off time in ms,
#                       default is 0
#   is='string'    : overrides the default
#                       include string '\ include'
#                       default strings: 'include',
#                       '#include','\ include'
# Caution: strings with blanks have to be put in ' '
# Colors can be named according tkinter: #rrggbb or
#   a color name

# end of file
```

Forth flashen

Ingolf Pohl, Wolfgang Strauß

Im Artikel über FEE (siehe Artikel „FEE — Forth Enhanced Editor“ in diesem Heft) wurde erwähnt, dass man ganz einfach ein Forth–Laufzeitsystem, wie z. B. Mecrisp, aus FEE heraus in das unbeschriebene Zielsystem laden kann. Wir möchten den Vorgang hier am Beispiel des Fietje–Miniboards beschreiben, welches mit einem GD32VF103 läuft. Das Verfahren funktioniert natürlich auch mit anderen GD32VF103–Boards wie etwa dem Longan Nano. Chips aus der Serie STM32F103 werden ebenfalls unterstützt.

Grundlagen

Viele moderne Mikrocontroller haben ihren Programmspeicher gleich mit auf dem Chip. Die gängige Speichertechnologie ist Flash[1]. Der Hersteller liefert seine Bausteine initialisiert, d. h., alle Bits des Flashs sind auf „1“ gesetzt. Bevor der Forther in der gewohnten interaktiven Weise mit dem Mikrocontroller arbeiten kann, muss eine Binärdatei, die ein passendes Forthsystem enthält, in den Baustein „geflasht“ werden. Beim hier betrachteten GD32VF103 der Firma GigaDevice kann das Flashen einer Binärdatei auf unterschiedliche Weise erfolgen:

- Über die JTAG–Schnittstelle (JTAG: Joint Test Action Group). Dafür wird spezielle Debug–Hardware benötigt
- Über SWD (Serial Wire Debug), benötigt ebenfalls spezielle Debug–Hardware
- Mittels Boot–Loader über das chipinterne USB–Hardware–Modul per DFU–Protokoll (Device Firmware Upgrade)
- Mittels Boot–Loader über die chipinterne serielle Schnittstelle (UART)

Der Boot–Loader–Modus per UART bietet sich hier an, denn den dazu benötigten USB–Seriell–Wandler verwenden wir nach dem Flashen eh für die Kommunikation mit dem Forth.

Start–Varianten

Der GD32VF103 kennt drei Start–Varianten, von denen eine durch die Pegel von zwei speziellen Pins (BOOT0, BOOT1) nach einem Reset des Chips ausgewählt wird. Tab. 1 listet die Varianten auf.

Boot–Pin–Level		Start–
BOOT0	BOOT1	Variante
`low`	`egal`	Flash
`high`	`low`	Boot–Loader
`high`	`high`	RAM

Tabelle 1: Start–Optionen

Es folgt nun die genauere Beschreibung der einzelnen Varianten:

- *Flash:* Dieses ist später der Normalbetrieb. Hier startet die CPU des Mikrocontrollers die Abarbeitung des Programms im Flash–Speicher. Dazu muss natürlich vorher eine passende Firmware (Forth) im Flash abgelegt (geflasht) worden sein.
- *Boot–Loader:* Diese Variante ist genau das, was wir für das Flashen brauchen. Sie startet den sogenannten Boot–Loader. Der Boot–Loader ist Software, die der Hersteller des Chips in einem speziellen Speicherbereich untergebracht hat. Diese Software ist schreibgeschützt, kann also nicht versehentlich oder absichtlich überschrieben werden.
- *RAM:* Hier würde die CPU Code aus dem RAM starten. Diese Variante ist für uns nicht weiter nützlich, da der Inhalt des RAMs nach einem Kaltstart[2] zufällig ist. Das gäbe einen schönen Crash.

Der Boot–Loader

Ok, jetzt sind wir schon ein ganzes Stück weiter. Wir wollen also den Boot–Loader verwenden, um den Chip zu überreden, eine Binärdatei über eine normale serielle Schnittstelle entgegenzunehmen und in sein Flash zu schreiben. Aber wie macht der Boot–Loader das genau?

Der Chip testet direkt nach einem Reset seine Pins BOOT0 und BOOT1. Wenn sich Pegel wie in Tab. 1, Zeile 2 ergeben, beginnt die Ausführung des Boot–Loader–Programms. Aber das hatten wir ja schon.

Als Erstes wird dann geprüft, auf welchem „Kanal“ Daten eintreffen. Kanäle können der USB–Port oder auch eine serielle Schnittstelle (UART) sein. Hier soll davon ausgegangen werden, dass Daten über einen UART eintreffen. Nach dem Empfang des ersten Zeichens wird dessen zeitliche Länge ermittelt und damit die passende Baudrate ermittelt und eingestellt. Jetzt ist alles bereit für eine interaktive Kommunikation mit der Gegenstelle, in unserem Falle FEE.

Der Boot–Loader hat u. a. Funktionen zum Löschen, Schreiben und Lesen des Flashs und zum Ermitteln der Chip–ID und Protokoll–Version. Damit lässt sich komfortabel arbeiten.

Wer mehr Details möchte, schaue sich die *Application Note* „AN3155“ an, zu finden auf `https://www.st.com`

[1] Flash: Speicher, der die Informationen auch nach dem Abschalten der Versorgungsspannung nicht verliert.

[2] Kaltstart: Anlaufen des Systems nach Einschalten der Versorgungsspannung.

Stückliste

Für das erste Laden eines Forthsystems mit FEE und danach natürlich auch für das interaktive Entwickeln braucht man

- einen PC mit installiertem Python 3, FEE (Forth Enhanced Editor)
- einen USB–Seriell–Adapter (TTL–Pegel und VCC auf 3,3 V); es empfiehlt sich ein Baustein mit den Handshakeleitungen DTR und RTS. Damit kann FEE dann den Flashvorgang automatisch ausführen. Ein „normaler“ Adapter, der nur RxD und TxD hat, geht aber auch.
- ein Zielsystem mit dem Mikrocontroller GD32VF103
- eine Binärdatei mit einem Forth–System (Mecrisp–Quintus) für den GD32VF103

Anschluss an das Fietje–Board

Für die bequeme Boot–Mode–Umschaltung braucht man die Verbindungen laut Tab. 2 zwischen USB–Seriell–Wandler und Fietje–Miniboard:

USB–Wandler	Daten–Richtung	Fietje–Miniboard
GND	—	GND
VCC (3,3 V)	—	3V3 (3,3 V)
TxD	→	RxD
RxD	←	TxD
DTR	→	RSTN
RTS	→	BOOT0

Tabelle 2: Verkabelung

Abb. 1 zeigt den Aufbau bildlich:

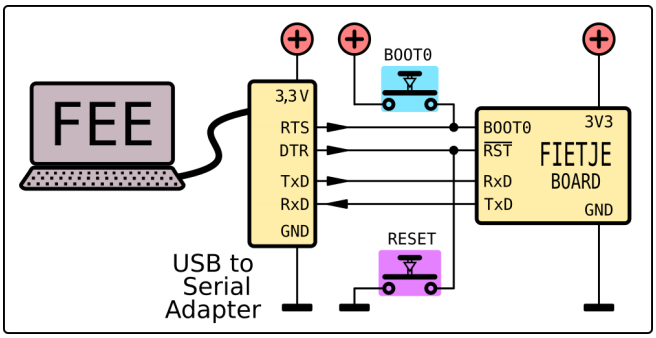


Abbildung 1: Schaltbild

Das Fietje–Mini–Board verfügt an den Pins BOOT0, BOOT1 und RSTN bereits über die nötigen Pull–Up– und Pull–Down–Widerstände, so dass diese für den Normalbetrieb nicht beschaltet werden müssen. Der BOOT1–Pin liegt auf GND (low) und muss daher nicht beschaltet werden.

Man kann das Fietje–Mini–Board auf einem Steckbrett mit ein paar Kabeln an den USB–Adpter anschließen (Abb. 2), oder mit Klammer–Programmieradapter über sechs dafür vorgesehene Pins kontaktieren (Abb. 3).

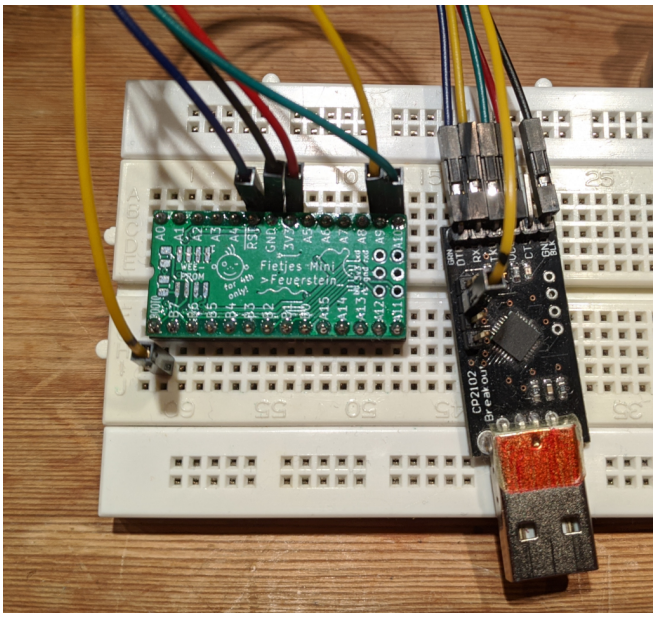


Abbildung 2: Variante „Steckbrett“

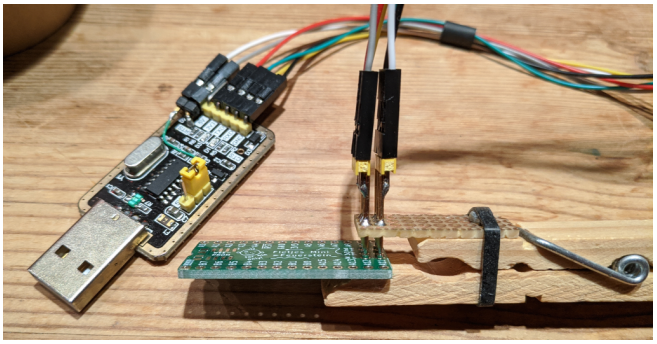


Abbildung 3: Variante „Pogo–Klammer“

Endlich das Flashen

Hat man die Verbindungen geschaffen, wird das Zielsystem via USB–Adapter mit Spannung versorgt und kann von FEE vollautomatisch oder halbautomatisch (per Reset–Taster und BOOT0–Taster, siehe Abb. 1) gesteuert werden. Man ruft im Menü „tools“ die Aktivität „firmware flash“ auf und FEE erklärt auf der Ausgabe im Terminalfenster, was zu tun ist: die Datei mit dem gewünschten Mecrisp auswählen, Anweisungen von FEE befolgen. Jetzt heißt es, zuzugucken, wie FEE eine Sicherheitskopie des aktuellen Flashinhaltes anlegt, das Flash komplett löscht, das Flash mit dem ausgewählten Binärfile beschreibt und zum Schluß eine Verifikation auf korrektes Schreiben durchführt — fertig. Bei Benutzung der Handshake–Steuersignale wird das Zielsystem anschließend sofort in das Forth gebootet und FEE ist bereit für die Codeentwicklung.

Ja, so einfach kann das sein.

Auf `https://forth-ev.de/wiki/projects:fee:start` findet ihr die Projektseite von FEE. Dort gibt es die FEE, Links und weitere Informationen.

First Steps Towards an Astroimaging Control System in Forth

Andrew Read

Astroimaging is a modern hobby using small telescopes equipped with digital cameras. Computerized mounts point at celestial targets and track them during long duration exposures. Setups can be portable equipment in a back–garden, small homemade observatories, or piers in "for–rent" observatories under excellent dark skies. A common factor is a local PC (most commonly Windows) accessed via remote–desktop software from inside the warm home, or even from the other side of the world.

Figure 1: My telescope in Chile

The usual astroimaging software is not especially pleasing to a person familiar with Forth: unstable GUI interfaces, intransparent control logic, clunky macro interfaces, problems solved mainly by guesswork. For exactly these reasons I am motivated to develop a Forth control system for operating my own astroimaging equipment, which is located at the Deep Sky Chile hosted observatory (fig. 1). The goals of the project are, firstly, to develop a logical and reliable control framework for the individual components: the *mount*, the *camera*, the motorized *focuser*, the motorized *filter wheel*, the various *sensors*, and secondly to develop a *domain–specfic language* to pursue astroimaging through a Forth interpreter.

The system components operate quite differently: the mount (a 10Micron GM2000) is the easiest — a TCP/IP network device with a plain–text command protocol. The USB 3.0 CMOS camera (an ASI6200) and filter wheel (a ZWO EFW) are supplied with a C–language SDK. The USB 2.0 focuser (a component of the Takahashi CCA–250 telescope) has an unpublished protocol that will need to be reverse–engineered, but it is a native USB device not a COM port.

I started work hands–on, trying to make contact with the mount. Out of familiarity I chose to work with VFX Forth on Windows. For TCP/IP communication VFX gives access to the WinSock kernal with words `TCPconnect`, `writesock`, `pollsock`, `readsock`. Looking at `10Micron.f` (listing), words `10u.ask` and `10u.tell` communicate text strings with the mount. I wasn't sure what to expect from the mount and so `10u.ask` was coded with error checking, a repeating loop and delays from the very beginning. The extra care was rewarded — the mount responded and gave its status on the first attempt!

Next I built up a series of "hard–coded" words corresponding one–to–one to 10Micron protocol commands. This gave me confidence with the protocol and clarified a few gaps in the documentation. After that the command words were replaced with three different defining words (some protocol commands need an input, some do not, and some make no response).

The mount expects celestial coordinates to be specified in degrees (–90 to +90) or hours (0 – 23), arcminutes (0 – 59) and arcseconds (0 – 59). A target on the celestial sphere requires coordinates in both Right Ascension and Declination, 6 integers in total. Anticipating the benefit of interactive control at the Forth interpreter I chose the simplest possible format: 3 integers on the stack for each of `RA` or `Dec`. `$DEC` (and `$RA`, `celestial.f`) pack the three integers into the string format required by the mount protocol.

Turning to the domain specific control language, `MAKE-TARGET` (in `mount.f`) is a defining word for celestial targets. `GOTO` combines three protocol control words.

Interactive sessions now become possible. Have you seen the Great Orion Nebula through a telescope? We define it and point.

```
( RA)   05 36 25
( DEC) -05 22 33
MAKE-TARGET M42
M42 GOTO
```

Even at this elementary stage I instinctively prefer to use the Forth interpreter to the usual software. Typing a word feels more precise and professional than hunting through a GUI and pressing buttons. Visibility to the mount's own command protocol makes problem solving well–informed and fast–moving. Overall, Forth encourages new ideas that I try out iteratively and interactively, saving my progress in Forth definitions and simple include files, meanwhile building up an engineer's instinct for the system.

The next steps will be more challenging: controlling the focuser will require learning how to approach a native USB device in VFX Forth. The camera will require mastering the C–language SDK and then interfacing it to Forth words.

This is an open source project (Ptolemy on Anding's GitHub page) and I hope to build a niche community around the project — if you are interested in astronomy or in controlling this kind of equipment from Forth, please get in touch!

I am grateful to Ulli Hoffmann for his ideas and suggestions on this and other projects through our many discussions.

February 2023 — andrew81244@outlook.com

Listings

```
1   \
2   \ 10Micron.f - code for controlling the 10Micron mount
3
4   0 value MNTSOC
5   \ value type holding the socket number of the 10Micron mount
6   256 buffer: MNTBUF
7   \ buffer to hold strings communicated with the 10Micron mount
8
9   : 10u.tell ( c-addr u --)
10  \ pass a command string to the mount
11      10u.checksocket if drop drop exit then
12      dup -rot                        ( u c-addr u)
13      MNTSOC writesock                ( u len 0 | u error SOCKET_ERROR)
14      SOCKET_ERROR = if ." Failed to write to the socket with error " . CR exit then
15      <> if ." Failed to write the full string to the socket" CR exit then
16  ;
17
18  : 10u.ask ( -- c-addr u)
19  \ get a response from the mount
20      10u.checksocket if MNTBUF 0 exit then
21      0 >R 5                          ( tries R:bytes)
22      begin
23          1- dup 0 >=
24      while
25          200 ms
26          MNTSOC pollsock             ( tries len | tries SOCKET_ERROR)
27          dup SOCKET_ERROR = if
28              drop ." Failed to poll the socket " CR
29          else
30              0= if
31                  ." 0 bytes available at the socket" CR
32              else
33                  MNTBUF 256 MNTSOC readsock   ( tries len 0 | tries error SOCKET_ERROR)
34                  SOCKET_ERROR = if            ( tries ior)
35                      ." Failed to read the socket with error " . CR
36                  else                         ( tries len)
37                      R> drop >R drop 0        ( 0 R:bytes)
38                  then                         ( tries R:bytes)
39              then
40          then
41      repeat
42      drop MNTBUF R>
43      dup 0= if ." No response from the mount" CR then
44  ;
45
46  : MAKE-COMMAND
47  \ defining word for a 10Micron command
48  \ s" raw-command-string" MAKE-COMMAND <name>
49      CREATE  ( caddr u --)
```

```
50          $,                          \ compile the caddr u string to the parameter field as a counted string
51      DOES>   ( -- caddr u)
52          count                       \ copy the counted string at the PFA to the stack in caddr u format
53          CR 10u.tell
54          10u.ask 2dup type
55  ;
56
57  s" :GR#"  MAKE-COMMAND      10u.RA? ( --)
58  \ get the 10Micron telescope right ascension in the raw format
59  s" :GD#"  MAKE-COMMAND      10u.DEC? ( --)
60  \ get the 10Micron telescope declination in the raw format
61  s" :Gstat#" MAKE-COMMAND    10u.status?
62  \ get the status of the mount
63
64  \
65  \ celestial.f - handle and convert between various celestial data formats
66
67  : $DEC ( deg min sec -- caddr u)
68  \ obtain a declination string in the format sDD*MM:SS from 3 integers
69      <#                          \ proceeds from the rightmost character in the string
70      0 # # 2drop                 \ numeric output works with double numbers
71      ':' HOLD
72      0 # # 2drop
73      '*' HOLD
74      dup >R
75      abs 0 # #
76      R> 0 < if '-' else '+' then HOLD
77      #>
78  ;
79
80  \
81  \ mount.f - develop a Forth language to control the mount
82
83  : MAKE-TARGET
84  \ defining word to name a set of coordinates in RA and DEC
85  \ (RA ) hh mm ss (Dec) deg mm ss MAKE-TARGET <target-name>
86      CREATE ( hh mm ss deg mm ss --)
87          , , , , , ,
88      DOES> ( -- hh mm ss deg mm ss)
89          dup 5 CELLS + DO
90              I @
91          -1 CELLS +loop
92  ;
93
94  : GOTO ( hh mm ss deg mm ss -- caddr u)
95  \ slew the mount to a target and return the signal from the mount
96  \ <target> GOTO
97      $DEC 10u.DEC 2drop
98      $RA 10u.RA 2drop
99      10u.slew    \ return only the final signal from the mount
100 ;
101
102 : PARK ( --)
103 \ park the mount
104     10u.park
105 ;
106
107 : STOP ( --)
108 \ stop (halt) the mount
109 \ HALT is a VFX multitasking word
110     10u.halt
111 ;
```

Aufgebrezelte SBCs

Rafael Deliano

Die meisten Anwendungen sind simpel. Ein 8–Bit–Controller ist für sie ausreichend. Hauptvorteil ist nicht, wie man meinen könnte, dass sie so niedrige Materialkosten haben, sondern deren geringe Komplexität. Das vermeidet Fehler, und verkürzt die Entwicklungszeit.

Einige Geräte haben aber höhere Anforderungen. Der Anwender liebäugelt dann gerne mit einem neuem Controller, der von allem mehr hat: Pins, IO, Flash, RAM, MHz. Alles fertig auf einem Chip. Die erste Ernüchterung kommt, wenn er feststellt, dass diese Controller typischerweise in Gehäusen mit Anschlussrastern kleiner 0,5 mm (z. B. QFPs oder BGAs) kommen, die von Hand schwer zu löten sind. Eine weitere Schwierigkeit ist, dass die Pins nur mühsam mit dem Oszilloskop zu kontaktieren sind.

Besonders bei Geräten, die in Einzelstücken oder kleinsten Stückzahlen gebaut werden, müssen Entwicklungskosten, nicht der Materialaufwand, minimiert werden. Naheliegender ist es dann, zu prüfen, ob der kleine 8–Bit–Controller, den man normalerweise verwendet, geeignet erweitert werden kann. Bei IO und Datenspeicher ist das leicht möglich.

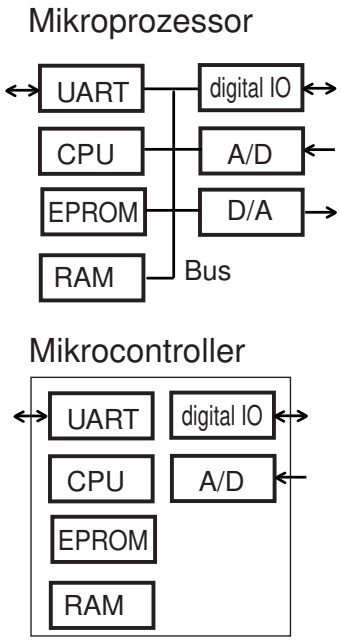


Abbildung 1: uP und uC

Mikroprozessor

Der ursprünglich verwendete 8–Bit–Mikroprozessor (CPU 65C02) auf meinen SBCs hatte CPU, Speicher und IO über einen parallelen Bus verbunden (Abb. 1), der auf der Platine liegt, extern sozusagen, damit zugänglich. Man hatte freie Wahl beim Umfang und Typ von Speicher und IO.

Einchip–Controller (MCUs) haben alles auf einem Chip integriert. Billig und klein. Aber der Mix an Features ist nun vorgegeben und entspricht den Vorstellungen des IC–Herstellers, was „typische Anwendungen“ so brauchen könnten. Programmspeicher ist meist üppig vorhanden, aber oft zu wenig RAM. Ihre Gehäuse, die zwar leicht zu verarbeiten sind, haben nur wenig Pins und damit auch nur eine begrenzte Anzahl an IO–Ports. Und wenn A/D– und D/A–Wandler integriert sind, genügt deren Auflösung meist nicht für Messfunktionen.

Simulierter Bus

Dennoch lässt sich so ein klassisches *Memory–Mapped–IO* mit den alten ICs für Mikroprozessoren aufbauen (Abb. 2). Der dafür benötigte Bus lässt sich per Software nachbilden, jedenfalls, soweit der Controller mit 5 V versorgt wird und man zwei 8–Bit–Ports erübrigen kann, ist das möglich.

Die Auswahl an ICs für den Mikroprozessorbus war ehedem üppig. Sie erleichtern bestimmte Anwendungen deutlich. Dual–UARTs mit integrierten tiefen FIFOs eignen sich z. B. prima für Logger via RS232 (Abb. 4 und Abb. 7). Um mehr Portpins zu bekommen, sind die alten LSIs[1] nicht mehr so attraktiv, meist begnügt man sich mit 74HC574 und 74HC(T)541. Die typische Anwendung für den externen Bus ist schneller Zugriff auf 12– bis 16–Bit–A/D– und D/A–Wandler. Diese analogen ICs können auch auf die ±5 V oder ±15 V umsetzen, was für die Analogtechnik oft günstiger ist.

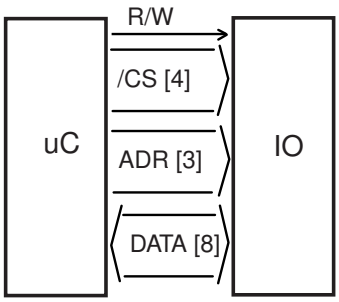


Abbildung 2: Controller mit parallelem Bus erweitert

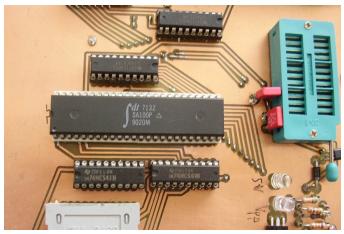


Abbildung 3: Dual–Port–RAM

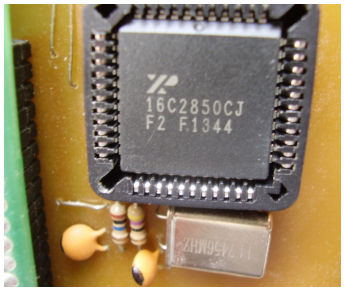


Abbildung 4: XR16C2850 Dual–UART

Dual–Port–RAM

Der Vorteil des FIFOs ist, dass es keine Adresspins hat und damit die Pinzahl gering bleibt. Bei diesem RAM

[1] LSI — Large Scale Integration

sind allerdings beide Busse komplett und doppelt vorhanden. Die Bauform ist dann bei bescheidenen 2 kByte bereits ein unförmiges DIL48. Die klassische Funktion war die schnelle Verbindung von zwei Mikroprozessoren über gemeinsames RAM. Das wird bei Controllern heute nur noch in Sonderfällen sinnvoll sein. Eine naheliegende Anwendung ist immer noch der EPROM–Simulator für 2716 (Abb. 3). Für Aufbauten mit Retro–CPUs wie 4004 und 8008 auch heute noch empfehlenswert.

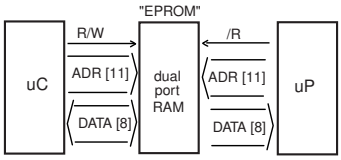


Abbildung 5: Anwendung Dual–Port–RAM

FIFO

Bei Messdaten hat man häufig das Problem, dass nur kurz, aber sehr schnell gesampelt werden muss. Der Controller selbst ist dafür zu langsam. Das ist eine sinnvolle Anwendung für alte FIFOs. Diese Sorte RAMs sind für 2 kByte bis 32 kByte mit Zugriffszeiten von etwa 35 nsec auch in DIL noch immer handelsüblich.

CCDs[2] sind eine typische Anwendung in Kombination mit einem 8–Bit–Flash–A/D–Wandler (Abb. 6). Auch das Mitloggen schneller binärer Bussignale von z. B. JTAG–Debug–Schnittstellen ist eine gängige Anwendung. Ein Controller als Logic–Analyzer sampelt nicht nur rohe Bitmuster, sondern kann das Protokoll später auch flexibel decodieren.

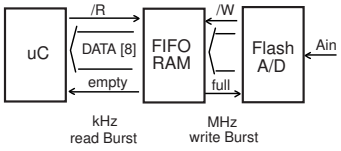


Abbildung 6: Anwendung FIFO

Serieller Bus

Die meisten Controller haben hardwareunterstützte serielle Schnittstellen nach außen geführt, wie SPI oder I²C. Dass die nur wenige Portpins benötigt, ist ein Vorteil, reduzierte Geschwindigkeit der Nachteil. I²C ist durch die bidirektionale Datenleitung mit Pull–up–Widerstand etwas störanfällig, braucht aber weniger Pins als SPI. Beim SPI wiederum kann man das Signal bei reduziertem Takt auch über ein Flachbandkabel führen. Das bietet Vorteile beim mechanischen Aufbau, wenn man Verbindungen von einer horizontalen Hauptplatine zur Platine mit Bedienelementen auf der Frontplatte benötigt. Denn Letztere müssen oft „schräg“ angeordnet werden (Abb. 9).

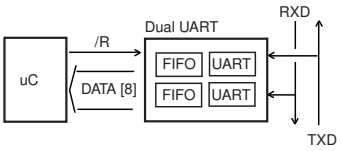


Abbildung 7: Anwendung RS232–Logger

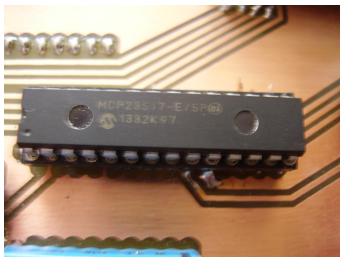


Abbildung 8: MCP23S17, eine serielle Porterweiterung

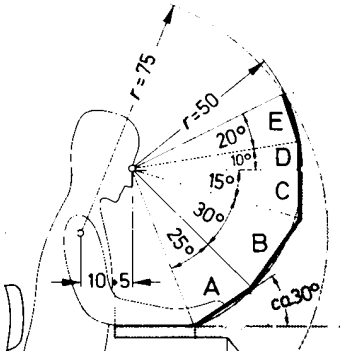


Abbildung 9: Ergonomie für Geräte

Speicher

Die traditionelle Anwendung waren immer kleine serielle EEPROMs. Selbst, wenn der Controller heute internes EEPROM hat, finden sich immer noch viele Anwendungen. Z. B. können viele arithmetische Funktionen mit guter Auflösung durch interpolierte Tabellen dargestellt werden. Weil diese Tabellen speicherintensiv sind, das Flash im Controller aber lieber für für den Programmcode reserviert wird, ist ein 128–kByte–SPI–EEPROM 25LC1024 eine sinnvolle Anwendung dafür.

Die Alternative zum EEPROM ist heute häufig das FRAM. Für 5 V gibt es das FM25W256 (32 kByte) in SOIC8. Der Vorteil gegenüber dem EEPROM ist, dass sie sich deutlicher schneller beschreiben lassen. D. h., wenn man Spannungseinbrüche der Versorgung rechtzeitig erkennt, kann man die Daten aus dem RAM ins nichtflüchtige FRAM retten.

Ehedem war man gewohnt, auf Mikroprozessoren reichlich RAM zur Verfügung zu haben. Für manche Anwendungen ist das auch zwingend nötig. Ein SPI–SRAM 23LC1024 arbeitet mit 5 V und hat üppige 128 kByte RAM.

Der Zugriff auf diese seriellen Speicher im 8–Pin–Gehäuse ist zwar nur mittelschnell, in vielen Anwendungen greift die Software aber relativ selten zu und es ergibt sich dann kaum ein Nachteil an Geschwindigkeit.

Ports

Inzwischen gibt es auch SPI–Porterweiterungen, wie den MCP23S17 (Abb. 8), die bitprogrammierbare Pins wie im Controller bieten. Tradionell verwendet man für IO aber die MSI[3] Schieberegister wie 74HC595 bzw. 74HC165.

Coprozessor

Man kann seinen einfachen „8–Bitter“ aber nicht nur mit simplem IO oder Speicher verstärken. Auch ICs mit komplexeren Funktionen sind heute oft verlockend preiswert geworden, z. B. der Floating–Point–Coprozessor AM9511 oder der Decoder für Barcode–Lesestifte HBCR2211 oder der Motorcontroller LM629. Das spart Entwicklungszeit für Software und ermöglicht Funktionen, die für 8–Bit–CPUs anders nicht in Reichweite wären.

[2] CCD — Charge Coupled Device, lichtempfindlicher Sensor, u. a. für Kameras
[3] MSI — Medium Scale Integration

Programme vom Datenträger booten

Rafael Deliano

Bei PCs ist es der Normalfall, ein passendes Testprogramm zu laden, wenn es gebraucht wird. Bei kleinen Controllern ist das nicht üblich. Aber wenn ein Kabeltester für eine Vielzahl von Kabeln eine Vielzahl passender kurzer Testprogramme benötigt, ist es eine sehr saubere Lösung, die Programme auf den Steckeradaptern gleich mitzuliefern. Das ist in Forth leicht umsetzbar.

Die erste Generation V1.0 des Kabeltesters (Abb. 1) hatte den GP32–Controller mit 8–Bit–A/D–Wandler und wenig RAM. Und als Option auf der Rückseite einen steckbaren Datenträger mit EEPROMs zur Konfiguration (Abb. 3). Das war umständlich und wurde daher nie verwendet. Zudem stellte sich heraus, dass die Konfiguration über Datentabellen nicht ausreichte. Jedes Kabel benötigte doch ein eigenes kleines Programm.

Abbildung 1: Kabeltester mit 64pol. Buchse

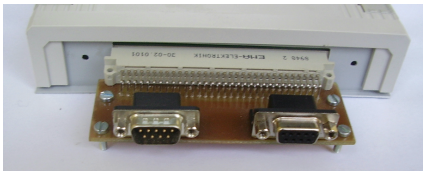


Abbildung 2: Kabeladapter gesteckt

Abbildung 3: V1.0 mit Datenträger gesteckt

Es war einfacher, auf den Adapter–PCBs Drahtbrücken vorzusehen, die abgefragt wurden, und anhand derer dann das richtige Prüfprogramm startete (Abb. 2). Mit steigender Anzahl Kabeltypen füllte sich der Programmspeicher. FORTH–Programme sind zwar kurz (Tab. 1), aber es summiert sich. Zudem wurde die Identifikation der Boards schwierig, da die Drahtbrücken–Codes oft willkürlich gewählt wurden.

Kabelname	Bytes
ROHR–MIL	1239
CABLE–560	395
MULTI–MIL	1382
SINGLE–MIL	654
506–50pol	389
SQG–neu	703
506–ODU	350
Hauptprogramm	7042

Tabelle 1: Speicherbelegung V2.0

Zu diesem Zeitpunkt war bereits der Wechsel zur V2.0 auf den Controller AW60 erfolgt. Die höhere Auflösung von dessen 10–Bit–A/D–Wandler ermöglichte zudem eine genauere Widerstandsmessung. Er hatte auch mehr RAM, von seinen 2 kByte sind 1,7 kByte frei. Die Von–Neumann–CPU 68HC08 kann dort Software ausführen. Zumindest die kurzen Programme, die hier benötigt werden.

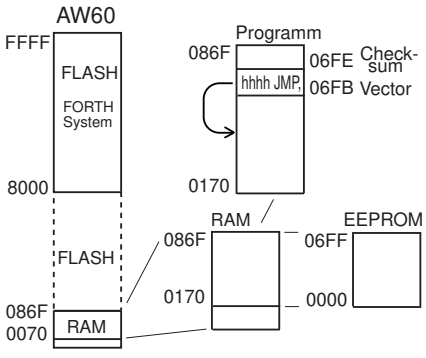


Abbildung 4: Memory–Map V3.0

Der Speicher dafür sollte sich direkt auf dem PCB des Kabel–Adapters befinden. Die ursprünglich verwendete 64pol. Anschlußbuchse hat die mittlere Pinreihe nicht belegt (Abb. 1)[1]. Damit war ein Upgrade auf die 96pol. Buchse möglich, durch die weitere 32 Pins verfügbar werden. Genug für den Anschluss des Speichers. Verwendet wird ein serielles EEPROM 25LC256. SPI benötigt zwar mehr Pins als I²C, hat aber sauberere Signale. Schnelles Booten ist damit unkritischer. Die Pinbelegung am Stecker ist auf ein Layout für die handelsübliche SO8–Bauform des EEPROMs abgestimmt (Abb. 5) und so einfach, weil das Kabeladapter–PCB fast immer einlagig ausgeführt ist. Die Schaltung dort hat nur zwei Bauteile (Abb. 6).

Abbildung 5: EEPROM

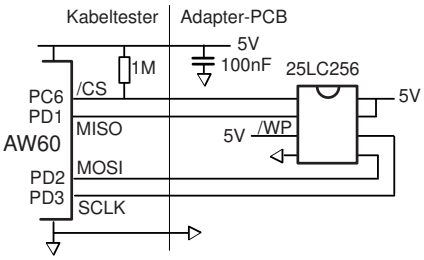


Abbildung 6: Schaltung EEPROM

Software

Die ist problemlos bei dem simplen Compiler von nanoFORTH. Dessen Variable `>C` zeigt normalerweise auf freien Programmspeicher im Flash. Sie wird aufs RAM an Adresse `0170h` umgebogen und compiliert

[1] Diese Leiterplatten–Backplane–Signalsteckverbinder der Serie DIN 41612, Typ C, sind verschleißfest, robust und haben eine lange Steckzyklus–Lebensdauer.

dann dorthin. Das Programm kann für Tests sofort ausgeführt werden. Es ist aber natürlich nicht reset–fest. Dazu muss der 1,7–kByte–Block ins externe EEPROM kopiert werden. Dabei wird auch eine simple additive 16–Bit–Prüfsumme erstellt und ein Programmstart–Vektor initialisiert.

Run

Nach dem Reset prüft das System, ob ein externes EEPROM ansprechbar ist, kopiert den Datenblock daraus ins RAM und berechnet gleichzeitig die Prüfsumme. Stimmt alles, erfolgt der Start des Programms über den Vektor.

Kabeladapter

Theoretisch könnte die überschaubare Zahl alter Adapter–PCBs weiterverwendet werden. Allerdings müsste man dafür parallel auch die alte Struktur der Software weiter unterstützen. Das ist aber unattraktiv, diese Komplexität meidet man besser. Support von Legacy–Systemen zahlt sich selten wirklich aus.

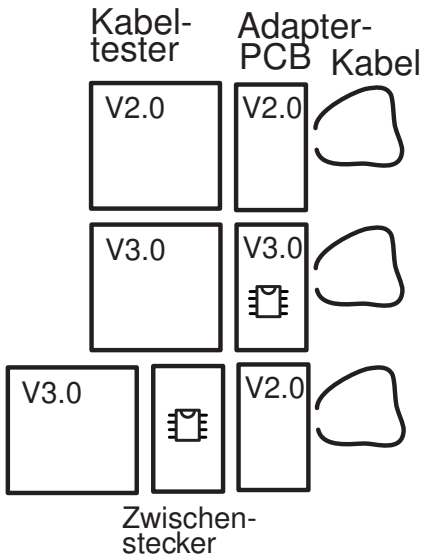


Abbildung 7: Konfiguration

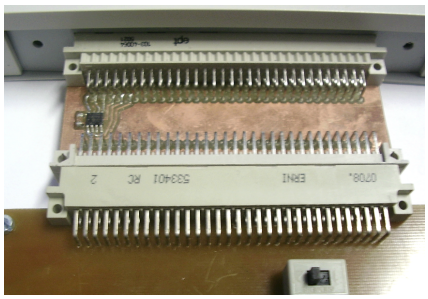


Abbildung 8: Zwischenstecker mit seriellem EEPROM

Erneutes Layouten der alten Platinen, jetzt mit EEPROMs, ist also die sauberste Lösung und langfristig angestrebt. Kurzfristige Alternative ist aber ein Zwischenstecker, in dem sich nur der Speicher befindet (Abb. 7, Bildmitte; Abb. 8). Allerdings müsste man Dutzende davon aufbauen, wenig erfreulich. Nicht elegant, aber wirtschaftlicher, ist, davon abgeleitet, ein Zwischenstecker mit 8 EEPROMs (Abb. 7, Bild unten), die man dann per Jumper an ihrem Pin `/CS` selektiert. Damit kann man mit wenigen Platinen alle alten Adapter konvertieren. Nachteil: Die Zwischenstecker haben ja auch wieder Steckverbinder, wodurch ein zusätzlicher Übergangswiderstand auftritt, der die Messwerte etwas verschlechtert.

Hintergrund

Produktion von Kabeln findet zwar in den meisten Elektronikfirmen statt, deren Test wird aber vernachlässigt. Auch, weil bezahlbare Geräte am Markt nicht mehr gängig sind. Eigenbau hilft. Die Schaltung ist nicht komplex, die 8–kByte–Software auch nicht.

Der Trick war, dass ich ein gutes, aber inzwischen obsoletes Gerät als Ausgangspunkt genommen habe: den WK2 der WEE GmbH, die darauf spezialisiert ist [1] (Abb. 9). Wie Newton es formuliert hat: „If I have seen further, it is by standing on the shoulders of giants.“

Abbildung 9: WK 2 (1989 – 1993)

Meins ist zwar kein kommerzielles Produkt, wird aber inzwischen in zwei Firmen für den Serientest in der Produktion eingesetzt. Da alle Leiterplatten der ersten Version nun verbraucht sind, musste die Neuproduktion gestartet werden. Ein sinnvoller Zeitpunkt für das Upgrade.

Abschließend noch ein Blick ins Gehäuse auf die Leiterplatte Version V1.0 mit dem GP32–SBC, um auch mal zu zeigen, wie denn diese SBCs in Geräte integriert sind (Abb. 10).

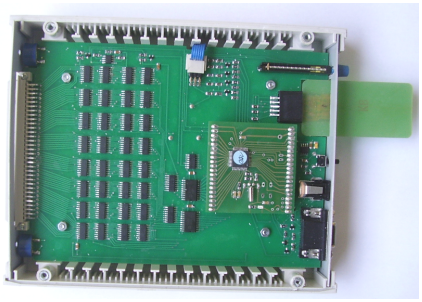


Abbildung 10: Innenansicht des Kabeltesters V1.0. Quadratische Platine rechts in der Bildmitte: SBC mit nanoFORTH auf GP32. Rechts oben gesteckt: Datenträger mit seriellen EEPROMs. Links das Array aus 74HC4051 Multiplexern, um die Leitungen zu messen.

[1] `www.weetech.de/unternehmen/geschichte` WK 2 (1989 – 1993)

Forth für SPI–Flash bildhauern

Michael Kalus

Neulich bot sich eine Gelegenheit, diese kleinen externen Flash–Datenspeicher am SPI–Bus näher kennenzulernen. Ein Freund hatte das Problem, dass sein Kopierer für die SPI–Flashs hin war. Er lebt in den USA und hat eine kleine Produktion an Platinen, die er ganz gut verkauft. Was tun? Eine kleine MCU, noForth und zwei Steckplätze = Flash–Kopierer. Sollte nicht so schwer sein, oder?

Doch vor den Erfolg hat der Herrgott den Schweiß gesetzt. Wenn auch die einzelnen Schritte gar nicht so schwer waren, manchmal ging es auch dabei steil bergan. Doch einmal erklommen, kommt man auf der SPI–Hochebene mit Forth erfreulich bequem voran, wandert dort mit Lust mal hierhin, mal dahin, bis man alle Ecken durchstreift hat. Interaktiv, wie Forth eben ist, läd es geradezu ein, die ganze Landschaft zu erkunden — darum benutzte ich es ja.

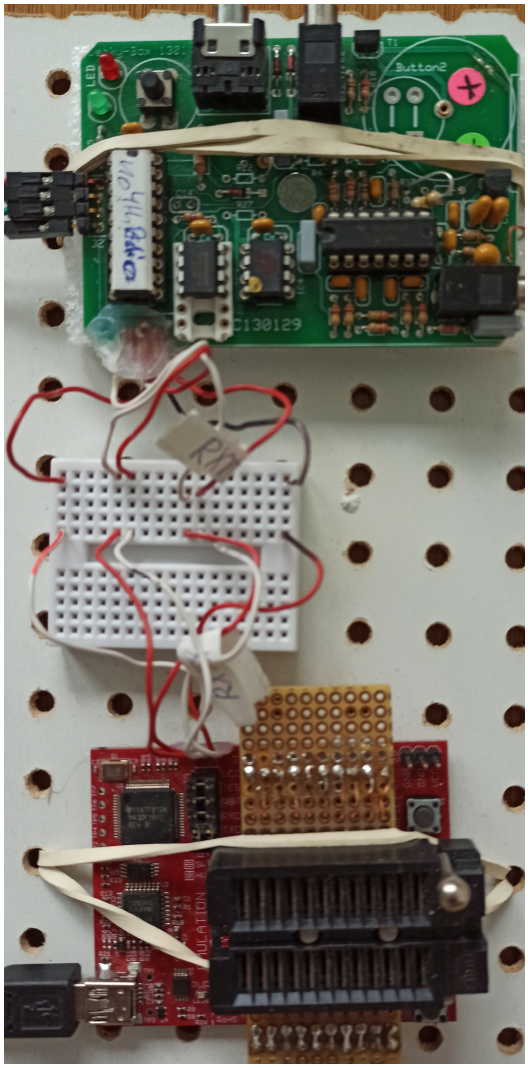


Abbildung 1: Oben im Bild das zweckentfremdete Audioboard. Die beiden DIP8 sind die Flashs. Links daneben die MCU mit noForth. Alles andere ist Audiozeugs; es wird zum Kopieren nicht gebraucht. Unten im Bild mein antikes Programmer–Launchpad, gleichzeitig USB–Seriell–Wandler.

Hardware

Der Aufbau war simpel. Die Platine des Freundes hatte schon zwei Steckplätze für die SPI–Flashs und den MSP430G2553 von Texas Instruments (Abb. 1).

Die MCU bekam nun statt ihres C–Programms das noForth verpasst. Ein paar Serial–Flash–Memories des Typs W25X40CL von Winbond in der Größe 4 MBit (512 KByte) zum Ausprobieren noch dazu. Einer der beiden SPI–Steckplätze hielt das Flash–Original, der andere bekam den leeren Chip für die Kopie.

Der SPI–Bus stellte sich als völlig unkritisch heraus auf so kurze Distanz von wenigen Zentimetern. Die beiden seriellen Leitungen, Data–Input und –Output, handhaben die Chips auf beiden Seiten der Leitungen selbst. Die MCU ist dabei der Master, die Flashs sind Slaves. Den Takt macht der Master — Serial Clock, `CLK`. Das alles erledigen die Hardwaremodule. Einmal initialisiert, läuft das SPI–Protokoll automatisch, Bit–Bang unnötig.

Aber um die Drähte zur Selektion der Chips muss man sich kümmern. Dazu genügten dann zwei I/O–Bits von einem der MCU–Ports, die man `high` oder `low` setzen kann. Jeder Chip bekam sein eigenes /CS, fertig war die Sache.

Woher ich das wusste? Wusste ich ja alles nicht. Also stöbert man erst einmal durch die Papiere, die es dazu gibt.

Dokumente

Das Datenblatt/Benutzerhandbuch von Winbond ließ keine Fragen offen. Das Handbuch zur MCU auch nicht. Und noForth ist ebenfalls gut dokumentiert. So kann man arbeiten. noForth hat ein kleines SPI–Paket für diese MCU. Das war ein guter Anfang, um sich im SPI–Hardware–Treiber der MCU zurechtzufinden. So war ein erster Test erfolgreich: die ID eines SPI–Flashs lesen.

So habe ich bald gelernt, dass diese SPI–Flashs eine ganze Reihe von Kommandos verarbeiten, die einem das Leben leicht machen, was das Schreiben und Lesen des Flashs angeht.

Werkzeuge

Dieses war ein Windows–10–Projekt. Das wollte der Freund so. Des Programmierers Federkiel ist daher Notepad++ gewesen, welcher ein feiner Editor ist. Die MCU selbst hängt am USB–Seriell–Wandler und der führt zu

TeraTerm, ein für Forth halbwegs brauchbarer Terminal–Simulator. Den nehme ich, weil da nichts Besseres ist — oder bisher war. Und hoffe jedesmal, dass sich da was tut in der Forth–Community.

Das TeraTerm ist forth–dumm, ja, leider. Doch was soll man machen? So hat es zeilenweise Übertragung, am Ende der Zeile gibt es blind Wartezeit fürs noForth, sodass es die Zeile verarbeiten kann und weiter gehts. Da die jeweiligen Forth–Test–Schnipsel doch immer recht klein sind, geht das. Und das Copy&Paste–Feature vom TeraTerm ist ok. Hat man dann ein größeres Stück Forth–Quellcode fertig, funktioniert der File–Download darin auch ganz passabel.

noForth kriegt man in die MCU mit dem FET–Pro–430–LITE–Programmer von Elprotronic — ein sehr zuverlässiges Werkzeug. Das UniFlash von Texas Instruments tuts auch. Doch da ich den anderen Programmer schon mal kenne und da habe, na ja, bleibt man dabei.

Verwendet wurde noForth–mcv. Das m bedeutet „MSP430“[1], c steht für „compact“ und v für „vocabularies“. Das noForth–Image könnt ihr als Intel–Hex–File von der Webseite der Entwickler holen. [1] Ein MSP–Launchpad als Programmer war mein Eigen, also keine Hürde. [2]

Software und ihre Tests

Die findet ihr im Listing. noForth, klein wie es ist, kennt `marker`, sodass man es rasch immer wieder auf seinen geschützten Kern zurücksetzen kann. Denn die Software–Versuche laufen ja nicht immer auf Anhieb richtig, und der Programm–Müll aus der Probierphase soll ja nicht im Programmspeicher der MCU bleiben. noForth hat eine kleine Bibliothek, `tools` genannt, die man für die Entwicklung läd und gut gebrauchen kann. Da ist `.s` drin, `words` und sowas. Später kann das wieder weg. In meinem Projektchen war Platz genug, also blieb das auch drin.

Wie das Peripherie–Modul für SPI im MSP430G2553 initialisiert werden kann, zeigt freundlicherweise eine kleine noForth–Library. Die hab ich übernommen, an das gegebene Board angepasst, was die Ports angeht, und schon ging es daran, Befehle an den SPI–Speicher zu senden.

Hierzu schreibt Winbond:

> „The instruction set of the W25X40CL consists of twenty basic instructions that are fully controlled through the SPI bus (see Instruction Set table). Instructions are initiated with the falling edge of Chip Select (/CS). The first byte of data clocked into the DI input provides the instruction code. Data on the DI input is sampled on the rising edge of clock with most significant bit (MSB) first.“

Simpel, oder? Denkt man, aber zunächst gilt es, Tests zu ersinnen, um alle Funktionen zu verifizieren. Im Listing der Tests ist diese Vorgehensweise im Einzelnen geschildert. Das Prinzip ist dabei immer gleich: zunächst überlegen, was als Beweis für richtiges Funktionieren gelten könnte, dafür ein Forthwort schreiben und es dann solange umformen, bis es nicht nur gut läuft, sondern auch gut aussieht. Dann zum nächsten gehen.

Das ist das Schöne am interaktiven Forth–Programmieren: Forth lässt sich modellieren, wie Ton in den Händen. Damit wird Programmieren zu einer ästhetischen Modellierarbeit. Jedes Forthwort, das entsteht, ist ein Klümpchen der Aufbauplastik des Bildhauers. Es wird solange in Form gedrückt, bis es richtig sitzt im Gesamtwerk.

Also frisch ans Werk. Die /CS–Leitungen zu prüfen, geschieht mit dem Multimeter, ist der H– oder L–Pegel da, so wie er soll? Das ist ein Prüfschritt, also gibts dafür ein Forthwortpaar: `csup` und `csdown`. Und weil entweder das eine oder andere Flash selektiert werden soll, bestimmt die Maske `CS ,` ob `IC3` oder `IC4` dran ist.

Das erlaubt dann schon Phrasen wie:

```
IC3 csdown ... csup
```

und nachher

```
\ read device ID & Manufacturer
IC3 ID ( -- M ID )
IC4 ID ( -- M ID )
```

Damit kann dann schon gesehen werden, ob die SPI–Flashs da sind und das ganze SPI funktioniert.

Ein Test der SPI–Hardware ist freundlicherweise seitens des MCU–Herstellers schon vorgesehen. Es gibt die Möglichkeit, das Modul in eine interne Schleife zu schicken: In dem Modus erscheint das Byte direkt im Empfangsregister, das man gerade ins Senderegister geschrieben hat. Damit kann man seine Sende– und Empfangs–Forthworte testen. Was auf der Sendeleitung rausgeht, muss ja auf der Empfangsleitung ankommen. Das Wort

```
SPI ( u1 -- u2 ) \ Master SPI routine
```

lässt sich so testen.

Nachdem also klar ist, dass SPI–Bus und /CS–Leitungen korrekt arbeiten, ist es eine Freude zu sehen, dass die Flashs Kommandos entgegennehmen und brav beantworten. Sie liefern bereitwillig ihre Kennzahlen „Device ID & Manufacturer“ auf dem Datenstack ab. `.s` zeigt das sofort — hach, ist Forth doch schön.

Das Manöver war natürlich etwas blauäugig. Aber Glück gehört nun mal dazu: noForth ist langsam genug, um dem SPI–Flash Zeit zu geben, sich nach dem /CS einzuschwingen.[2] So war das Timing kein Problem bei so einer einfachen Anfrage an das Flash, wie: „Was ist deine ID?“ Nur nach dem INIT braucht es einmal ganz zu Anfang `5 ms` extra, bis alle Komponenten betriebsbereit sind.

Ab da ist es dann ein Spaziergang in der SPI–Hoch(sprachen)–Ebene, die Kommandos für das SPI–Flash in Worte zu fassen.

[1] MSP430G2553 MCU, Texas Instruments

[2] /CS Active Setup Time relative to CLK beträgt 5 ns laut Datenblatt des SPI–Flash.

```
: status ( -- u )
  csdown
    05 >spi   \ read status register
    spi>
  csup ;

: ID ( -- M ID )
  csdown
    90 >spi   \ read device ID
    0 0 >adr24 spi> spi>
  csup ;
```

Gesendet werden nacheinander die Instruktion, eine Adresse und dann Daten, entweder hin oder her.

Das besondere am SPI: Derweil von der Master–Seite auf der Sende–Datenleitung geschrieben wird, empfängt das Modul auf der anderen Datenleitung bereits Bytes vom Flash. Das war Anfangs etwas verwirrend für mich. Aber man lernt bald, wann die empfangen Bytes nur Dummies sind, die man fallen lassen kann, und ab wann Gültiges kommt.

Die Flash–Instruktionen sind alle nur ein Byte lang, die Adressen 3 Bytes — 24 Bit bei der Flashgröße, die hier im Einsatz war — und die Daten kommen wieder byteweise.

Für die interaktive Arbeitsweise ist die Senderoutine `>spi` bereits gut genug, um ein Instruktions–Byte an das Flash zu schicken. Die nachfolgende Adresse sendet man mit `>adr24` und die Daten wieder byteweise. Start und Stopp der Übertragung sind auch Forthwörter: `csdown` für die fallende Flanke, mit der eine Sequenz eingeleitet wird und `csup ,` um diese zu beenden. Das Timing dafür ist in `spi` ja bereits enhalten: dort wird gewartet, bis das *busy flag* in den Sende– und Empfangsregistern anzeigt, dass der Chip fertig ist.

Tja, das war es eigentlich schon, was ich dazu zu sagen hätte. Um zu verdeutlichen, wie ich da 'ran gegangen bin und dass es Spaß gemacht hat.

Den Rest kriegt ihr selber heraus mit Blick in den Quellcode.

Benutzeroberfläche(n)

Das `blockcopy` ist das eigentliche Arbeitstier, das habt ihr da sicherlich schon gesichtet. Die darauf aufbauende Verwaltung der Abschnitte bis zur `copyapp` ist nichts weiter, als ein Mechanismus, um den Benutzer froh zu stimmen. Denn das war ja das eigentliche Ziel: eine eigenständige Platine zu haben, die Flashs kopiert. Flash stecken, einschalten, um es zu kopieren, ein bischen „Blinken Light“ für die menschliche Orientierung, wieder ausschalten, kopiertes Flash entnehmen, und das Ganze nochmal von vorn. Keine weiteren Taster, nur die eine LED — die Minimalausstattung des Boards, „wie es ist“.

Doch da das Board auch über ein Terminal an der seriellen Schnittstelle betrieben werden kann, wurde noch eine zweite Benutzerebene spendiert: Die Komandozeile nimmt `copy+` und `copy-` an. Setzt man den `report`–Value, spricht der Haufen Draht zu einem, sonst wird still kopiert.

Aber das hat mit dem SPI ja schon nichts mehr zu tun, das ist so ein Drumherum, dass andere Benutzer wohl anders haben möchten. Wie man den *Buisiness Code* sauber trennt von der Benutzeroberfläche und seiner Sprache[3], also lokalisiert, das muss ich noch herausfinden. Im Gforth ist das schon möglich[4], noForth unterstützt das, soweit ich weiß, noch nicht. Und ob so ein sehr kleines Forth das überhaupt kann, muss auch noch erforscht werden.

Übrigens, das `copy` braucht 15 s. Zum Vergleich: Das flash–interne komplette Chiperase dauert 12 s. Und `copy` macht dabei bereits die Verifikation und „spricht“ zum Benutzer, was aufhält. Nicht schlecht, oder?

Ideen, um das Verwechseln von Quell– und Zielbaustein zu verhindern, sind auch willkommen. Jetzt ist das erstmal primitiv gelöst: Die Quelle steckt fest im Sockel IC4 und nur von IC4 nach IC3 kann geschrieben werden.

Ich wünsche viel Vergnügen mit Forth.

Links

[1] `https://home.hccnet.nl/anij/nof/noforth.html`

[2] `https://www.ti.com/tool/MSP-EXP430G2ET`

Listings

Natürlich müssen vorher die `tools` des noForth geladen werden. Die sind hier nicht abgedruckt, um das Magazin nicht zu überfrachten.

spi

```
1   \ spi for noForth m(cv) -- september 2022
2   \ MSP430G2553
3
4   tools\
5   hex
6   : spi-  01 69 *bis ; \ UCB0CTL1  stop bit set
7   : spi+  01 69 *bic ; \ UCB0CTL1  released --> run
8   : MASTER-SETUP
9     spi-
10      \ ports general i/o
11      A0 22 *bis  \ P1DIR     P1.5&7 output
12      40 22 *bic  \ P1DIR     P1.6 input
13      \ special function pins
14          E0 26 *bis  \ P1SEL     P1.5+6+7 is SPI
15      E0 41 *bis  \ P1SEL2
16      69 68 *bis  \ UCB0CTL0  Clk=high, MSB first,
```

[3] Uncle Bob: „Clean Code“

[4] Bernd Paysan; Internationalisation mit Gforth; Heft 4d2022-04, S. 10 ff

```
                \ Master, Synchroon
     80 69 *bis  \ UCB0CTL1  USCI clock = SMClk
     08 6A c!    \ UCB0BR0   Clock 16Mhz/8 = 2 MHz
     00 6B c!    \ UCB0BR1
     00 6C c!    \ UCB0MCTL  Not used must be zero!
     E0 21 *bis  \ P1out     P1.5&6&7 set
   spi+ ;

: SPI      ( u1 -- u2 ) \ Master SPI routine
    begin  8 3 bit* until  6F c!    \ IFG2  TX?
    begin  4 3 bit* until  6E c@ ;  \ IFG2  RX?
: >SPI      ( u -- )    spi drop ;
: SPI>      ( -- u )    0 spi ;

shield SPI\
chere u.  \ memory check
( finis)
```

copy

```
\ handle flash memory                    20221020mk
\ Board: BC130129 DB
\ Any other board will do, breadboard is fine too

\ Copy v06 is to use vocabularies.
\ EXTRA words are the factorized parts.
\ Copy V05 with/without report and copyapp
\ TIB holds 80 characters,
\ longer lines will be cut!

SPI\         \ marker from file: SPI.f

hex  fresh   \ see note at the bottom.

value cs  \  mask
: IC3   80 to cs ;   \ mask P2.7 = IC3
: IC4   40 to cs ;   \ mask P2.6 = IC4

v: extra definitions
: csup     CS 29 *bis ;  \ P2out  deselect Flash
: csdown   CS 29 *bic ;  \ select Flash

v: forth definitions
: init    \ prepare chips and control lines
  master-setup  \ SPI
  \ ports
  00 29 c!     \ P2OUT     P2out is all low
  FF 2A c!     \ P2DIR     P2dir all output
  00 2E c!     \ P2SEL     P2 is general i/o
  11 22 *bis   \ P1DIR  P1.0&4 out
  \ enable flashs
  11 21 *bis   \ P1out  P1.0&4 high
               \ = hold high, enable both Flashs
  C0 29 *bis   \ P2out  P2.6&7 high
               \ = deselect both Flashs
  5 ms   \ allow Flashs to initialize
  ic4 ;  \ IC4 is master Flash


\ some "keep the user happy" stuff
: holdH  11 21 *bis ; \ both H
: holdL  11 21 *bic ; \ both L
: bliz   holdh 50 ms holdl 50 ms holdh ; \ LED

V: extra definitions
: beph    20 29 *bis ;   \ P2.5 high
: bepl    20 29 *bic ;   \ P2.5 low
: bep    beph 1 ms  bepl 1 ms ;  \ 1 beep wave
v: forth definitions
: beep  ( -- ) \ short beep of 500Hz
  100 0 do bep loop ;


\ Some read instructions
```

```
V: extra definitions
: >adr24 ( adr24 -- )  \ send 24bit
   \ adr24 = double-cell number ( adrL adrH -- )
  >spi dup 8 rshift >spi >spi ;
: dummies ( n -- )     \ send n dummy bytes
  0 ?do 0 >spi loop ;

v: forth definitions
: ID ( -- M ID )   \ read device ID & manufacturer
  csdown 90 >spi  0 0 >adr24  spi> spi> csup ;
: status ( -- u )  \ read status register
  csdown 05 >spi spi> csup ;

v: extra definitions
: read     ( adr24 -- )  \ set read address
  csdown 03 >spi >adr24 ;



\ write instructions
: ready?  ( -- f ) \ get busy status of Flash
  status 1 and 0= ;  \ tested
: welH  \ set write enable bit
  csdown 06 >spi csup ;
: welL  \ reset write enable bit
  csdown 04 >spi csup ;

v: forth definitions
: chip3erase  \ set all memory to erased state $FF
  ic3 \ never erase IC4
  welH csdown 60 >spi csup
  begin ready? until ;

v: extra definitions
: sector3erase ( adr24 -- )
  ic3
  welH csdown 20 >spi >adr24  csup
  begin ready? until ;

: page ( adr24 -- ) \ write to page
  welH  begin ready? until
  csdown 02 >spi >adr24 ;
  \ endpage is csup


\ copy
hex

\ Organisation of the Flash memory
\ adr24 = adrL adrH = SPBI 000G
\ mit:
\ G = seGment number
\ S = Sector number
\ P = Page number
\ B = Block number
\ I = Byte number
\ There are two address pointers:
\ The "Flash address pointer" contains the numbers
\ of the sectors down to the bytes.
\ The parent segment has its own pointer.

value fap  ( -- fap ) \ Flash address pointer
value sep  ( -- sep ) \ Flash segment pointer

(*
The fastest copy would be to have SPI 2x, read
from one flash and write to the other right away.
This board was designed so that both sockets were
on one SPI bus. The limited RAM in the MCU allowed
only a small buffer for read bytes - at least this
way the copy process didn't get too slow.
A second small buffer is used for verification.
What has been written is read back into this
buffer and compared with the first one.
Copying errors are detected early this way.
*)
```

```
131
132  here 10 allot  constant buffer0  \ 16 bytes buffer
133  here 10 allot  constant buffer1  \ 16 bytes buffer
134
135  value buc  ( -- buc ) \ buffer character counter
136
137  \ Reading and writing of the buffers
138  : buc0   ( -- )    0 to buc ;
139  : buc+   ( -- )    buc 1+ f and to buc ;
140  : >buf0  ( c -- )  buffer0 buc + c!  buc+ ;
141  : buf0>  ( -- c )  buffer0 buc + c@  buc+ ;
142  : >buf1  ( c -- )  buffer1 buc + c!  buc+ ;
143
144  : fapsep0 ( -- ) \ for convenience
145    0 to fap 0 to sep ;
146
147  \ Set important Flash addresses
148  : setblock   ( i -- )
149    f and 10   * fap ff00 and +  to fap ;
150  : setpage    ( i -- )
151    f and 100  * fap f000 and +  to fap ;
152  : setsector  ( i -- )
153    f and 1000 *                 to fap ;
154  : setsegment ( i -- )  f and   to sep ;
155
156
157  \ The "Report" feature was added later. It also
158  \ belongs to the "keep the user happy" category
159  \ and does not add anything to the copy function.
160  value report
161
162  \ The feature "Block=FF" was also added later.
163  \ This uses a property of the master Flash: Its
164  \ segments are not fully written. As soon as a
165  \ block is "FF", nothing more comes in the
166  \ segment, the copy function can go to the next
167  \ segment.
168  value FFsum  ( -- n )  \ sum of bytes in the block
169  : FFcnt ( x -- x )  dup +to FFsum ;
170  : ff?   ( -- f )    FFsum ff0 = ;
171
172  : rbuf     ( -- ) \ read current Flash block
173    0 to FFsum
174    fap sep read  buc0
175    10 0 do spi> FFcnt >buf0 loop  csup ;
176  : wbuf     ( -- ) \ write current Flash
177    ff? if exit then
178    fap sep page  buc0
179    10 0 do buf0> >spi loop  csup ;
180  : bbuf     ( -- )  \ read back written Flash
181    ff? if exit then
182    fap sep read  buc0
183    10 0 do spi> >buf1 loop  csup ;
184
185  \ LED pulses indicate an error
186  : sos ( -- )
187    3 0 do bliz loop 200 ms
188    3 0 do bliz 100 ms loop 200 ms
189    3 0 do bliz loop 500 ms ;
190  : buf<> ( -- f ) \ f = true if different
191    ff? if exit then
192    buffer0 10 buffer1 10 s<>
193    if begin sos again then ;
194  \ Yes, we terminate by "power off"
195
196
197  \ Now, let us copy for real
198  : blockcopy  ( i -- ) \ copy current block
199    setblock
200    ic4 rbuf
201    ic3 wbuf bbuf buf<> ; \ buf<> is verification
202
203  : pagecopy   ( i -- ) \ 0xFF  &256 bytes
204    setpage
205      report if ." block " then
206    10 0 do
207    ff? if unloop exit then
208      report if i . then
209    i blockcopy  loop ;
210
211  : sectorcopy ( i -- ) \ 0x1000  &4096 bytes
212    setsector
213    10 0 do
214    ff? if unloop exit then
215       report if cr ." page " i . then
216    i pagecopy  loop ;
217
218  : segmentcopy ( i -- ) \ 0x10000  &65536 bytes
219    setsegment
220    10 0 do
221    ff? if unloop bliz exit then
222       report if cr ." sector " i . then
223    i sectorcopy loop bliz ;
224
225  : flashcopy   ( -- )  \ 0x80000 = &524288 bytes
226    fapsep0
227    8 0 do
228       report if cr cr ." segment " i . then
229    0 to FFsum  i segmentcopy loop ;
230
231
232  shield flash\
233
234  (* And finally the main word: COPYAPP
235  This will be the "turnkey app", it starts after
236  reset. Actually FLASHCOPY is the copy function.
237  But then you can't see that the board is working
238  correctly. So you have to use "blinken lights"
239  and because the board has only one LED, this one
240  has to do the job. Oh yes, the board can also
241  make a sound.
242  *)
243
244  \
245  v: forth definitions
246  hex
247  : copy ( -- ) \ blink while copying entire flash
248    \ stop watchdog: noforth itself does that
249    \ 5A80 120 !
250      report if
251      cr ." start copying and wait for the system
252      to stabilize ... " then
253    500 ms  init  0 to FFsum
254      bliz bliz bliz   \ keep the user happy
255      report if ." done" then
256      report if cr ." chip-erase ... " then
257    chip3erase
258      bliz bliz
259      report if ." done"
260      cr ." copy 8 flash segments, "
261         ." skip empty (0xFF) blocks." then
262    flashcopy
263      report if cr cr ." finis " then
264      beep ;
265
266  : copy+ ( -- ) \ copy with report
267    true  to report  copy ;
268
269  : copy- ( -- ) \ copy without reporting
270    false to report  copy ;
271
272  : copyapp ( -- )
273    copy-
274    \ ready -
275    \ now we bliz back and forth (pun intended)
276    holdL 200 ms begin bliz again ;
277
278   shield copy\
279  \ ' copyapp to app   \ undo: ' noop to app
280
281  : .. ( adr24 -- ) \ Test: see some 32 bytes of IC
```

```
282     read
283     cr 10 0 do spi> . loop
284     cr 10 0 do spi> . loop
285     csup ;
286   \ passed: 20221018mk
287
288   fresh
289   freeze          \ application done  20221020mk
290   init chere u.   \ check memory: ok
291   ( finis)
292
293   (* Notes:
294   Only in noForth with vocabularies:
295   EXTRA is a vocabulary with non-standard useful
296   words.
297   INSIDE is a vocabulary with internal words.
298   : FRESH ( -- )
299     only extra also forth also definitions ;
300
301   fresh order ↩ ( FORTH FORTH EXTRA ONLY : FORTH )
302
303   When noForth starts, FRESH is executed.
304   .VOC ( wid -- ) \ show the vocabulary name.
305    'wid' is a number in the range 0..127
306   *)
```

Glossar

```
1   \ handle flash memory  for BC130129    2022-10-17
2   \ Glossar of usable forth words
3   \ Type WORDS to get all forth words.
4   \ Type EXTRA WORDS to get sub-words too. See source code for use of extra words.
5
6   cs  ( -- mask ) \  value of IC mask
7   Example:
8   cs . 80  OK.0
9
10
11  IC3 ( -- )  \ set cs mask to P2.7 = IC3
12  IC4 ( -- )  \ set cs mask to P2.6 = IC4
13
14  init   ( -- )       \ prepare chips and control lines
15
16  beep   ( -- )       \ short beep of 500Hz
17
18  ID     ( -- M ID )  \ read device ID & Manufacturer
19  Example:
20  init ic4 id .s ( EF 12 ) OK.2
21
22  status ( -- u )     \ read status register.
23  Example:
24  init ic3 status .s ( 0 ) OK.1
25
26
27  chip3erase ( -- )   \ set all memory of flash in IC3 to erased state $FF.
28  copy       ( -- )   \ copy entire flash from IC4 to IC3.
29  copy+      ( -- )   \ copy with report. Same as: true to report copy
30  copy-      ( -- )   \ copy without reporting. Same as: false to report copy
31
32
33  copyapp    ( -- )   \ use this as turnkey app
34  Example:
35  ' copyapp to app   \ undo: ' noop to app
36
37
38  ..         ( adr24 -- ) \ Testing: look at 32 bytes of IC
39  Example:
40  hex  init IC3 dn 12000 ..  \  look at content of flash in IC3 at address 0x12000.
41  \ adr24 is a double number:
42  @)dn 12000  OK.2
43  @).s ( 2000 1 ) OK.2
44
45  ( finis)
```

Internationalization mit Gforth und MINΩΣ2 — Teil 2

Bernd Paysan

Über die im Teil 1 behandelte Übersetzungsproblematik gibt es eine Reihe weiterer Themen, die zu einer gelungenen Internationalisierung und Lokalisierung gehören, und dabei meist nach Lösungen außerhalb von Standards verlangen. Der Artikel baut auf private Kommunikation mit Claus Vogt *auf, der dem Autor seine Erfahrungen in einer länglichen und unstrukturierten E–Mail mitgeteilt hat, und auf einem Vortrag zu MINΩΣ2, in der verschiedene Satzfragen behandelt wurden.*

Datenbank oder CSV für Übersetzer?

Übersetzer sind, wie schon im ersten Teil angemerkt, typisch eher keine Programmierer. Der erste Teil enthielt daher folgenden Text:

> „Das Verwenden einer Tabelle ist vielleicht für das Warten von Programmen einfacher — insbesondere, weil die CSV–Tabelle selbst dann mit einem Tabellenprogramm gepflegt werden kann; etwas, was viele Leute beherrschen."

Dieser Ansicht hat Claus Vogt voll zugestimmt — auch er verwendet seit längerem CSV–Dateien, in die seine Übersetzer die Übersetzungen eintragen. Neben dem Programmtext kann man dort auch noch weitere Spalten einfügen, die dem Übersetzer das Leben leichter machen, und ggf. auch für automatisiertes Einfügen der Texte an die richtige Stelle sinnvoll sind. So hat Claus eine Spalte „Kontext", in der die Identifier der UI–Elemente drinstehen, zu der die Texte gehören. Der Übersetzer lernt daraus etwa, auf was für einem Bedienelement ein Text landet, und kann bei einem Button eine kurze, knappe Übersetzung wählen, bei einem Textfeld eine längere, präzisere. Das Programm kann dabei mögliche Mehrdeutigkeiten auflösen, und die jeweilige Übersetzung zuverlässig in das passende UI–Element einfügen. Auch nicht zu übersetzende Kommentare können in eine eigene Spalte gehen; Kommentare als Teil des Programm–Texts landen recht zuverlässig auch in der Übersetzung.

Gforth hat aufgrund dieser Überlegungen jetzt einen CSV–Importer bekommen, mit dem dann auch Übersetzungen eingelesen werden können. Dieser Importer findet sich in der aktuellen Entwicklerversion in der Datei `csv.fs` und hat ein einziges, für den Nutzer interessantes Wort:

```
READ-CSV ( addr u xt -- )
```

mit dem die Datei mit dem Namen im String *`addr u`* gelesen, und jedes Feld an *`xt ( addr u col line -- )`* übergeben wird, zusammen mit der Spalte (ab `0`) und der Zeile (ab `1`) des Feldes.

Darauf aufbauend wird dann in `i18n.fs` ein neues Wort definiert:

```
LOCALE-CSV ( "name" ---- )
```

lädt dann die angegebene CSV–Datei zur Befüllung der Übersetzungs–Datenbank, wobei die erste Zeile die jeweiligen Locales enthält, der Rest dann die entsprechenden Texte.

Unicode, Fonts und Schriften

Die Gründer des Unicode–Konsortiums gingen davon aus, dass man den Ansatz für westliche Sprachen, für jedes Zeichen einfach einen Codepoint in einem Zeichensatz zu belegen, einfach so auf die Zeichensätze im Rest der Welt übertragen kann. Dabei würde ein einziger, wesentlich größerer Font entstehen, den man dann mit maximal 16 Bits statt nur 8 adressiert, und fertig.

> "Unicode gives higher priority to ensuring utility for the future than to preserving past antiquities. Unicode aims in the first instance at the characters published in modern text (e.g. in the union of all newspapers and magazines printed in the world in 1988), whose number is undoubtedly far below $2^{14} = 16,384$. Beyond those modern–use characters, all others may be defined to be obsolete or rare, these are better candidates for private–use registration than for congesting the public list of generally–useful Unicodes." — Joseph D. Becker, [1]

Die Quelle enthält übrigens eine Tabelle der Schriften nach Bruttosozialprodukt, und die ganzen indischen Schriften sind als „kommerziell irrelevant" eingestuft, weil Indien damals etwas über 1% des weltweiten GDP hatte ...

Die Realität ist natürlich komplexer. Die Alphabete der modernen Welt stammen praktisch alle von den altägyptischen Hieroglyphen ab; die Silbenschriften Ostasiens dagegen von der Orakel–Knochen–Schrift des antiken Chinas.

Zu den Alphabeten habe ich einen schönen Stammbaum gefunden (Abb. 1), der neben den Beispielen auch noch die unterschiedlichen Schreibrichtungen (meistens links nach rechts, teilweise rechts nach links, seltener von oben nach unten) angibt.

Diesem Stammbaum fehlt eine wichtige Information: Die meisten solchen Schriften sind Schreibschriften (nicht nur

die in Indien), die sehr viel mit Ligaturen[1] und diakritischen[2] Zeichen arbeiten, und daher nicht einfach so als traditioneller Computer–Zeichensatz benutzt werden können.

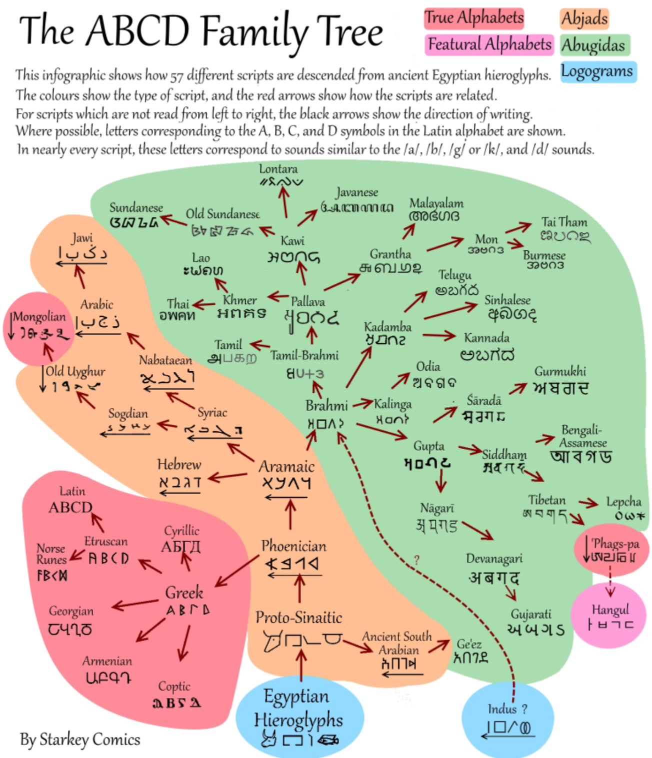


Abbildung 1: The ABCD Family Tree

Hier die mit MINΩΣ2 gerenderten Beispiele für Indien:

Devanagari	देवनागरी
Gurmukhi	ਗੁਰਮੁਖੀ
Meetei Mayek	ꯃꯤꯇꯩ ꯃꯌꯦꯛ
Bengali	অসমীয়া লিপি
Oriya	ଓଡ଼ିଆ ଅକ୍ଷର
Gujarati	ગુજરાતી લિપિ
Telugu	తెలుగు లిపి
Kannada	ಕನ್ನಡ ಅಕ್ಷರಮಾಲೆ
Tamil	தமிழ் அரிச்சுவடி
Malayalam	മലയാളലിപി
Tibetan	ཨོཾ་མ་ཎི་པདྨེ་ཧཱུྃ

Abbildung 2: Indische Schriften, links englische Bezeichnung, rechts im Original

Dabei fällt Tibetisch besonders aus der Reihe, denn dort werden die Ligaturen gleich auch noch senkrecht geschrieben, weshalb sich auffällig große Unterlängen ergeben können.

Zum Satz dieser komplexen Schriften gibt es Metadaten in den OpenType–Schriften, die man unter Verwendung der Library Harfbuzz relativ einfach nutzen kann. Zur Bestimmung der Schreibrichtung gibt es den Unicode–Bidi–Algorithmus, der eine relativ aufwendige Heuristik und Hilfsmarker benutzt, und den MINΩΣ2 ebenfalls implementiert. Da sich dieser Algorithmus mit jeder Unicode–Version etwas verändert, ist nicht gewährleistet, dass ein einmal richtig dargestellter Text auch in der nächsten Unicode–Version noch richtig ist; außer, man codiert die jeweiligen Schreibrichtungen fest über entsprechende Marker–Zeichen mit.

Was überhaupt nicht passiert ist, war die Vereinheitlichung der jeweiligen Fonts einer Sprache zu einem einzigen universellen Font. Die Codepoints der jeweiligen Fonts sind zwar jetzt Unicode, aber eben nur für eine Sprache. Man muss also zum Rendern eines Unicode–Texts zunächst die Schreibrichtung herausfinden, dann den sprachspezifischen Font auswählen, und dann mit Hilfe von Harfbuzz die Ligaturen und diaktritischen Marker setzen.

Für fernöstliche Sprachen ist das dann aber immer noch nicht gut. Japaner und Chinesen verwenden beide die in China Hanzi, in Japan Kanji genannten Zeichen. Die Zeichen haben aber verschiedene Varianten. So nutzt die Volksrepublik seit den 1950ern vereinfachte Zeichen, während Taiwan und Hong Kong traditionelles Chinesisch schreiben. Japanisch hat seine eigenen vereinfachten Zeichen (ebenfalls in den 1950ern als Shinjitai eingeführt), wobei nicht alle Vereinfachungen genau gleich durchgeführt wurden.

Unicode hat jeweils separate Codepoints für vereinfachtes und traditionelles Chinesisch. Dabei handelt es sich um die gleiche Sprache, und die Varianten sind lokale Besonderheiten und im Zuge der Vereinfachungen in Japan und China auch nicht auf einmal ersetzt, sondern in mehreren Abschnitten (die letzte in Japan ist von 2010); hier wäre es sinnvoller, der jeweilige Nutzer installiert sich einen Font nach seinem Geschmack, und die Zeichen werden dann halt entweder vereinfacht oder traditionell gerendert. Für besondere Zwecke (etwa Wörterbücher, die mehr als eine Form abbilden wollen) könnte man noch ein Metazeichen einführen, das definitiv auf traditionell oder simplified umschaltet. Damit könnten Festlandschinesen und Taiwaner sich problemlos schriftlich austauschen, ohne über die jeweils ungewohnte Darstellung der Zeichen zu stolpern.

[1] Eine Ligatur oder Buchstabenverbund bezeichnet in der Typografie die Verschmelzung zweier oder mehrerer Buchstaben einer Satzschrift zu einer Glyphe.

[2] Diakritische Zeichen sind an Buchstaben angebrachte Punkte, Striche, Häkchen, Bögen oder Kreise, die eine vom unmarkierten Buchstaben abweichende Aussprache oder Betonung anzeigen.

Bei Japanisch, das ja eine deutlich andere Sprache ist (auch wenn ein signifikanter Teil des Wortschatzes chinesische Fremdwörter sind, mit einer für einige südostchinesische Dialekte typischen Lautverschiebung), und in deren Satz sich auch eine andere Variante der traditionellen Kalligraphie im Drucksatz durchgesetzt hat, hat man einfach den Weg gewählt, alle Zeichen auf die ähnlichsten chinesischen Zeichen abzubilden (überwiegend simplified); die Varianten werden also nicht anders codiert. Man muss wissen, dass es ein japanischer Text ist, und dann entsprechend einen japanischen Font auswählen, dann passt das. Eine passende Heuristik dazu wäre, nach Katakana oder Hieragana zu suchen, die ja in rein chinesischen Texten nicht vorkommt. Allerdings sind besonders kurze japanische Texte, wie sie in UIs zu finden sind, aber auch Chat–Messages, oft komplett in Kanji, enthalten also keinen Anhaltspunkt, was für eine Sprache das sein soll.

Schriftgrad

Ein Unterthema ist hier noch die Schriftgröße. Traditionell bemisst sich der Schriftgrad nach der Höhe der im Bleisatz verwendeten Kegel, und gemäß dieser Tradition werden auch digitale Zeichensätze bemaßt. Das ist ein für den Benutzer weitgehend irrelevantes Maß; wichtig ist, dass die verwendeten Schriften bei einem typischen Betrachtungsabstand etwa gleich lesbar sind. Dazu setzt MINΩΣ2 chinesische Schriften, Emojis und Hieroglyphen größer als Alphabete (letztere erheblich größer). Dann passt das.

Wie hätte man das richtig machen sollen?

Alphabete, egal welches der über 50, sind eben Alphabete, d. h., die Zeichen sehen zwar je nach Region anders aus, haben aber grundsätzlich die gleiche Bedeutung. Der Codepoint sollte daher besser codieren, welches Zeichen gemeint ist, nicht, welche Schrift und Sprache (wobei natürlich Schriften Einzelzeichen enthalten können, die in anderen Schriften mit mehreren Zeichen umschrieben werden; die Zeichen sind aber auf jeden Fall phonetische Einheiten). Eigentlich würde das schon für unsere Groß– und Kleinbuchstaben gelten, die ja auch nur unterschiedliche Schreibweisen der gleichen Phoneme sind; aber die nutzen wir ja zusammen in einem Text.

Die jeweils verwendete Schrift ist eine dazu orthogonale Information, die wichtig ist, um den richtigen Zeichensatz, die Meta–Informationen für Ligaturen & diakritische Zeichen und die Schreibrichtung zu wählen, wobei die Schreibrichtung bei Zahlen dann auch noch abweichend sein kann (und dann ggf. eine Heuristik bei der Eingabe befolgt wird, die dann mit dem Text mit abgespeichert wird). Die Information über die verwendete Sprache ist z. B. für Trenn–Algorithmen wichtig, selbst wenn die Schrift an sich exakt gleich aussieht (also insbesondere für mittel– und westeuropäische Texte relevant, bei der sich nach der Abschaffung der Fraktur eine einheitliche Schrift durchgesetzt hat).

Eine entsprechende Codierung hätte also Zustände (bzw. sogar einen Zustands–Stack), und die Zeichen müssen auf jeden Fall kontextabhängig interpretiert werden. Das müssen sie aber schon aufgrund der Auswahl der Schreibrichtung, und damit ist das kein Verlust: Die korrekte Interpretation eines Texts ergibt sich aus dem ganzen String, nicht aus einzelnen Zeichen. Löst man einen Substring heraus, muss man die passenden Meta–Zeichen finden, die den korrekt beschreiben. Zeichensatz–Varianten wie fett oder kursiv (bzw. in anderen Schriften die Wahl vergleichbarer Kalligraphie–Stile) wären darin enthalten, und nicht Teil eines Markups auf einer anderen Ebene; auch, weil diese Auszeichnungen sonst auf überraschende Weise mit dem Bidi–Algorithmus interagieren.

Mein Vorschlag wäre, die bislang in UTF–8 nicht benutzen Codes $FC (+Bytes im Bereich $80..$BF) bis $FF zu nutzen, um entsprechende Metainformationen bezüglich Fontwahl und Schreibrichtung zu codieren, wobei $FF der Pop ist, und alle anderen Zustandsänderungen den vorherigen Zustand auf einen Stack werfen. Fehlende Pops am Ende des Texts sind kein Problem, da wird auf jeden Fall aufgeräumt; das Parsen von an sich ungültigen UTF–8–Codes sollte auch nicht zu viele Software aus dem Tritt werfen, denn die selbst–synchronisierende Eigenschaft von UTF–8 bleibt ja erhalten. Der im ursprünglichen Unicode–Entwurf vorhandene Teil von $F8..$FB wäre noch für Fünf–Byte–Sequenzen nutzbar, falls mal Bedarf nach so großen Code–Points entsteht. Wählt man eine Sprache aus, codieren die folgenden UTF–8–Buchstaben–Codepoints und Codepoints $\geq$ `$7F` dann nur noch einen Offset innerhalb der entsprechenden Code–Blocks; Control–Zeichen, westlich–„arabische“ Ziffern und Sonderzeichen bleiben erhalten, weil sie in vielen Schriften verwendet werden, in den Fonts auch vorhanden sind, teilweise bewusst in den benutzten Varianten, und dafür dann nicht unnötig umgeschaltet werden muss.

Letztlich wäre das kein „Unicode“ mehr, also keine Vereinheitlichung, sondern das Gegenteil: ein Localicode, bei dem insbesondere vereinheitlicht ist, wie man in die entsprechende Locale kommt und zurück.

Referenzen

[1] Joseph D. Becker, *10 Years of Unicode Standard 1988 to 1998,* `https://www.unicode.org/history/unicode88.pdf`

Forth–Gruppen regional

Bitte erkundigt euch bei den Veranstaltern, ob die Treffen stattfinden. Das kann je nach Pandemie–Lage variieren.

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@net2o.de`
Treffen: Jeden 4. Donnerstag im Monat um 19:00 auf `http://public.senfcall.de/forth-muenchen`, Passwort `over+swap`.

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://www.meetup.com/Essen-Forth-Meetup/`

Dienste der Forth–Gesellschaft

Nextcloud `https://cloud.forth-ev.de`

GitHub `https://github.com/forth-ev`

Twitch `https://www.twitch.tv/4ther`

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 58 46) – 98 04 00 8**$_p$
`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

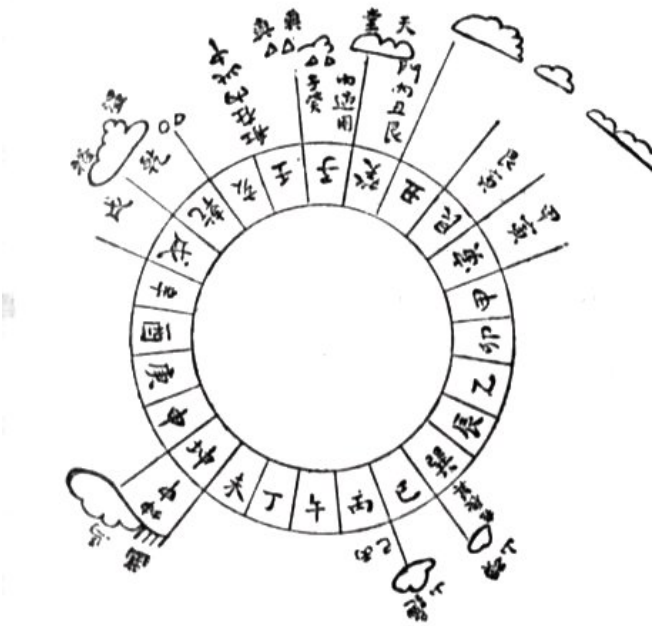
Donnerstags ab 20:00 Uhr
Forth–Chat net2o `forth@bernd` mit dem Key `keysearch kQusJ`, voller Key:
`kQusJzA;7*?t=uy@X}1GWr!+0qqp_Cn176t4(dQ*`

Jeder 1. Montag im Monat ab 20:30 Uhr
Forth–Abend
Videotreffen (nicht nur) für Forthanfänger
Info und Teilnahmelink: E–Mail an `wost@ewost.de`

Jeder 2. Samstag im Monat
ZOOM–Treffen der Forth2020 Facebook–Gruppe
Infos zur Teilnahme: `www.forth2020.org`

Forth–Tagung (online) 05.–07. Mai 2023
`https://tagung.forth-ev.de`

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.

Achtung: Geänderter Termin!
Forth–Tagung 2023 vom *05. bis 07. Mai 2023* (online)

Organisation: Vorstandskreis der FG

Auch 2023 wird die Forth–Gesellschaft ihre Tagung und die Mitgliederversammlung ***online*** *durchführen. Da dieses Format der Video–Konferenz in den letzten Jahren gut angenommen worden ist, setzen wir das mit dem schon bewährten BBB–System fort. Reisen entfällt — Besichtigungen von Klöstern und Landschaften, zusammen lecker Essen gehen und bis tief in die Nacht an der Bar fachsimpeln auch. Dafür ist es im Mitgliedsbeitrag enthalten, kostet nichts zusätzlich.*

Anmeldung

Auf `https://tagung.forth-ev.de` findet ihr weitere Informationen dazu, das aktuelle Programm sowie die Möglichkeit, euch zur Tagung elektronisch anzumelden.

Programm

Freitag, 05.05.2023

- Abends: Informeller Treff online ohne besonderes Programm;
  ab 19:00 Uhr

Samstag, 06.05.2023

- Vormittags: Vorträge und Workshops
- Nachmittags: Vorträge und Workshops
- Abends: Gemeinsames Abendessen (Online–Edition)

Sonntag, 07.05.2023

- Mitgliederversammlung der Forth–Gesellschaft e.V.;
  10:00 — 13:00 Uhr
- Platz für weitere Workshops;
  15:00 — 18:00 Uhr

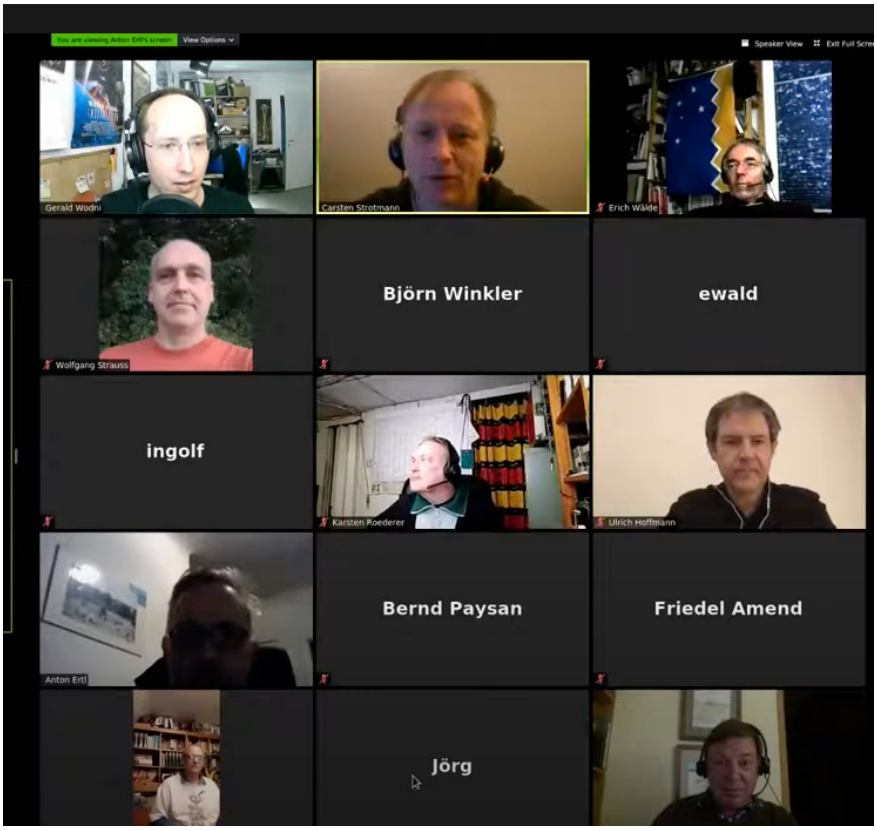


Abbildung 1: Neulich sah das dann so aus wie auf diesem Bild.
Probiert schon mal eine gute Beleuchtung und einen hübschen Hintergrund für euch aus.