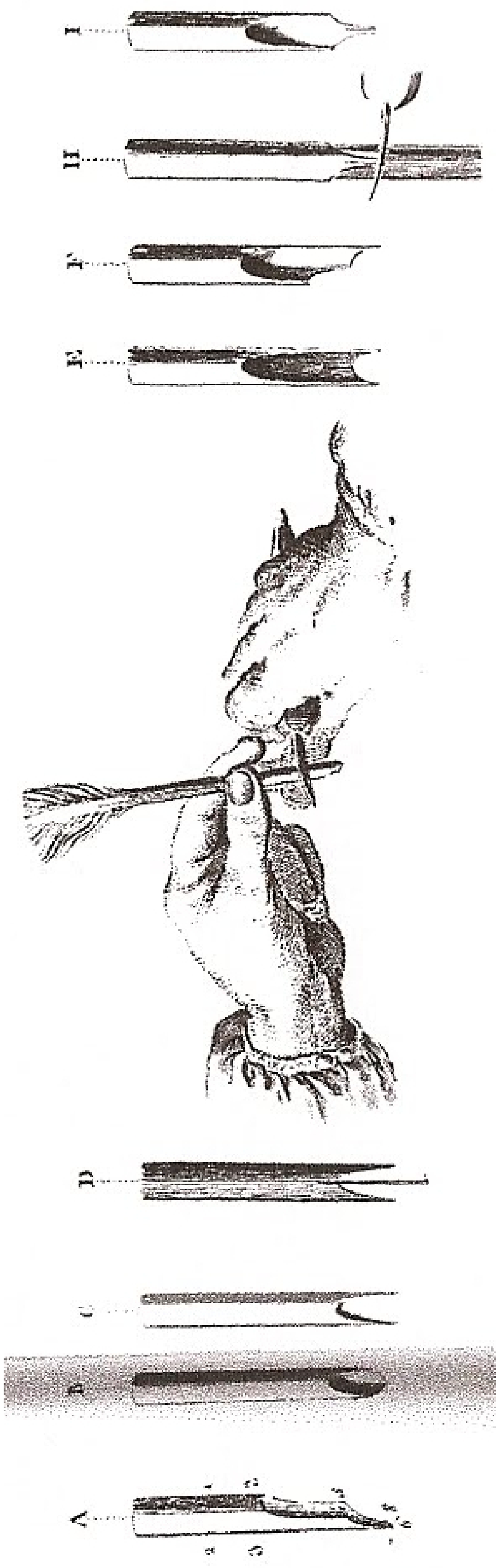
Vierte Dimension

2023/1

The Forth Magazine

for science and technology, for commercial EDP, for ICE–technology, for the passionate hobbyist



In this Issue:

Services and Products

Fahrtregler - Lichtanlagen - Soundmodule - Modellfunk

tematik GmbH
Technische Informatik

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

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

Forth Trainings

Do you want to learn the Forth programming language or get trained in the new Forth developments? Do you have products based on Forth and would like to train employees in the maintenance and further development of these products?

We offer training in legacy Forth systems (FIG–Forth, Forth83), ANSI–Forth and the latest Forth 200x standards. Our trainers have over 20 years of experience with Forth programming on embedded systems (ARM, MSP430, Atmel AVR, M68K, 6502, Z80 and many more) and on PC systems (Linux, BSD, macOS and Windows).

Carsten Strotmann carsten@strotmann.de
`https://forth-schulung.de`

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

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

Cornu GmbH
Ingenieurdienstleistungen
Elektrotechnik

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

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

KIMA Echtzeitsysteme GmbH

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

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

FORTecH Software GmbH

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

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

Ingenieurbüro
Klaus Kohl–Schöpe

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

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

Microcontroller Pool
Forth–Gesellschaft e. V.

We provide high quality evaluation boards, also FPGA, including Forth systems: Cypress, RISC–V, TI, GA144, MicroCore, SeaForth, MiniMuck, Zilog, Motorola, 68HC11, ATMEL, Hitachi, Renesas, Lego …
`https://wiki.forth-ev.de/doku.php/mcv:mcv2`

Cover picture: Carving quills AUTHOR: Sketch processed by mk

Source: Fetched from the internet

Dear Readers,

our annual Forth meeting is coming up! Soon it will be finally time again. On the back page is explained how you can participate. You have probably noticed: The date planned before Christmas in the last issue has been postponed. The meeting has been moved to May.

No need to travel, we'll stick with the video conferencing format. Why? Because now we are able to. After the quill pen and riding messengers came letterpress, letter mail, typewriter, keyboard, and now microphone and camera — the videophone made arguably its first appearance in film history in FRITZ LANG'S *Metropolis*, now we're doing videoconferencing.

The simplified letters of the initial terminals are already *retro* again. Writing as an art has prevailed. Computer fonts and handwriting are converging again. And the tools around Forth are not standing still either. If Forth in its smallest form on MCUs sometimes seems like a quill, the large Forth systems are far ahead in progress. But the tools for the smallest ones are catching up.

INGOLF POHL has also taken up this theme. What do you think of his *FEE*? It is of course tailored to his projects — *Embedded* with Mecrisp. I am curious about the response. And so that you can get Mecrisp into the target as well, the tool will do that for you too.

ANDREW READ takes us into the world of astronomy and its great modern possibilities. Did CHARLES MOORE, when developing Forth, see that it would be possible to use such professional equipment for celestial exploration from home?

Of course, such telescope control systems require solid and reliable hardware. RAFAEL DELIANO again lets us participate in such developments, which are robust and adapted to the purpose. Of course, it is important that the components are also available. Let's hope it stays that way. Last year was not so good — you remember: supply shortages.

Finally, BERND PAYSAN describes how far the representation of quite different writings of the world has already progressed in his MINΩΣ2. And thus follows up on his report from the last issue. Here we are finally away from the quill — although who knows? Maybe I'll be around to see a Gforth humanoid send me a handwritten letter, written with a self–carved swan–wing quill.

Last but not least, even I was able to contribute something Forthy this time, because I was allowed to explore SPI. DIRK BRÜHL was my sponsor. He provided the components and the board. Many thanks for that.

[Translated to English by Wolfgang Strauß]

You do not have to manually type the source codes in the VD. You can easily download them from the web page of the association.
`http://fossil.forth-ev.de/vd-2023-01`

Imprint

Title of the Magazine

Vierte Dimension

Publisher

Forth–Gesellschaft e. V.
P.O. Box 1030
48481 Neuenkirchen
Germany
Email: `Secretary@forth-ev.de`
`Direktorium@forth-ev.de`
Bank account: Postbank Hamburg
BLZ 200 100 20
Kto 563 211 208
IBAN: DE60 2001 0020 0563 2112 08
BIC: PBNKDEFF

Editorial & Layout

Bernd Paysan, Ulrich Hoffmann
Email: `4d@forth-ev.de`

Ad Management

Office of the publisher

Editorial Deadline

January, April, July, October each in the third week

Publication Frequency

1 issue per calendar quarter

Unit Price

4,00 € plus postage and packing

Manuscripts and Rights

All manuscripts sent in will be considered. Letters to the editor may be reproduced without consultation. For articles marked with the name of the author, the editorial team only accepts responsibility under press law. The articles published in this magazine are protected by copyright.
Translation, reproduction, as well as storage on any media, in whole or in part, is only permitted with exact reference to the source. The submitted contributions must be free from claims of third parties. Published programs go — as far as nothing else is noted — into the public domain.
No liability can be accepted for text, circuit diagrams or assembly sketches which lead to the non–functioning or possible damage of components. All publications are made without consideration of possible patent protection. Trade names are used without any warranty of free use.

Figure 1: Chris Hay

Why Forth Language is Important

Chris Hay (fig. 1) recently posted an initially unassuming video on YouTube titled: "Introduction to Forth Programming Language — Tutorial for Beginners!" But he doesn't leave Forth standing in and of itself as it so often does. He impressively shows the usefulness of such concepts as Forth using the example of Bitcoins.

So his introduction is reproduced here in full. Well done, I think.

> "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."

There is a division into chapters that can be called up specifically. So you don't have to go through the 44–minute clip from beginning to end. Pick out what you're particularly interested in. 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) ..."

The friends of vintage computing should be happy. Guillaume Tello published version 0.5.4 on his website at the end of January this year. His Forth, which is completely specialized for these machines, can now do even more:

- 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

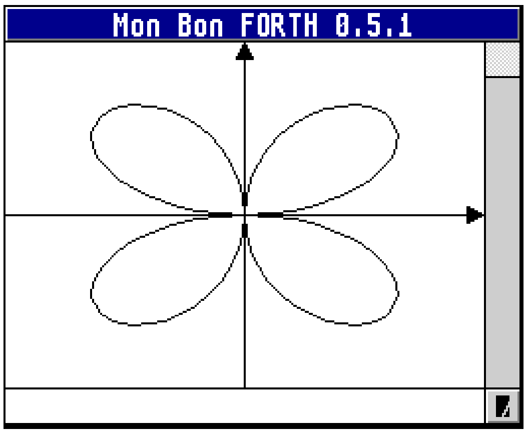


Figure 2: Polar curve r(t)

His website shows quite a few Forth examples, e.g. the 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 (fig. 2)

By the way, a 3D Math is already included.

His Forth is comfortable, just not dependent on foreign editors and has many helps, which make the simple dealing with it possible.

The Forth manual, a PDF with 219 pages, covers everything about Forth and the operating system of the machines itself.

Congratulations, great job! cas/mk

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

From Gforth to Lichen

While we're on the subject of editing and creating something presentable:

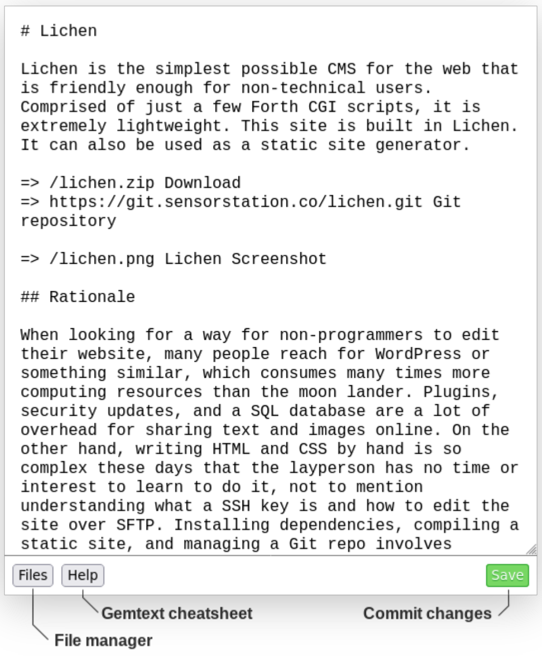
> "Lichen is the simplest possible CMS[1] for the web that is friendly enough for non–technical users. Comprised of just a few Forth CGI scripts, it is extremely lightweight . . ."

And what do you need for Lichen?

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

The Lichen web site itself is written in Lichen, so it's an example of the ease with which you can do this kind of thing — very elegant, in my opinion. cas/mk

[1] Content Management System



```
# Lichen

Lichen is the simplest possible CMS for the web that
is friendly enough for non-technical users.
Comprised of just a few Forth CGI scripts, it is
extremely lightweight. This site is built in Lichen.
It can also be used as a static site generator.

=> /lichen.zip Download
=> https://git.sensorstation.co/lichen.git Git
repository

=> /lichen.png Lichen Screenshot

## Rationale

When looking for a way for non-programmers to edit
their website, many people reach for WordPress or
something similar, which consumes many times more
computing resources than the moon lander. Plugins,
security updates, and a SQL database are a lot of
overhead for sharing text and images online. On the
other hand, writing HTML and CSS by hand is so
complex these days that the layperson has no time or
interest to learn to do it, not to mention
understanding what a SSH key is and how to edit the
site over SFTP. Installing dependencies, compiling a
static site, and managing a Git repo involves
```

Figure 3: Screenshot of the Lichen source code editor

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

busy? oder ready?

How should one turn the logic? This question came up recently when I was exploring an external Flash via SPI. Wolfgang Strauss, who saw my SPI article (further back in this magazine), said that if you take the phrase

`begin busy? until`

literally, it would not make sense, even if it worked. What had happened in my Forth code? Let's turn this back to the beginning: What was that again with the BUSY flag?

In the datasheet of the Winbond SPI–Flash it is explained what the bits in its *status register* mean, especially the 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

Table 1: Flash status bit S0

The higher bits also have meanings, but we will ignore them for now.

So before you can do anything with the Flash, you have to get its status.

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

There it is on the stack now.

As long as S0 is set, you have to wait. If one interprets the set bit S0 as a true flag, one would have to formulate in pseudo code:

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

Alright, extract the `flag` from the status byte:

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

Because `1` does not necessarily equal `true` in Forth, you test for “different from zero” and get your `true`.

The waiting loop then reads in Forth:

```
begin busy? while repeat
```

Or you test for `0=` and get the inverted logic. Now S0 = 0 is `true` and the pseudocode reads:

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

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

I kind of like the “positive” waiting loop better. What do you think?

```
begin ready? until
```

Like the traditional one:

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

mk

Spotlight: Memory Accesses with Different Word Widths

The world of 16–bit Forth systems was simple. To access bytes or characters, there were `C@` and `C!`, to edit cells — 16 bits at that time — there were `@` and `!`. But the time of 16–bit systems is gone and now there are 32– and even 64–bit systems. How do you deal with the 16–bit and 32–bit words in these systems? One possibility is the definition of numerous additional memory operators. At least one should be able to agree on suitable names. Here’s an overview of the naming schemes so far.

Memory Operators in 32–Bit Systems

On a 32–bit system, you typically have three word widths to work with: bytes (8 bits), halfwords (16 bits) and longwords (32 bits).

[2] However, VfX [2] has the operator `c@s` for sign extension.

8–Bit Bytes

Here the traditional access is done with `c@` and `c!` and the *bytes* are considered unsigned, therefore no sign extension takes place.[2]

16–Bit (Half) Words

Such accesses are named inconsistently, but it has become common to call this address bit width *word* and to use operators with `W` as a prefix for it. (Half) words are considered to be unsigned or signed. Therefore, there are read accesses with or without sign extension; tab. 2 shows some examples. Others do it differently.

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!`

Table 2: Some 32–bit systems and their halfword memory operators

32–Bit (Long) Words

64–bit systems also offer 32–bit accesses, but these accesses are also named inconsistently. But it has become common to call them *longwords* and to use the `L` prefix for operators used for 32 bits. Longwords are considered unsigned or signed.

Forth	unsigned fetch	singned fetch	store
Open Boot	`l@`	`l@`	`l!`
Gforth	`ul@`	`sl@`	`l!`
Swiftforth	`l@`	`l@s`	`l!`
VFX–Forth	`@`	(no symbol)	`!`

Table 3: 32–bit and 64–bit systems and their 32–bit memory operators

64–Bit Words

The 64–bit systems of course also access the memory on a 64–bit basis. These operators have the prefix `X`. But memory cells of the (data) stack width are typically processed only with `@` and `!` *without* prefix. These prefixless `fetch` and `store` are then at the same time the matching aliases to `w@ w!` (16–bit system), `l@ l!` (32–bit system) or `x@ x!` (64–bit system).

Outlook

The memory operators are in the current standardization discussion.

Using this zoo of operators is highly unpleasant.

In seedForth (32 bit on PCs) there is no halfword support. There are only `c@ c!` (unsigned) and `@ !` (32 bit unsigned or signed, but irrelevant). But you don't have to save memory there anyway.

The Forth community goes in the direction of *Values* and *Value Flavoured Structs* (structures whose fields are like Values, see below). For smaller memory sizes you then have `cValue` and `uwValue swValue`, `ulValue uxValue`, etc., if any.

Data is placed on the stack by `value` taking care (with or without sign extension). Writing access is always done via `TO`, which of course internally uses the appropriate operator.

The application programmer is thereby relieved and does not have to pick the correct operators constantly. The readability and maintainability is increased too.

In a prototypical implementation of Value Flavoured Structs an example struct would be defined like this:

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

`int8:` etc. are defining words for fields of matching size (and type).

The field names (here `u8 s8` ...) then perform the offset calculation and reading of the value: e.g. reads

```
S s16
```

the field `s16` with sign extension (because of type `sint16`) and puts the value on the stack.

It can be set e.g. with

```
-42 S TO s16
```

In this case, the most significant bits above bit 15 are truncated; analogously for all other fields.

And is there also `SEX` ?[3]

This then must be able to do an 8–bit and 16–bit sign extension, so `cSEX` and `wSEX`, or how? Or one ignores the memory requirement and stores everything in 32 bits. And is that acceptable for a small system?

[3] SignEXtend (taken over from 6809)

Such questions dissolve into thin air if one works object–oriented. *Values* and *Value Flavoured Structs* are an intermediate way, but also need a polymorphic `TO`. But this would be comparably easy to implement.

With Value Flavoured Structs there are no other methods (maybe `+TO` and `ADDR`, if you absolutely like to have them).

Objects are the logical progression.

But for a standardization we would need a *best practice* supported by the Forth community. But this is not to be seen. What can it look like?

Ulrich Hoffmann

ESP32forthStation

Zoom, December 10, 2022 — that's where I introduced the *ESP32forthStation*. It's a standalone single board Forth computer with WLAN networking capabilities. It has ports for a keyboard (PS/2), a video monitor (VGA), communicates via USB or WLAN, and can be used as a full development platform for experiments.

Based on open source projects, the small *TTGO VGA32* board from *LilyGo* houses the complete ESP32forthStation. And the ESP32forth inside it now works with the *FABGL* and *ESP32forth* libraries made by Fabrizio Di Vittorio and Brad Nelson. The code is licensed under the terms of the GNU GENERAL PUBLIC LICENSE.

The ESP32forthStation can be flashed via the *Arduino IDE*.

Ulrich Hoffmann

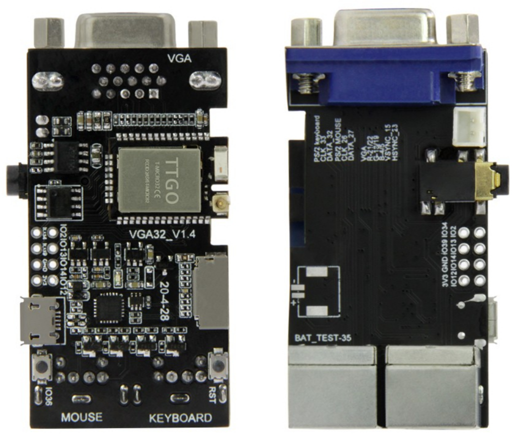


Figure 4: TTGO VGA32 board from LilyGo

More information and contact with the author can be found on the website of his project:

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

[Translated to English by Wolfgang Strauß]

FEE — Forth Enhanced Editor

Ingolf Pohl

In the last issue (VD2022–04) Wolfgang Strauss *reported about the project Feuerstein and mentioned there also my editor FEE (Forth Enhanced Editor). I would like to describe here how the idea came up to develop an own tool for the communication with a microcontroller target and how to work productively with the tool.*

My Queries to the Forthers

I started programming Forth because it allows interactive programming of a target system. You communicate with the compiler in the target system through a terminal — wait, through a terminal? What about the code you just sent to the compiler, where did it go? It's gone? It can't be! I asked a few Forth programmers at Project Feuerstein and got answers like, "What's your problem? You write in your favorite editor, Emacs, and then copy the code into the terminal window". Copy? From the editor to the terminal? Possibly line by line, because the timing is not right? No, I would not have expected that at all . . .

Background

I'm already spoiled by another stack–oriented and interactive programming language/environment — Fifth. Yes, you wouldn't believe it, in the 80s an old professor of mine developed a very Forth–like programming language.[1] It was also used for programming heterogeneous multiprocessor systems (controllers, DSPs, transputers etc.) for multibeam echo–sounders. The thing ran on the PC under DOS; a system for X86, C167 and ADSP2181 was barely 160 KB in size and connected to the target system via serial interface. Special was that the crosscompiler was controlled by its integrated editor. Programming was quite simple and interactive:

1. Write a piece of source code.
2. Press the "Try" button to try it out: the compiler compiles the code, transfers it to the target system and executes it there, outputs are displayed in the editor window.
3. The source code remains in the editor window; you can improve it, change it, discard it or you decide it's fine.
4. When the source code is ready, press the compile button: the compiler compiles, stores the code in the target system and extends the dictionary in the compiler. The new words can now be called from new code or from the PC. The source code is put on the stack (where else?) with compiled source code.

Also the source code management was actually quite simple — only the one, small DOS window was a hindrance. There were three text areas, of which, unfortunately, you could only see one at a time:

Area A: Finished source code — what you had compiled and was fine

Area B: Sourcecode under construction — what you write and try out right now

Area C: Forthcoming code — code that is not yet compilable, test code

I would be delighted to have something like this for Forth as well . . .

An Interactive Editor is Needed

I would like to have such an editor for Forth source code, with at least three text areas that are also editable in a normal editor:

```
--- start of text ---------------
  mature, already compiled code
--- text border -----------------
  code just to be compiled
--- text border -----------------
  code not yet compiled
--- end of text -----------------
```

- A button to have a try: current text is sent to the target system
- A button to compile: current text is sent to the target system and then pushed to the upper area

The idea is to build the source code within an editor window, and then easily transfer the code to the target machine. The inputs and outputs for controlling the program should be handled in a terminal window. I want it to be an editor with an extra terminal window, not a terminal with an editor window!

Quickly Fiddle Something in Python

I am not the most experienced PC programmer. So I need something that makes the PC and its windows easily accessible to me. It would also be great if every Feuerstein member could use the editor under his currently running environment. So I came to Python 3 with `tkinter` and `pyserial`. A text window with ready to use editing functions is quickly set up. The serial interface directly, via USB or even via Telnet can be easily accessed. I would have liked to do it in Forth, but did not find a suitable system for me.

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

The editor works according to the principle described above. From the terminal the editor is called and appears in its own window. In the editor window, text that sits between two text boundaries (stoppers) can be sent to the target system in the desired two ways:

- `ctrl+enter` to try things out
- `alt+enter` for final compilation

The source code to be sent is the text area between two stoppers where the cursor is currently placed.

Of course, you can work with only one text area or with any number of text areas, so text start and text end are also text boundaries.

The terminal window from which the editor was started acts as the terminal window for the input/output of the target system.

Working With the FEE Editor

I personally work with FEE and Mecrisp on a GD32VF103 board as follows and divide the source code by stoppers into at least 4 areas, better more. Thereby I use the ability of Mecrisp to compile either into Flash or RAM:

Area 1: Comments and `eraseflash` for quick access

Area 2: Code of great trust — goes into Flash

Area 3: Code of little trust — goes into RAM

Area 4: Code being worked on — also uses RAM

Areas >4: Ideas, test functions, drafts ...

If I have a new system or a completely messed up Flash, I simply run sections 1 to 3 in sequence using the "trial function" and quickly have my development status back.

The `eraseflash` is in area 1 because Mecrisp resets afterwards and does not return an "`ok.`". Otherwise you could execute everything in one go.

In area 2 with the "code of great confidence" are often only a few include statements that load proven code from external files into Flash.

Area 3 with the "low confidence code" loads into RAM.[2] This code is either not yet ready for Flash or is only used for development.

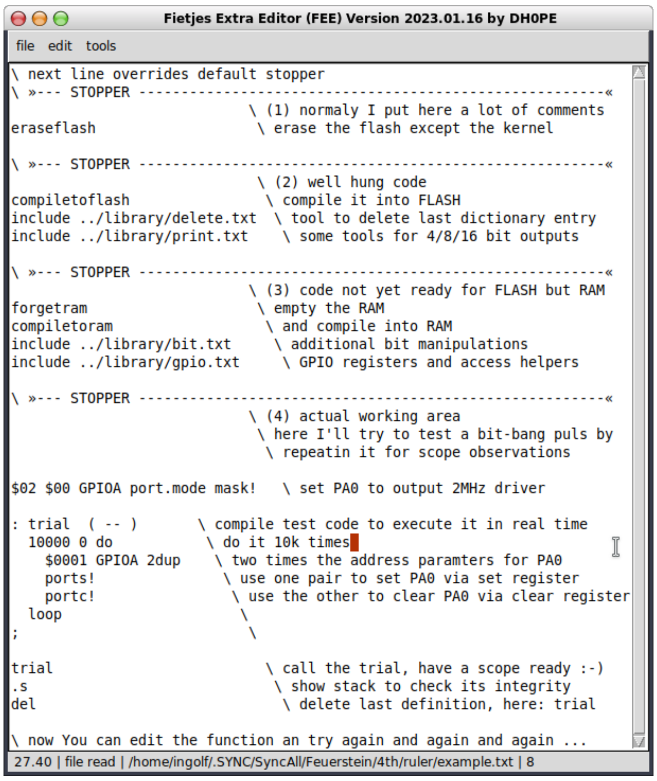
My current workspace is in area 4 — of course, it can also be in another area, depending on the developer discipline. The code in this block is edited, tried, edited again, tried again ... until I am satisfied with it. "Trying" is ideally done simply by hotkey `ctrl+enter`. And when the code is ready, I move it to the text area above with a final compilation via hotkey `alt+enter`.

When trying, I like to use `forget` or `del` from Matthias Koch. In the example I use the `del` method, which deletes the last compiled word from the dictionary and thus prevents the dictionary from getting bigger and bigger and eventually overflowing.

Higher areas I use as storage for test code, ideas, alternative approaches, comments. Often I already design the top–down code there, while in area 4 its counterparts grow bottom–up.

When I'm done, I have a large block of text that is compiled in one pass into the final memory areas of the target system. I can then store this as source code together with the include files and keep it.

Figure 1 shows a short example with about 40 lines of code, as it may actually look like in my work.



```
\ next line overrides default stopper
\ »--- STOPPER -------------------------------------------------------«
                            \ (1) normaly I put here a lot of comments
eraseflash                   \ erase the flash except the kernel

\ »--- STOPPER -------------------------------------------------------«
                             \ (2) well hung code
compiletoflash                \ compile it into FLASH
include ../library/delete.txt  \ tool to delete last dictionary entry
include ../library/print.txt    \ some tools for 4/8/16 bit outputs

\ »--- STOPPER -------------------------------------------------------«
                            \ (3) code not yet ready for FLASH but RAM
forgetram                    \ empty the RAM
compiletoram                  \ and compile into RAM
include ../library/bit.txt     \ additional bit manipulations
include ../library/gpio.txt     \ GPIO registers and access helpers

\ »--- STOPPER -------------------------------------------------------«
                            \ (4) actual working area
                             \ here I'll try to test a bit-bang puls by
                              \ repeatin it for scope observations

$02 $00 GPIOA port.mode mask!   \ set PA0 to output 2MHz driver

: trial  ( -- )       \ compile test code to execute it in real time
  10000 0 do           \ do it 10k times
    $0001 GPIOA 2dup    \ two times the address paramters for PA0
    ports!               \ use one pair to set PA0 via set register
    portc!                \ use the other to clear PA0 via clear register
  loop                     \
;                           \

trial                         \ call the trial, have a scope ready :-)
.s                             \ show stack to check its integrity
del                             \ delete last definition, here: trial

\ now You can edit the function an try again and again and again ...
```

```
27.40 | file read | /home/ingolf/.SYNC/SyncAll/Feuerstein/4th/ruler/example.txt | 8
```

Figure 1: Example of all aspects of interactive work

The areas are marked with `(1)` to `(4)` for clarification. I mess around in the source code of the word `trial` and try it with `ctrl+enter`. I do this until I am satisfied. Due to `del` the dictionary stays without overflow when trying repeatedly. For the final compile, the helper lines with the preparations are removed — here the configuring of the port, the function call, the stack inspection and the `del` statement.

What Else Does the Editor Offer?

Relatively quickly one after the other, I added a few useful features to the system.

- There is an `include` to load external source code.
- Optionally, comments and blank lines can be omitted when transferring.

[2] The ability to compile into Flash or RAM is a special feature of Mecrisp–Forth.

- Text stoppers can be inserted via hotkey.
- A hardware reset via the DTR[3] line can be triggered by hotkey.
- Transmission of a single line can be initiated by hotkey.
- The transmission is tailored to Mecrisp.
- There is a magic textblock–to–file conversion with `include`–paste.
- “Empty” GD32VF103 or STM32F103 chips, which do not yet contain Mecrisp, can have the Forth uploaded via the serial interface.
- The baud rate for communication can be switched by a special word in the comment during compilation.
- Before critical operations, the editor saves a timestamped copy in the subdirectory ./sav/
- Optionally, you can have the output of the target system written to a log file.
- Now there is also an undo/redo function.

In the appendix you will find a list of functions and assigned hotkeys. By the way, they only work in the active editor window and therefore do not interfere elsewhere.

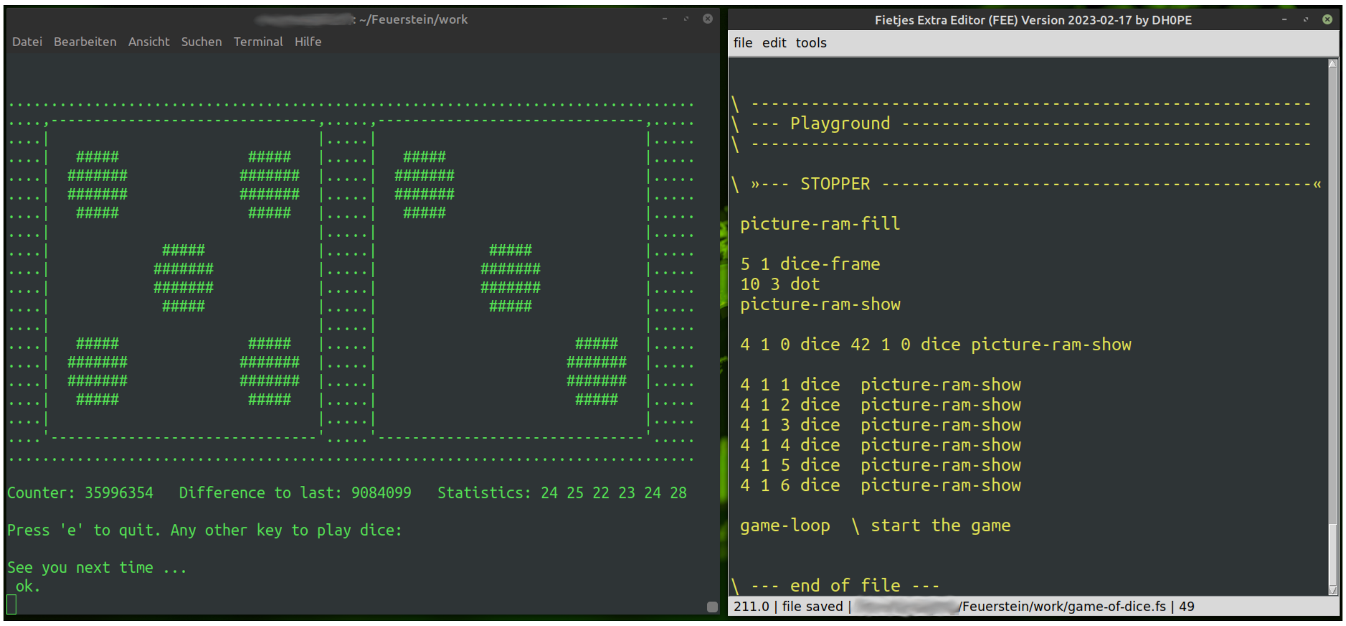
What Does FEE Stand For Anyway?

Originally, the TLA (Three Letter Acronym) FEE stands for “Forth Enhanced Editor”. But there’s no fun in that, it’s much too formal. With Feuerstein we thought that a girl/boy called “Fietje” is such a “Feuerstein”, which deals with the Forth. Therefore the F can also stand for Fietje or Feuerstein. The second E then stands for extended, excellent, elegant, elaborate, etc. The last E remains for editor — I would not call it an environment without a built–in compiler.

Where to Get It From?

You can find the project page of FEE at
`https://forth-ev.de/wiki/projects:fee:start`

[Translated to English by Wolfgang Strauß]



[redacted]: ~/Feuerstein/work
Datei Bearbeiten Ansicht Suchen Terminal Hilfe
Fietjes Extra Editor (FEE) Version 2023-02-17 by DH0PE
file edit tools

```
................................................................................
....,-------------------------------,.....,-------------------------------,.....
....|                               |.....|                               |.....
....|    #####             #####    |.....|    #####                      |.....
....|   #######           #######   |.....|   #######                     |.....
....|   #######           #######   |.....|   #######                     |.....
....|    #####             #####    |.....|    #####                      |.....
....|                               |.....|                               |.....
....|             #####             |.....|             #####             |.....
....|            #######            |.....|            #######            |.....
....|            #######            |.....|            #######            |.....
....|             #####             |.....|             #####             |.....
....|                               |.....|                               |.....
....|    #####             #####    |.....|                      #####    |.....
....|   #######           #######   |.....|                     #######   |.....
....|   #######           #######   |.....|                     #######   |.....
....|    #####             #####    |.....|                      #####    |.....
....|                               |.....|                               |.....
....'-------------------------------'.....'-------------------------------'.....
................................................................................

Counter: 35996354   Difference to last: 9084099   Statistics: 24 25 22 23 24 28

Press 'e' to quit. Any other key to play dice:

See you next time ...
 ok.
```

```
\ -----------------------------------------------------------
\ --- Playground --------------------------------------------
\ -----------------------------------------------------------

\ »--- STOPPER ------------------------------------------------«

 picture-ram-fill

 5 1 dice-frame
 10 3 dot
 picture-ram-show

 4 1 0 dice 42 1 0 dice picture-ram-show

 4 1 1 dice  picture-ram-show
 4 1 2 dice  picture-ram-show
 4 1 3 dice  picture-ram-show
 4 1 4 dice  picture-ram-show
 4 1 5 dice  picture-ram-show
 4 1 6 dice  picture-ram-show

 game-loop  \ start the game


\ --- end of file ---
211.0 | file saved | [redacted]/Feuerstein/work/game-of-dice.fs | 49
```

Figure 2: FEE in action, color scheme “Dark Mode”. Side by side the terminal window (left) and the editor window (right). In the editor you can see the test code for a simple dice game. Each line can be sent individually to the Forth system with `shift+enter`. By the way: FEE can be customized by setting parameters when called. A sample shell script can be found in Appendix E.

Appendices

Appendix A: File Functions, Menu “File”

`ctrl+o`	= Open a file
`ctrl+r`	= Reload file, overwrite editor content
`ctrl+s`	= Save file
`ctrl+shift+s`	= Save file as …
`ctrl+shift+w`	= Enable/disable log file function
`ctrl+shift+q`	= Quit, close editor window

Appendix B: Editor Functions, Menu “Edit”

`ctrl+z`	= Undo last action
`ctrl+shift+z`	= Redo last action
`ctrl+x`	= Cut selected text to clipboard

[3] Data Terminal Ready (handshake signal of the serial interface)

`ctrl+c` = Copy selected text to clipboard

`ctrl+v` = Paste selected text from clipboard

`ctrl+l` = Cut entire line of text to clipboard

`alt+l` = Copy entire line of text to clipboard

`ctrl+v` = Paste entire line of text from clipboard

`shift+enter` = Try line — send current line to target

`ctrl+enter` = Try block — send current block to target

`alt+enter` = Submit block — send current block to target and push up

`ctrl+j` = Insert stopper–flag line

`alt+j` = Export current text block, replace with `include`

`ctrl+shift+m` = Show/hide whitespace

Appendix C: Support Functions, Menu "Tools"

`ctrl+q` = Clear content of the terminal window

`alt+q` = Hardware reset of target system

`no hotkey` : Flash target system with binary file

Appendix D: Some Important Hotkeys of "tkinter"

`ctrl+x` = Cut selected text to clipboard

`ctrl+c` = Copy selected text to clipboard

`ctrl+v` = Paste selected text from clipboard

`ctrl+y` = Same as `ctrl+v`

`ctrl+b` = Move cursor back one character

`ctrl+f` = Move cursor forward one character

`ctrl+p` = Move cursor to previous line

`ctrl+n` = Move cursor to next line

`ctrl+a` = Move cursor to start of line

`ctrl+e` = Move cursor to end of line

`ctrl+i` = Insert tabulator

`ctrl+d` = Delete character at cursor

`ctrl+h` = Remove character to the left of cursor

`ctrl+k` = Kill rest of line

`ctrl+t` = Swap the two chars to the left of cursor

Appendix E: Shell Script to Start FEE Easily

```
#!/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
```

Flashing Forth

Ingolf Pohl, Wolfgang Strauß

In the article about FEE (see article "FEE - Forth Enhanced Editor" in this issue) it was mentioned that you can easily load a Forth runtime system, such as Mecrisp, from FEE into the blank target system. Here we would like to describe the procedure using the Fietje miniboard as an example, which runs with a GD32VF103. Of course the procedure also works with other GD32VF103 boards like the Longan Nano. The STM32F103 series is supported as well.

Basics

Many modern microcontrollers have their program memory right on the chip. The common memory technology is Flash[1]. The manufacturer delivers his devices initialized, i.e. all bits of the Flash are set to "1". Before the Forther can work with the microcontroller in the usual interactive way, a binary file containing a suitable Forth system must be "flashed" into the device. In the case of the GD32VF103 from GigaDevice considered here, flashing a binary file can be done in different ways:

- Via the JTAG interface (JTAG: Joint Test Action Group). Special debug hardware is required for this
- Via SWD (Serial Wire Debug), also requires special debug hardware
- By boot loader using the chip–internal USB hardware module via DFU protocol (Device Firmware Upgrade)
- By boot loader via the on–chip serial interface (UART)

The boot loader mode via UART is a good choice, because we use the required USB–serial converter for the communication with the Forth after flashing anyway.

Start–up Variants

The GD32VF103 supports three start–up variants, one of which is selected by the levels of two special pins (BOOT0, BOOT1) after a reset of the chip. Tab. 1 lists the variants.

Boot–Pin Level		Start–up Variant
BOOT0	BOOT1	
`low`	`don't care`	Flash
`high`	`low`	Boot Loader
`high`	`high`	RAM

Table 1: Start options

Now the more detailed description of the individual variants follows:

- *Flash:* This is later the normal operation. Here the CPU of the microcontroller starts the execution of the program in the flash memory. Of course a suitable firmware (Forth) must have been stored (flashed) in the flash before.
- *Boot loader:* This variant is exactly what we need for flashing. It starts the so called boot loader. The boot loader is software that the manufacturer of the chip has placed in a special memory area. This software is read-only, so it cannot be overwritten accidentally or intentionally.
- *RAM:* Here the CPU would start code from RAM. This variant is not very useful for us, because the contents of the RAM will be random after a cold start[2]. This would give a nice crash.

The Boot Loader

Ok, now we are quite a bit further. So we want to use the boot loader to persuade the chip to accept a binary file via a normal serial port and write it to its Flash. But how does the boot loader do that exactly?

The chip tests its pins BOOT0 and BOOT1 directly after a reset. If levels like in tab. 1, line 2 result, the execution of the boot loader program starts. But we had that already.

The first thing to check is on which "channel" data arrives. Channels can be the USB port or a serial interface (UART). Here we shall assume that data arrives via a UART. After the first character is received, its time length is determined and the appropriate baud rate is calculated and set. Now everything is ready for an interactive communication with the other end, in our case FEE.

The boot loader has among other things functions for erasing, writing and reading the flash and for determining the chip ID and protocol version. With this it is possible to work comfortably.

If you would like more details, please have a look at the *application note* "AN3155", which can be found at `https://www.st.com`

Bill of Materials

For the first loading of a Forth system with FEE and afterwards of course for interactive development you need

- a PC with Python 3 installed, FEE (Forth Enhanced Editor)

[1] Flash: memory that does not lose the information even after the supply voltage is switched off

[2] Cold start: Start–up of the system after switching on the supply voltage

- a USB–serial adapter (TTL level and VCC at 3.3 V); it is recommended to use a device with the handshake lines DTR and RTS. With this, FEE can execute the flashing process automatically. But a “normal” adapter with only RxD and TxD works too.
- a target system with the microcontroller GD32VF103
- a binary file with a Forth system (Mecrisp Quintus) for the GD32VF103

Wiring Up the Fietje Board

For a comfortable boot mode switching you need the connections according to tab. 2 between the USB serial converter and Fietje Miniboard:

USB Converter	Data Flow	Fietje Miniboard
GND	—	GND
VCC (3,3 V)	—	3V3 (3,3 V)
TxD	→	RxD
RxD	←	TxD
DTR	→	RSTN
RTS	→	BOOT0

Table 2: Wiring

Fig. 1 shows the setup in a graphic form:

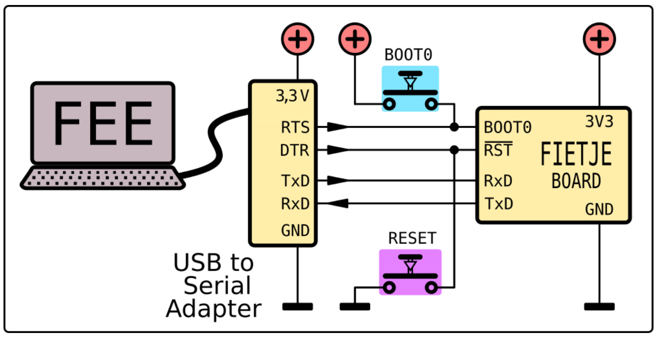


Figure 1: Schematic

The Fietje Miniboard already has the necessary pull–up and pull–down resistors at the pins BOOT0, BOOT1 and RSTN, so that these do not have to be connected for normal operation. The BOOT1 pin is connected to GND (low) and therefore does not need to be routed at all.

You can connect the Fietje Miniboard to the USB–Adpter on a breadboard with a few cables (fig. 2), or contact it with clamp programming adapters via six pins provided for this purpose (fig. 3).

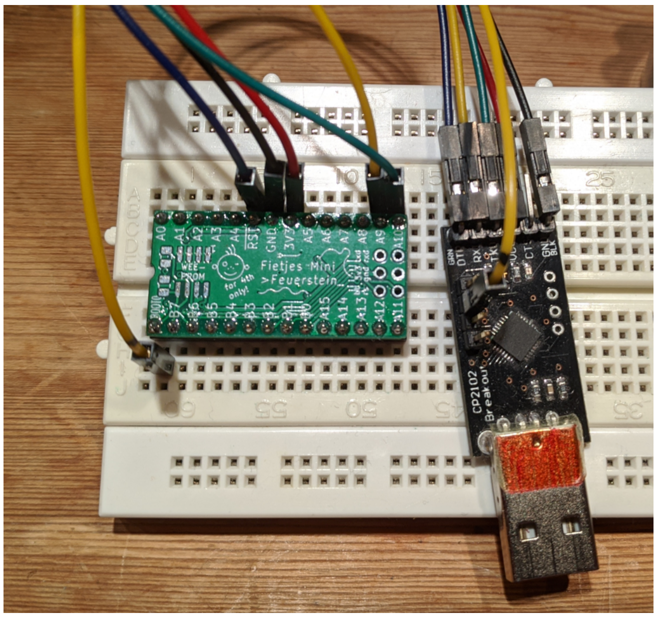


Figure 2: Variant “Breadboard”

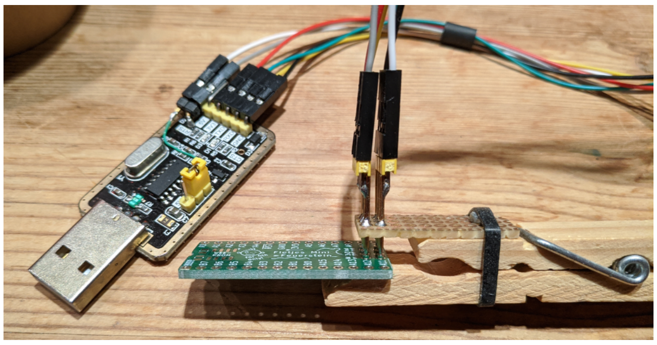


Figure 3: Variant “Clothespin”

Flashing at Last

Once the connections have been made, the target system is powered via USB adapter and can be controlled by FEE fully automatically or semi–automatically (via reset button and BOOT0 button, see fig. 1). You call the activity “firmware flash” in the menu “tools” and FEE explains on the output in the terminal window what to do: select the file with the desired Mecrisp, follow FEE’s instructions. Now it is time to lean back and watch as FEE creates a backup copy of the current Flash contents, erases the Flash completely, writes the Flash with the selected binary file and finally performs a verification for correct writing — done. When using the handshake control signals, the target system is then immediately booted into the Forth and FEE is ready for code development.

Yes, it can be that simple.

At `https://forth-ev.de/wiki/projects:fee:start` you can find the project page of FEE. There you can find the FEE, links and further information.

[Translated to English by Wolfgang Strauß]

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

```
\
\ 10Micron.f - code for controlling the 10Micron mount

0 value MNTSOC
\ value type holding the socket number of the 10Micron mount
256 buffer: MNTBUF
\ buffer to hold strings communicated with the 10Micron mount

: 10u.tell ( c-addr u --)
\ pass a command string to the mount
    10u.checksocket if drop drop exit then
    dup -rot                          ( u c-addr u)
    MNTSOC writesock                  ( u len 0 | u error SOCKET_ERROR)
    SOCKET_ERROR = if ." Failed to write to the socket with error " . CR exit then
    <> if ." Failed to write the full string to the socket" CR exit then
;

: 10u.ask ( -- c-addr u)
\ get a response from the mount
    10u.checksocket if MNTBUF 0 exit then
    0 >R 5                            ( tries R:bytes)
    begin
        1- dup 0 >=
    while
        200 ms
        MNTSOC pollsock               ( tries len | tries SOCKET_ERROR)
        dup SOCKET_ERROR = if
            drop ." Failed to poll the socket " CR
        else
            0= if
                ." 0 bytes available at the socket" CR
            else
                MNTBUF 256 MNTSOC readsock    ( tries len 0 | tries error SOCKET_ERROR)
                SOCKET_ERROR = if             ( tries ior)
                    ." Failed to read the socket with error " . CR
                else                          ( tries len)
                    R> drop >R drop 0         ( 0 R:bytes)
                then                          ( tries R:bytes)
            then
        then
    repeat
    drop MNTBUF R>
    dup 0= if ." No response from the mount" CR then
;

: MAKE-COMMAND
\ defining word for a 10Micron command
\ s" raw-command-string" MAKE-COMMAND <name>
    CREATE  ( caddr u --)
```

```
        $,                          \ compile the caddr u string to the parameter field as a counted string
    DOES>   ( -- caddr u)
        count                       \ copy the counted string at the PFA to the stack in caddr u format
        CR 10u.tell
        10u.ask 2dup type
;

s" :GR#"  MAKE-COMMAND      10u.RA? ( --)
\ get the 10Micron telescope right ascension in the raw format
s" :GD#"  MAKE-COMMAND      10u.DEC? ( --)
\ get the 10Micron telescope declination in the raw format
s" :Gstat#" MAKE-COMMAND    10u.status?
\ get the status of the mount

\
\ celestial.f - handle and convert between various celestial data formats

: $DEC ( deg min sec -- caddr u)
\ obtain a declination string in the format sDD*MM:SS from 3 integers
    <#                              \ proceeds from the rightmost character in the string
    0 # # 2drop                     \ numeric output works with double numbers
    ':' HOLD
    0 # # 2drop
    '*' HOLD
    dup >R
    abs 0 # #
    R> 0 < if '-' else '+' then HOLD
    #>
;

\
\ mount.f - develop a Forth language to control the mount

: MAKE-TARGET
\ defining word to name a set of coordinates in RA and DEC
\ (RA ) hh mm ss (Dec) deg mm ss MAKE-TARGET <target-name>
    CREATE ( hh mm ss deg mm ss --)
        , , , , , ,
    DOES> ( -- hh mm ss deg mm ss)
        dup 5 CELLS + DO
            I @
        -1 CELLS +loop
;

: GOTO ( hh mm ss deg mm ss -- caddr u)
\ slew the mount to a target and return the signal from the mount
\ <target> GOTO
    $DEC 10u.DEC 2drop
    $RA 10u.RA 2drop
    10u.slew    \ return only the final signal from the mount
;

: PARK ( --)
\ park the mount
    10u.park
;

: STOP ( --)
\ stop (halt) the mount
\ HALT is a VFX multitasking word
    10u.halt
;
```

Souped Up SBCs

Rafael Deliano

Most applications are simple. An 8–bit controller is sufficient for them. The main advantage is not, as one might think, that they have such low material costs, but their low complexity. This avoids errors, and shortens the development time.

However, some devices have higher requirements. The user then likes to flirt with a new controller that has more of everything: pins, IO, Flash, RAM, MHz. Everything ready on one chip. The first disillusionment comes when he realizes that these controllers typically come in packages with pitches smaller than 0.5 mm (e.g. QFPs or BGAs), which are difficult to solder by hand. Another difficulty is that the pins are a pain to contact with the oscilloscope.

Especially for devices that are built in single or very small quantities, development costs, not material costs, must be minimized. It is then more obvious to investigate whether the small 8–bit controller that one normally uses can be suitably extended. For IO and data memory, this is easily achievable.

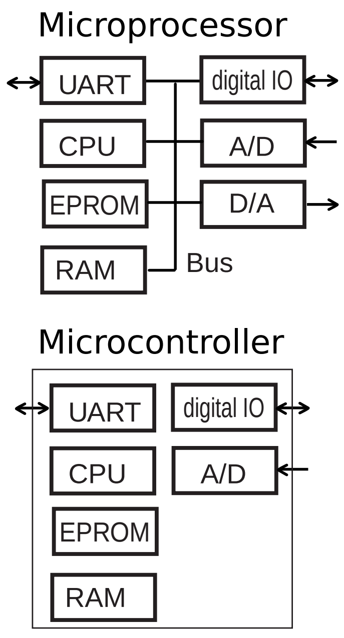


Figure 1: µP and µC

Microprocessor

The originally used 8–bit microprocessor (CPU 65C02) on my SBCs had CPU, memory and IO connected via a parallel bus (Fig. 1), which is on the board, externally so to speak, thus accessible. One had free choice in the amount and type of memory and IO.

Single–chip controllers (MCUs) have everything integrated on one chip. Cheap and small. But the mix of features is now predefined and corresponds to the IC manufacturer's ideas of what "typical applications" might need. Program memory is usually plentiful, but there is often too little RAM. Their packages, while easy to work with, have few pins and thus a limited number of IO ports. And if A/D and D/A converters are integrated, their resolution is usually not sufficient for measurement functions.

Simulated Bus

Nevertheless, a classic memory mapped IO can be built with the old ICs made for microprocessors (Fig. 2). The bus required for this can be simulated by software, at least as far as the controller is supplied with 5 V and one can spare two 8–bit ports, this is possible.

The selection of ICs for the microprocessor bus used to be abundant. They make certain applications much easier. Dual UARTs with integrated deep FIFOs, for example, are ideal for loggers via RS232 (Fig. 4 and Fig. 7). To get more port pins, the old LSIs[1] are not so attractive anymore, mostly one is happy with 74HC574 and 74HC(T)541. The typical application for the external bus is fast access to 12– to 16–bit A/D and D/A converters. These analog ICs can also convert to the ±5 V or ±15 V, which is often more favorable for analog technology.

[1] LSI — Large Scale Integration

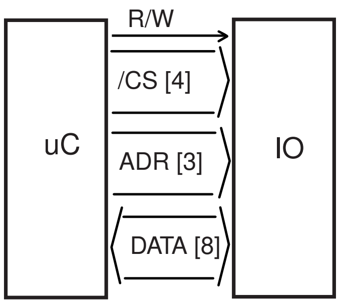


Figure 2: Controller extended with parallel bus

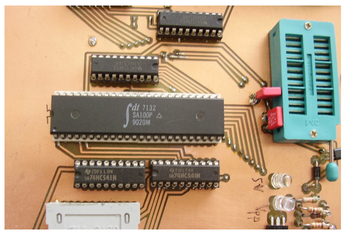


Figure 3: Dual Ported RAM

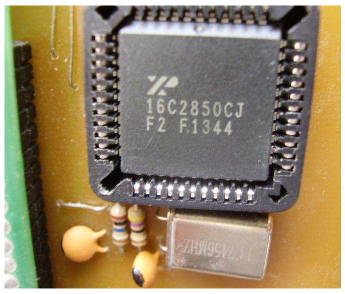


Figure 4: XR16C2850 Dual UART

Dual Ported RAM

The advantage of the FIFO is that it has no address pins and thus the pin count remains low. With this RAM, however, both busses are complete and double. The design is then with modest 2 kByte already a bulky DIL48. The classical function was the fast connection of two microprocessors via shared RAM. Today, this will only be useful for controllers in special cases. An obvious application is still the EPROM simulator for 2716 (Fig. 3). For setups with retro CPUs like 4004 and 8008 FIFOs are worth recommending even today.

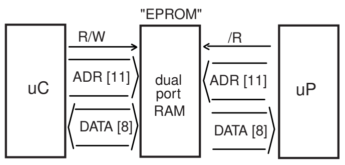


Figure 5: Application Dual Ported RAM

FIFO

With measurement data, one often has the problem that only short, but very fast sampling is required. The controller itself is too slow for this. That is a useful application for old FIFOs. This kind of RAMs are still commercially available in sizes from 2 kByte to 32 kByte with access times of about 35 nsec also in DIL.

CCDs[2] are a typical application in combination with an 8–bit Flash A/D converter. Logging fast binary bus signals from e.g. JTAG debug interfaces is also a common application. A controller acting as a logic analyzer not only samples raw bit patterns, but can also flexibly decode the log later.

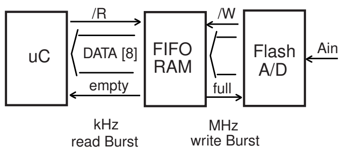


Figure 6: Application FIFO

Serial Bus

Most controllers have hardware supported serial interfaces led to the outside, like SPI or I^2C. The fact that these only need a few port pins is an advantage, reduced speed is the disadvantage. I^2C is somewhat susceptible to interference due to the bidirectional data line with pull–up resistor, but needs fewer pins than SPI. With SPI, on the other hand, the signal can also be routed via a ribbon cable at reduced clock speeds. This offers advantages in the mechanical setup, if one needs connections from a horizontal main board to the board with control elements on the front panel. This is because the latter often have to be oriented at a user-friendly angle (Fig. 9).

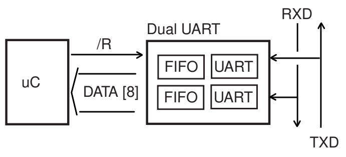


Figure 7: Application RS232 Logger

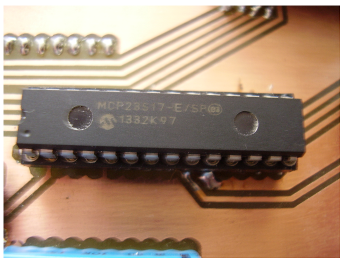


Figure 8: MCP23S17, a serial port expander

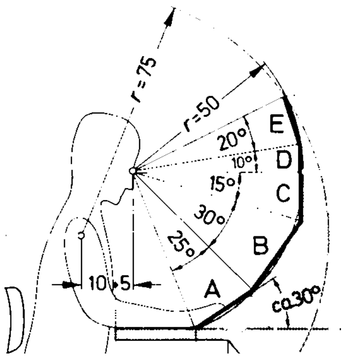


Figure 9: Ergonomics for devices

Memory

The traditional application has always been small serial EEPROMs. Even if the controller has internal EEPROM today, many applications can still be found. E.g. many arithmetic functions can be represented with good resolution by interpolated tables. Because these tables are memory–intensive, but the Flash in the controller is better used for program code, a 128 kByte SPI EEPROM 25LC1024 is a reasonable application for this.

Today, the alternative to the EEPROM is often the FRAM. For 5 V there is the FM25W256 (32 kByte) in SOIC8. The advantage over the EEPROM is that they can be written much faster. That means, if you detect a voltage drop of the supply in time, you can save the data from the RAM into the non–volatile FRAM.

In the past, people were used to having plenty of RAM available on microprocessors. For some applications this is absolutely necessary. A SPI SRAM 23LC1024 operates with 5 V and provides 128 kBytes of RAM.

The access to this serial memory in the 8–pin package is only medium fast, but in many applications the software accesses the external memory relatively seldom and there is then almost no disadvantage in speed.

Ports

Nowadays there are also SPI port extensions, like the MCP23S17 (Fig. 8), which offer bit programmable pins like in the microcontroller. However, traditionally the MSI[3] shift registers like 74HC595 or 74HC165 are used for IO.

Coprocessor

Nevertheless, there is more than just simple IO or memory that can be added to your humble "8–bitter". ICs with more complex functions have also become temptingly inexpensive today, e.g. the floating point coprocessor AM9511 or the decoder for barcode reading pens HBCR2211 or the motor controller LM629. These devices save development time for software and enable functions that would otherwise be out of reach for 8–bit CPUs.

[Translated to English by Wolfgang Strauß]

[2] CCD — Charge Coupled Device, light–sensitive sensor, e.g. for cameras
[3] MSI — Medium Scale Integration

Booting Programs from Disk

Rafael Deliano

With PCs it is common practice to load a suitable test program when it is needed. With small controllers this is not usual. But if a cable tester needs a large number of suitable short test programs for a large number of cables, it is a very neat solution to supply the programs on the connector adapters at the same time. This is easy to implement in Forth.

The first generation V1.0 of the cable tester (Fig. 1) had the GP32 controller with an 8–bit A/D converter and little RAM. And as an option on the rear a pluggable board with EEPROMs for configuration (Fig. 3). This was cumbersome and therefore never used. In addition, it turned out that configuration via data tables was not sufficient. Each cable needed its own small program after all.

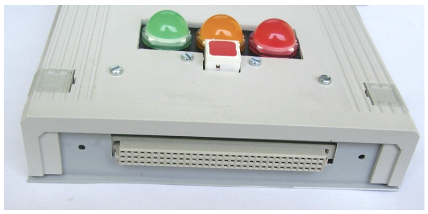


Figure 1: Cable tester with 64pin socket

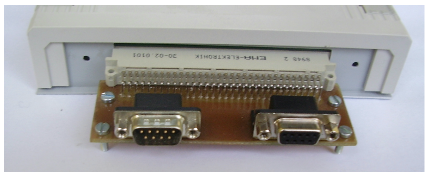


Figure 2: Cable adapter plugged in

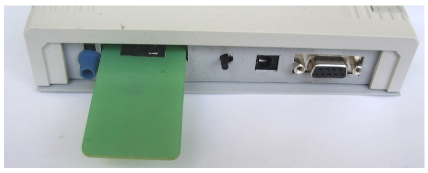


Figure 3: V1.0 with data board plugged

It was easier to place wire jumpers on the adapter boards, read them, and then use that information to start the correct test program (Fig. 2). As the number of cable types increased, the program memory filled up. FORTH programs are short (Tab. 1), but it adds up. In addition, board identification became difficult because the wire jumper codes were often randomly chosen.

Cabel name	Bytes
ROHR–MIL	1239
CABLE–560	395
MULTI–MIL	1382
SINGLE–MIL	654
506–50pol	389
SQG–neu	703
506–ODU	350
Main program	7042

Table 1: Memory map V2.0

At this point, the transition to V2.0 had already been made using the AW60 controller. The higher resolution of its 10–bit A/D converter also allowed a more accurate cable resistance metering. It also had more RAM, of its 2 kBytes, 1.7 kBytes are free. The Von Neumann CPU 68HC08 can run software there. At least the short programs that are needed for this application.

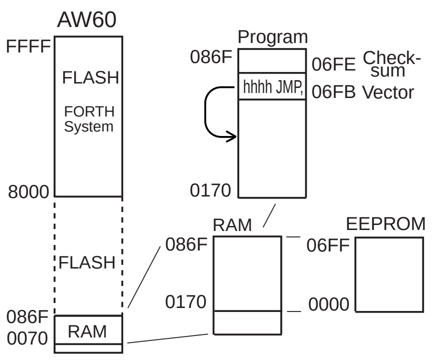


Figure 4: Memory map V3.0

The idea was to place the memory for this directly on the PCB of the cable adapter. The originally used 64pin connector[1] did not occupy the middle row of pins (Fig. 1). This allowed an upgrade to the 96pin socket, which makes an additional 32 pins available. Enough for the connection of the memory. A serial EEPROM 25LC256 is used. SPI needs more pins than I^2C, but has cleaner signals. Fast booting is therefore less critical. The pinout on the connector is matched to a layout for the standard SO8 footprint of the EEPROM (Fig. 5) and is so simple because the cable adapter PCB is almost always single layer. The circuit there has only two components (Fig. 6).

Figure 5: EEPROM

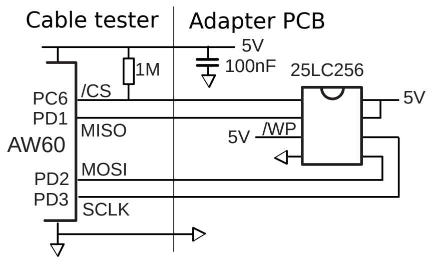


Figure 6: Schematic EEPROM

Software

This is problem–free with the simple compiler of nanoFORTH. Its variable `>C` normally points to free program memory in Flash. It is redirected to RAM at address `0170h` and compiles there. The program can be executed immediately for tests. But of course it is not reset–proof. This requires copying the 1.7 kByte block to the external EEPROM. A simple additive 16–bit checksum is also created and a program start vector is initialized.

[1] These DIN 41612 series PCB backplane signal connectors are wear resistant, rugged and have a long mating cycle life.

Run

After the reset the system checks whether an external EEPROM is accessible, copies the data block from it into RAM and calculates the checksum at the same time. If everything is correct, the program is started via the vector.

Cabel Adapter

Theoretically, the manageable number of old adapter PCBs could still be used. However, the old software structure would have to be maintained in parallel. But that's unattractive, and it is better to avoid this complexity. Supporting legacy systems rarely pays off.

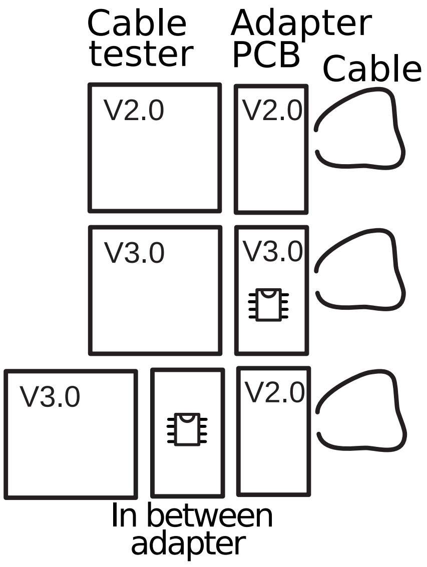


Figure 7: Configuration

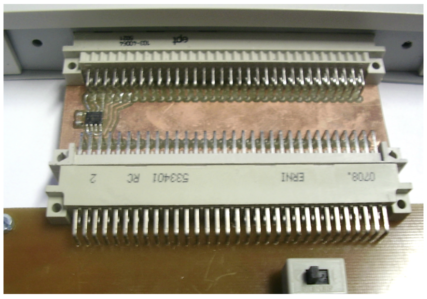


Figure 8: Adapter with serial EEPROM

Re–layout of the old boards, this time with EEPROMs, is therefore the cleanest solution and is the long–term goal. The short–term alternative, however, is an adapter that only contains the memory (Fig. 7, center of image; Fig. 8). However, one would have to build dozens of them, not very pleasant. Not elegant, but more economical, is an adapter with 8 EEPROMs (Fig. 7, bottom of image), each selected by a jumper at its `/CS` pin. So you can convert all old adapters with only a few boards. Disadvantage: The adapters have connectors as well, so that an additional contact resistance occurs, which has a slightly negative effect on the measured values.

Background

Cable production takes place in most electronics companies, but their testing is neglected. Also, because affordable devices are no longer commonly available on the market. Do–it–yourself helps. The circuit is not complex, neither is the software with its 8 kBytes.

The trick was, that I took a good, but meanwhile obsolete device as a starting point: the WK2 from WEE GmbH, a company specialized in that [1] (Fig. 9). As Newton put it, "If I have seen further, it is by standing on the shoulders of giants."

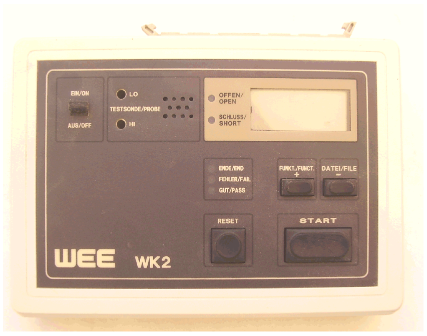


Figure 9: WK 2 (1989 – 1993)

Mine is not a commercial product, but is now used in two companies for batch testing in production. Since all PCBs of the first version are now used up, the new production had to be started. A reasonable time for the upgrade.

Finally a look into the case of the PCB version V1.0 with the GP32 SBC to show how these SBCs are integrated into devices (Fig. 10).

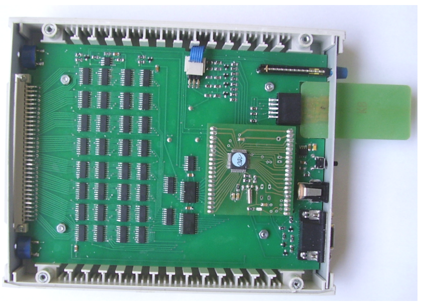


Figure 10: Inside view of the cable tester V1.0. Square board on the right in the center of the picture: SBC with nanoFORTH on GP32. Plugged in top right: data board with serial EEPROMs. On the left the array of 74HC4051 multiplexers for wire measurement.

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

[Translated to English by Wolfgang Strauß]

Sculpting Forth for SPI Flash

Michael Kalus

The other day there was an opportunity to get a closer look at these small external Flash data storage devices on the SPI bus. A friend had the problem that his copier for the SPI Flash chips was broken. He lives in the USA and has a small production of boards, which he sells quite well. What to do? A small MCU, noForth and two sockets = Flash copier. Shouldn't be that hard, right?

But the Lord God put sweat before success. Even if the individual steps were not so difficult, sometimes it went steeply uphill. But once you have climbed the SPI highlands with Forth, you can move forward comfortably, wandering here and there with pleasure until you have roamed all the corners. Interactive as Forth is, it invites you to explore the whole landscape — that's why I use it.

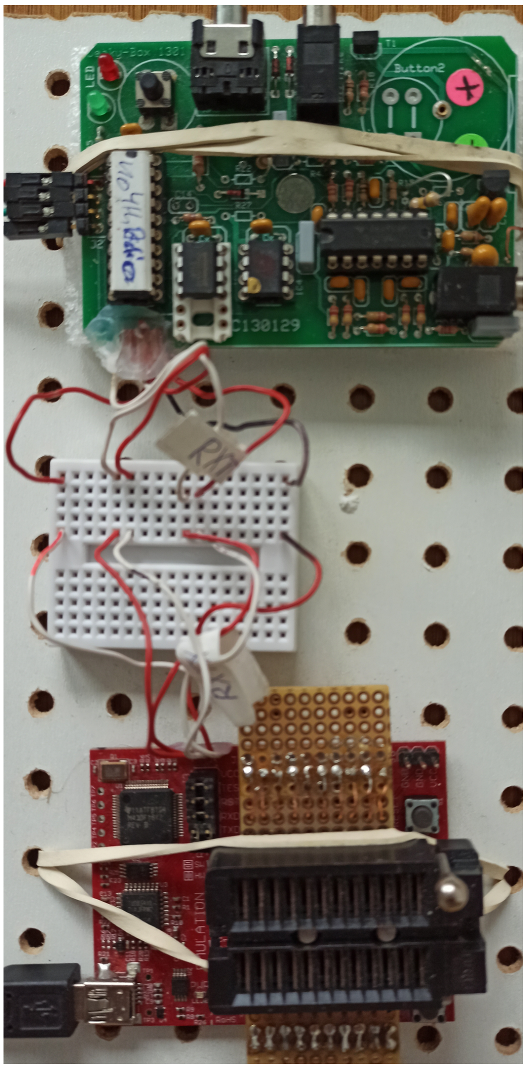


Figure 1: At the top of the picture you see the audio board. The two DIP8 are the Flashs. To the left is the MCU with noForth. Everything else is audio stuff; it is not needed for copying. Below in the picture my antique programmer launchpad, at the same time USB–serial converter.

Hardware

The setup was simple. The board of the friend already had two sockets for the SPI Flashs and the MSP430G2553 from Texas Instruments (fig. 1).

The MCU now got the noForth instead of its C program. A couple of serial Flash memories of the type W25X40CL from Winbond in the size 4 MBit (512 KByte) for trying out in addition. One of the two SPI sockets held the Flash original, the other got the empty chip for the copy.

The SPI bus turned out to be completely uncritical at such a short distance of a few centimeters. The two serial lines, data input and output, are handled by the chips on both sides of the lines on their own. The MCU is the master, the Flashes are slaves. The clock is done by the master — Serial Clock, `CLK`. All this is done by the hardware modules. Once initialized, the SPI protocol runs automatically, bit banging not needed this time.

But you have to take care of the wires to select the chips. Two I/O bits from one of the MCU ports, which can be set `high` or `low`, were sufficient for this. Each chip got its own `/CS`, and the job was done.

How did I know that? I had no idea at all. So first you browse through the papers that are out there.

Dokuments

The Winbond data sheet / user manual left no questions unanswered. Neither did the manual for the MCU. And noForth is also well documented. That's the way to work. noForth has a small SPI package for this MCU. That was a good start to get familiar with the SPI hardware driver of the MCU. So a first test was successful: reading the ID of a SPI Flash.

I soon learned that these SPI Flashes can handle a whole set of commands that make your life easy regarding writing and reading the Flash.

Toolbox

This was a Windows 10 project. That's what the friend wanted. The programmer's quill has therefore been Notepad++, which is a fine editor. The MCU itself is connected to the USB–serial converter and that links to TeraTerm, a halfway usable terminal simulator when working with Forth. I use it, because there is not much better — or there wasn't so far. And I hope every time that the Forth community will develop something more suitable.

The TeraTerm is forth–dumb, yes, unfortunately. But what to do? So it has line–by–line transmission, at the end of the line there is blind waiting for the noForth, so it can process the line and on it goes. Since the individual Forth test snippets are always quite small, this works. And the copy&paste feature of TeraTerm is ok. If you have a larger piece of Forth source code ready, the file download also works quite well.

noForth can be loaded into the MCU with the FET–Pro–430–LITE programmer from Elprotronic — a very reliable tool. The UniFlash from Texas Instruments also does the job. But since I already know the other programmer and have it around, well, I stick with it.

Used was noForth–mcv. The m means "MSP430"[1], c stands for "compact" and v for "vocabularies". You can get the noForth image as an Intel hex file from the developers website [1]. I had a MSP–Launchpad to use as a programmer available, so no problem [2].

Software and Testing

You can find them in the listing. noForth, small as it is, knows `marker`, so that you can quickly reset it to its protected core again and again. The software experiments do not always run correctly at the first attempt, and the program garbage from the trial phase should not remain in the program memory of the MCU. noForth has a small library, called `tools`, which you can load and use for the development. There is `.s` in it, `words` and such. Later you can get rid of it. There was enough free space in my project, so I kept it.

How to initialize the peripheral module for SPI in the MSP430G2553 is kindly shown in a small noForth library. I took this library, matched the ports to the given board and was ready to send commands to the SPI memory.

Winbond writes:

> "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."

Simple, isn't it? You think so, but first you have to design tests to verify all functions. This procedure is described in detail in the listing of the tests. The principle is always the same: first think about what could be considered as proof of correct functioning, write a Forth word for it and then modify it until it not only runs well, but also looks good. Then move on to the next.

That's the beauty of interactive Forth programming: Forth can be modeled, like clay in your hands. This makes programming an aesthetic modeling activity. Each Forth word that is created is a lump of the sculptor's building clay. It is pushed into shape until it sits correctly in the overall work.

So let's get to work. Checking the /CS lines is done with the multimeter, is the H or L level there as it should be? This is a test step, so there is a pair of Forth words for it: `csup` and `csdown`. And because either one or the other Flash is to be selected, the mask `CS` determines whether `IC3` or `IC4` is meant.

This allows phrases like:

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

And then afterwards:

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

With this it can already be seen whether the SPI Flashes respond and the whole SPI works.

An internal test of the SPI hardware is kindly already provided by the MCU manufacturer. There is the possibility to put the module into an internal loop: In this mode the byte you just wrote to the transmit register appears directly in the receive register. This way you can test your send and receive Forth words. After all, what goes out on the transmit line must arrive on the receive line. The word

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

can be tested this way.

After it is clear that SPI bus and /CS lines work correctly, it is a pleasure to see that the Flash chips accept commands and answer them well. They willingly deliver their identification numbers "Device ID & Manufacturer" on the data stack. `.s` shows this immediately — gee, Forth is nice.

Of course, the maneuver was a bit naive. But luck is a part of the game: noForth is slow enough to give the SPI Flash time to settle down after the /CS.[2] So timing was not a problem with such a simple request to the Flash like, "What is your ID?" Only after the INIT it needs an extra 5 ms once at the very beginning until all components are operational.

From then on it is a walk in the SPI high (language) plain to translate the commands for the SPI Flash into words.

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

[1] MSP430G2553 MCU, Texas Instruments

[2] /CS Active Setup Time relative to CLK is 5 ns according to the datasheet of the SPI Flash

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

Sent are one after the other the instruction, an address and then data, either back or forth.

The special thing about SPI: While the master side is writing on the send data line, the module is already receiving bytes from the Flash on the other data line. This was a bit confusing for me in the beginning. But you soon learn when the received bytes are just dummies that you can drop, and from when valid stuff comes.

The Flash instructions are all only one byte long, the addresses 3 bytes — 24 bits with the Flash size that was used here — and the data comes byte by byte once again.

For the interactive way of working, the send routine `>spi` is already good enough to send one instruction byte to the Flash. The following address is sent with `>adr24` and the data again byte by byte. Start and stop of the transmission are also Forth words: `csdown` for the falling edge to start a sequence and `csup` to end it. The timing for this is already included in `spi`: it waits until the *busy flag* in the send and receive registers indicates that the chip is ready.

Well, that's about all I would have to say about this. To show how I did it and that it was fun.

You can find out the rest yourself by looking into the source code.

User Interface(s)

The `blockcopy` is the real workhorse, you've probably seen it there. The management of the sections up to the `copyapp`, which is based on it, is nothing more than a mechanism to make the user happy. After all, that was the real goal: to have a standalone board that copies Flashes. Plug in the Flash, turn it on to copy it, a little "Blinken Light" for human orientation, turn it off again, take out the copied Flash, and do it all over again. No more buttons, just the one LED — the minimal equipment of the board, "as is".

But since the board can also be operated via a terminal using the serial interface, a second user level was added: The command line accepts `copy+` and `copy-`. If you set the `report` value, the heap of wire talks to you, otherwise it copies silently.

But this already has nothing to do with the SPI, this is an extra that other users probably want to have differently. How to separate the *business code* cleanly from the user interface and one's language[3], i.e. localized, I still have to find out. In Gforth that's already possible[4], noForth doesn't support that yet, as far as I know. And if such a very small Forth can do that at all, also needs to be researched.

By the way, the copy takes 15 s. For comparison: The Flash internal complete chiperase takes 12 s. And copy already does the verification and "talks" to the user, which slows it down. Not bad, right?

Ideas to prevent confusion of source and target are also welcome. For now, this is solved primitively: The source is in the socket IC4 and only to IC3 can be written.

Have fun with Forth.

[Translated to English by Wolfgang Strauß]

Links

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

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

Listings

Of course, the `tools` of noForth have to be loaded first. To avoid bloating the magazine, they are not listed here.

spi

```
\ spi for noForth m(cv) -- september 2022
\ MSP430G2553

tools\
hex
: spi-  01 69 *bis ; \ UCB0CTL1  stop bit set
: spi+  01 69 *bic ; \ UCB0CTL1  released --> run
: MASTER-SETUP
  spi-
    \ ports general i/o
    A0 22 *bis  \ P1DIR     P1.5&7 output
    40 22 *bic  \ P1DIR     P1.6 input
    \ special function pins
        E0 26 *bis  \ P1SEL     P1.5+6+7 is SPI
    E0 41 *bis  \ P1SEL2
    69 68 *bis  \ UCB0CTL0  Clk=high, MSB first,
                \ 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 ;

```

[3] Uncle Bob: "Clean Code"

[4] Bernd Paysan; Internationalisation mit Gforth; issue 4d2022–04, p. 10 ff

```
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.
*)

here 10 allot  constant buffer0  \ 16 bytes buffer
here 10 allot  constant buffer1  \ 16 bytes buffer

value buc  ( -- buc ) \ buffer character counter

\ Reading and writing of the buffers
: buc0   ( -- )    0 to buc ;
: buc+   ( -- )    buc 1+ f and to buc ;
: >buf0  ( c -- )  buffer0 buc + c!  buc+ ;
: buf0>  ( -- c )  buffer0 buc + c@  buc+ ;
: >buf1  ( c -- )  buffer1 buc + c!  buc+ ;

: fapsep0 ( -- ) \ for convenience
```

```
  0 to fap 0 to sep ;

\ Set important Flash addresses
: setblock    ( i -- )
  f and 10    * fap ff00 and +  to fap ;
: setpage     ( i -- )
  f and 100   * fap f000 and +  to fap ;
: setsector   ( i -- )
  f and 1000 *                  to fap ;
: setsegment ( i -- )  f and    to sep ;


\ The "Report" feature was added later. It also
\ belongs to the "keep the user happy" category
\ and does not add anything to the copy function.
value report

\ The feature "Block=FF" was also added later.
\ This uses a property of the master Flash: Its
\ segments are not fully written. As soon as a
\ block is "FF", nothing more comes in the
\ segment, the copy function can go to the next
\ segment.
value FFsum   ( -- n )  \ sum of bytes in the block
: FFcnt ( x -- x )  dup +to FFsum ;
: ff?   ( -- f )    FFsum ff0 = ;

: rbuf      ( -- ) \ read current Flash block
  0 to FFsum
  fap sep read  buc0
  10 0 do spi> FFcnt >buf0 loop  csup ;
: wbuf     ( -- ) \ write current Flash
  ff? if exit then
  fap sep page  buc0
  10 0 do buf0> >spi loop  csup ;
: bbuf     ( -- )  \ read back written Flash
  ff? if exit then
  fap sep read  buc0
  10 0 do spi> >buf1 loop  csup ;

\ LED pulses indicate an error
: sos ( -- )
  3 0 do bliz loop 200 ms
  3 0 do bliz 100 ms loop 200 ms
  3 0 do bliz loop 500 ms ;
: buf<> ( -- f ) \ f = true if different
  ff? if exit then
  buffer0 10 buffer1 10 s<>
  if begin sos again then ;
\ Yes, we terminate by "power off"


\ Now, let us copy for real
: blockcopy   ( i -- ) \ copy current block
  setblock
  ic4 rbuf
  ic3 wbuf bbuf buf<> ; \ buf<> is verification

: pagecopy    ( i -- ) \ 0xFF  &256 bytes
  setpage
    report if ." block " then
  10 0 do
  ff? if unloop exit then
    report if i . then
  i blockcopy  loop ;

: sectorcopy ( i -- ) \ 0x1000  &4096 bytes
  setsector
  10 0 do
  ff? if unloop exit then
     report if cr ." page " i . then
  i pagecopy  loop ;

: segmentcopy ( i -- ) \ 0x10000  &65536 bytes
  setsegment
  10 0 do
  ff? if unloop bliz exit then
     report if cr ." sector " i . then
  i sectorcopy loop bliz ;

: flashcopy   ( -- )  \ 0x80000 = &524288 bytes
  fapsep0
  8 0 do
     report if cr cr ." segment " i . then
  0 to FFsum  i segmentcopy loop ;


shield flash\

(* And finally the main word: COPYAPP
This will be the "turnkey app", it starts after
reset. Actually FLASHCOPY is the copy function.
But then you can't see that the board is working
correctly. So you have to use "blinken lights"
and because the board has only one LED, this one
has to do the job. Oh yes, the board can also
make a sound.
*)

\
v: forth definitions
hex
: copy ( -- ) \ blink while copying entire flash
  \ stop watchdog: noforth itself does that
  \ 5A80 120 !
    report if
    cr ." start copying and wait for the system
    to stabilize ... " then
  500 ms  init  0 to FFsum
    bliz bliz bliz   \ keep the user happy
    report if ." done" then
    report if cr ." chip-erase ... " then
  chip3erase
    bliz bliz
    report if ." done"
    cr ." copy 8 flash segments, "
       ." skip empty (0xFF) blocks." then
  flashcopy
    report if cr cr ." finis " then
    beep ;

: copy+ ( -- ) \ copy with report
  true  to report  copy ;

: copy- ( -- ) \ copy without reporting
  false to report  copy ;

: copyapp ( -- )
  copy-
  \ ready -
  \ now we bliz back and forth (pun intended)
  holdL 200 ms begin bliz again ;

 shield copy\
\ ' copyapp to app   \ undo: ' noop to app

: .. ( adr24 -- ) \ Test: see some 32 bytes of IC
  read
  cr 10 0 do spi> . loop
  cr 10 0 do spi> . loop
  csup ;
\ passed: 20221018mk

fresh
freeze          \ application done  20221020mk
init chere u.   \ check memory: ok
( finis)

(* Notes:
Only in noForth with vocabularies:
EXTRA is a vocabulary with non-standard useful
words.
INSIDE is a vocabulary with internal words.
: FRESH ( -- )
  only extra also forth also definitions ;

fresh order ↩ ( FORTH FORTH EXTRA ONLY : FORTH )

When noForth starts, FRESH is executed.
.VOC ( wid -- ) \ show the vocabulary name.
 'wid' is a number in the range 0..127
*)
```

Glossar

```
\ handle flash memory  for BC130129    2022-10-17
\ Glossar of usable forth words
\ Type WORDS to get all forth words.
\ Type EXTRA WORDS to get sub-words too. See source code for use of extra words.

cs  ( -- mask ) \  value of IC mask
Example:
cs . 80  OK.0


IC3 ( -- )  \ set cs mask to P2.7 = IC3
IC4 ( -- )  \ set cs mask to P2.6 = IC4

init   ( -- )       \ prepare chips and control lines

beep   ( -- )       \ short beep of 500Hz

ID     ( -- M ID )  \ read device ID & Manufacturer
Example:
init ic4 id .s ( EF 12 ) OK.2

status ( -- u )     \ read status register.
Example:
init ic3 status .s ( 0 ) OK.1


chip3erase ( -- )   \ set all memory of flash in IC3 to erased state $FF.
copy       ( -- )   \ copy entire flash from IC4 to IC3.
copy+      ( -- )   \ copy with report. Same as: true to report copy
copy-      ( -- )   \ copy without reporting. Same as: false to report copy


copyapp    ( -- )   \ use this as turnkey app
Example:
' copyapp to app   \ undo: ' noop to app


..         ( adr24 -- ) \ Testing: look at 32 bytes of IC
Example:
hex  init IC3 dn 12000 ..  \  look at content of flash in IC3 at address 0x12000.
\ adr24 is a double number:
@)dn 12000  OK.2
@).s ( 2000 1 ) OK.2

( finis)
```

Internationalization With Gforth and MINΩΣ2 — Part 2

Bernd Paysan

Beyond the translation issues discussed in Part 1, there are a number of other topics that are part of successful internationalization and localization, most of which require solutions outside of standards. The article builds on private communication with Claus Vogt, *who shared his experiences with the author in a longish and unstructured e-mail, and on a lecture on MINΩΣ2, in which various typesetting aspects were discussed.*

Database or CSV for Translators?

Translators are, as already noted in the first part, typically rather not programmers. The first part therefore contained the following text:

> „Using a table may be easier for maintaining programs — especially because the CSV table itself can then be maintained with a spreadsheet program; something many people are proficient at.“

Claus Vogt fully agreed with this view — for a long time he too has been using CSV files into which his translators enter the translations. In addition to the program text, it is also possible to insert other columns there that make life easier for the translator and may also be useful for automated insertion of the texts in the right place. For example, Claus has a „Context“ column that contains the identifiers of the UI elements to which the texts belong. The translator learns from this, for example, on which kind of control element a text is placed, and can choose a short, concise translation for a button, and a longer, more precise one for a text field. The program can resolve possible ambiguities and reliably insert the respective translation into the appropriate UI element. Comments that are not to be translated can also go into a separate column; comments as part of the program text end up quite certainly in the translation as well.

Based on these considerations, Gforth has now got a CSV importer, which can then also be used for loading translations. This importer is found in the current developer version in the file `csv.fs` and has a single word that is of interest to the user:

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

which reads the file with the name in the string *`addr u`*, and passes each field to *`xt ( addr u col line -- )`*, along with the column (starting at 0) and line (starting at 1) of the field.

Based on this, a new word is then defined in `i18n.fs`:

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

then loads the specified CSV file to populate the translation database, with the first line containing the respective locales, and the rest containing the corresponding texts.

Unicode, Fonts and Writing Systems

The founders of the Unicode Consortium assumed that the approach for Western languages of simply assigning a code point in a character set for each character could simply be transferred to the character sets in the rest of the world. This would create a single, much larger font, which would then be addressed with a maximum of 16 bits instead of just 8, and that would be it.

> “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]

The source, by the way, includes a table of writing systems by Gross National Product, and all the Indian writing systems are classified as “commercially irrelevant” because India had a little over 1% of the world’s GNP at the time …

The reality, of course, is more complex. The alphabets of the modern world are virtually all descended from ancient Egyptian hieroglyphics; the syllabic scripts of East Asia, on the other hand, from the oracle–bone script of ancient China.

I have found a nice family tree for the alphabets (fig. 1), which besides the examples also gives the different directions of writing systems (mostly left to right, sometimes right to left, more rarely from top to bottom).

This family tree is missing an important piece of information: Most such scripts are cursive scripts (not only those

[1] A ligature or compound letter in typography refers to the merging of two or more letters of a typesetting script to form a glyph.

[2] Diacritical marks are dots, dashes, ticks, arcs, or circles attached to letters that indicate a pronunciation or stress that differs from the unmarked letter.

in India), which use a lot of ligatures[1] and diacritical[2] characters, and therefore cannot be used simply as a traditional computer character set.

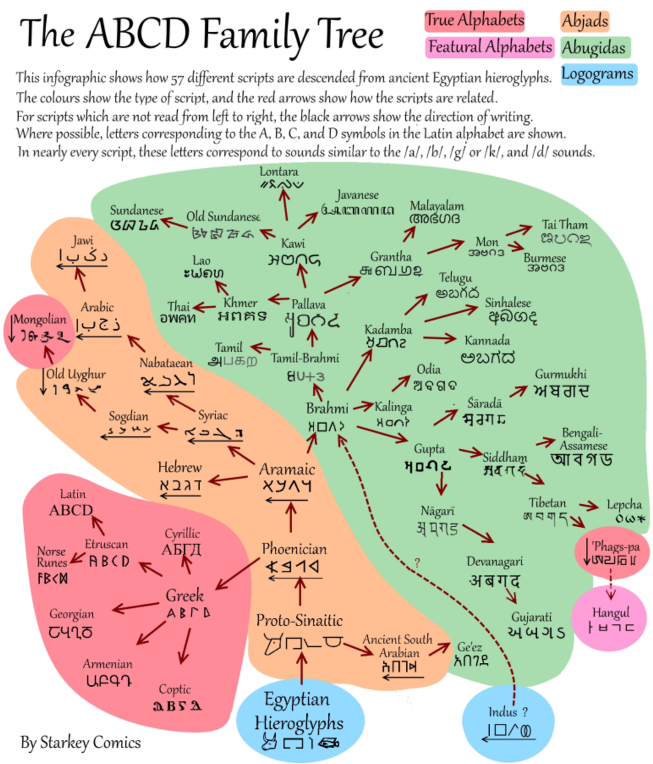


Figure 1: The ABCD Family Tree

Here are the examples for India rendered with MINΩΣ2:

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

Figure 2: Indian scripts, English name on the left, original on the right

Tibetan is especially unusual because the ligatures are also written vertically, which can result in remarkably large descenders.

For the typesetting of these complex writing systems, there is metadata in the OpenType fonts, which can be used relatively easily using the Harfbuzz library. To determine the direction of writing, there is the Unicode Bidi Algorithm, which uses a relatively elaborate heuristic and auxiliary markers, and which MINΩΣ2 also implements. Since this algorithm changes somewhat with each Unicode version, there is no guarantee that text once correctly rendered will still be correct in the next Unicode version; unless one hard–codes the respective writing directions using appropriate marker characters as well.

What did not happen at all was the unification of the respective fonts of a language into a single universal font. The code points of the respective fonts are now Unicode, but only for one language. So, to render a Unicode text, you first have to find out the writing direction, then select the language–specific font, and then set the ligatures and diacritical markers with the help of Harfbuzz.

For Far Eastern languages, however, this is still not sufficient. Japanese and Chinese both use the characters called Hanzi in China and Kanji in Japan. However, the characters have different variants. For example, the People's Republic has used simplified characters since the 1950s, while Taiwan and Hong Kong write traditional Chinese. Japanese has its own simplified characters (also introduced in the 1950s as Shinjitai), though not all simplifications have been done exactly the same way.

Unicode has separate code points for simplified and traditional Chinese respectively. These are the same language, and the variants are local features and in the course of the simplifications in Japan and China also not replaced at once, but in several steps (the last one in Japan is from 2010); here it would make more sense, the respective user installs a font according to his taste, and the characters are then just rendered either simplified or traditional. For special purposes (such as dictionaries that want to represent more than one variant), a meta–character could be introduced that switches definitively to traditional or simplified. This would make it possible for Mainland Chinese and Taiwanese to communicate in writing without stumbling over the unfamiliar representation of the characters.

In Japanese, which is a clearly different language (even if a significant part of the vocabulary is made up of Chinese foreign words, with a sound shift typical of some southeast Chinese dialects), and in which typesetting has also adopted a different variant of traditional calligraphy in print, one has simply chosen the path of mapping all characters to the most similar Chinese characters (mostly simplified); the variants are thus not encoded differently. You have to know that it is a Japanese text, and then choose a Japanese font accordingly, then it fits. A suitable heuristic would be to search for Katakana or Hieragana, which does not occur in purely Chinese texts. However, especially short Japanese texts, such as those found in UIs, but also chat messages, are often completely in Kanji, so they contain no clue as to what language this should be.

Font Size

The font size is still a sub–topic here. Traditionally, type size is measured by the height of the metal blocks, known as "sorts", used in lead type, and digital character sets are also sized according to this tradition. This is a largely irrelevant measurement for the user; what is important is that the fonts used are roughly equally legible at a typical viewing distance. To this end, MINΩΣ2 sets Chinese fonts, emojis, and hieroglyphs larger than alphabets (the latter considerably larger). Then it fits.

What Would Have Been the Right Way to Do It?

Alphabets, no matter which of the more than 50, are just alphabets, i.e., the characters look different depending on the region, but have basically the same meaning. The code point should therefore better encode which character is meant, not which writing system and language (whereby writing systems can of course contain single characters, which are circumscribed in other writing systems with several characters; the characters are however phonetic units in any case). Actually, this would already apply to our upper and lower case letters, which are also only different spellings of the same phonemes; but we use them together in a text.

The writing system used in each case is orthogonal information, which is important for selecting the correct character set, the meta information for ligatures & diacritic characters and the writing direction, whereby the writing direction can also be different for numbers (and then a heuristic is followed during input, which is then stored with the text). The information about the language used is important e.g. for hyphenation algorithms, even if the font itself looks exactly the same (i.e. relevant especially for Central and Western European texts, where a uniform font has become established after the abolition of Gothic type).

A corresponding encoding would therefore have states (or even a state stack), and the characters must be interpreted context–dependently in any case. But they must already because of the selection of the writing direction, and thus this is no loss: The correct interpretation of a text results from the whole string, not from individual characters. If you separate out a substring, you have to find the appropriate meta characters that describe it correctly. Character set variants like bold or italic (or in other fonts the choice of comparable calligraphy styles) would be included in that, and not part of a markup on another level; also because otherwise these markups interact with the Bidi Algorithm in surprising ways.

My suggestion would be to use the codes $FC (+ bytes in the range $80..$BF) to $FF, which have not been used so far in UTF–8, to encode appropriate meta–information regarding font choice and writing direction, where $FF is the pop, and all other state changes throw the previous state onto a stack. Missing pops at the end of the text are not a problem, there will be tidying up in any case; parsing invalid UTF–8 code should not throw off too much software either, since the self–synchronizing property of UTF–8 is preserved. The $F8..$FB range present in the original Unicode draft would still be usable for five–byte sequences, should the need for such large code points ever arise. If one selects a language, the following UTF–8 letter code points and code points ≥ $7F then only encode an offset within the corresponding code blocks; control characters, western "Arabic" numerals, and special characters are preserved because they are used in many writing systems, are also present in the fonts, sometimes deliberately in the variants used, and then do not have to be switched unnecessarily for this.

Ultimately, this would no longer be "Unicode", i.e. no unification, but the opposite: a "Localicode", in which in particular it is unified how to get to the corresponding locale and back.

[Translated to English by Wolfgang Strauß]

References

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

Regional Forth Groups

Please ask the organisers if the meetings will take place. This may vary depending on the pandemic situation.

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/`

Services of the Forth–Gesellschaft

Nextcloud	`https://cloud.forth-ev.de`
GitHub	`https://github.com/forth-ev`
Twitch	`https://www.twitch.tv/4ther`
μP–Controller–Pool	**Carsten Strotmann** `microcontrollerverleih@forth-ev.de` `mcv@forth-ev.de`

Special Fields

Forth hardware in VHDL microcore (uCore)	**Klaus Schleisiek** Tel.: **(0 58 46) – 98 04 00 8**$_p$ `kschleisiek@freenet.de`
AI, Object Oriented Forth, Safety–critical systems	**Ulrich Hoffmann** Tel.: **(0 41 03) – 80 48 41** `uho@forth-ev.de`
Forth distribution volksFORTH ultraFORTH RTX / FG / Super8 KK–FORTH	Ingenieurbüro **Klaus Kohl–Schöpe** Tel.: **(0 82 66)–36 09 862**$_p$

Events

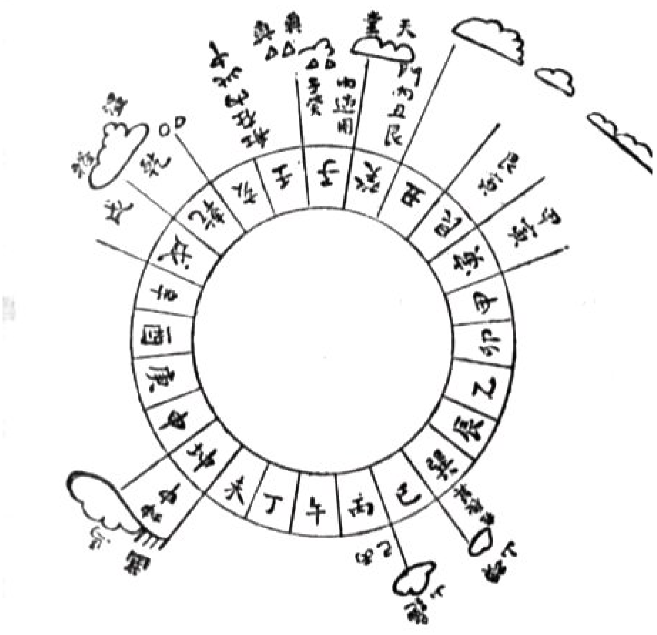
Thursdays from 20:00
Forth–Chat net2o `forth@bernd` using the key `keysearch kQusJ`, complete key:
`kQusJzA;7*?t=uy@X}1GWr!+0qqp_Cn176t4(dQ*`

Every 1st Monday of the month from 20:30
Forth Evening
Video meeting (not only) for beginners.
Info and participation link: Email to `wost@ewost.de`

Every 2nd Saturday of the month
ZOOM meeting of the Forth2020 Facebook group
Participation info: `www.forth2020.org`

Annual Forth Conference (online) May 05–07, 2023
`https://tagung.forth-ev.de`

You can find details on the dates at `http://forth-ev.de`



Would you like to start a local Forth group in your area, or just initiate regular meetings? Or can you imagine providing assistance to Forthers seeking advice on Forth (or other topics)? Would you like to make contacts that go beyond the VD and the annual membership meeting? Just write to the VD — or call — or send us an email!

Notes on the abbreviations after the telephone numbers:
Q = Answering machine
p = Private, outside typical working hours
g = Relating to business
The office addresses of the Forth–Gesellschaft e.V. and of the VD can be found in the imprint of the magazine.

Attention: Date has been Changed! Forth Conference 2023 from 05 to 07 May 2023 (Online)

Organization: Board of the Forth Gesellschaft

In 2023, the Forth Gesellschaft (Forth fellowship Germany) will again hold its conference and membership meeting online. Since this format of video conferencing has been well received in recent years, we will continue this with the already proven BBB system. Traveling is omitted — visiting monasteries and landscapes, having a delicious meal together and talking shop at the bar until late at night is also omitted, unfortunately. But the meeting is free of charge, so it costs nothing extra, except the effort to register. Non–members are also most welcome.

Registration

At `https://tagung.forth-ev.de` you will find further information, the current program and the possibility to register for the conference electronically.

Program

Friday 05 May 2023

- Evening: Informal meeting online without special program;
  starting at 19:00

Saturday, 06 May 2023

- Morning: Lectures and workshops
- Afternoon: Lectures and workshops
- Evening: Dinner together (online edition)

Sunday, 07 May 2023

- Membership meeting of the Forth Gesellschaft e.V.;
  from 10:00 to 13:00
- Space for some more workshops;
  from 15:00 to 18:00

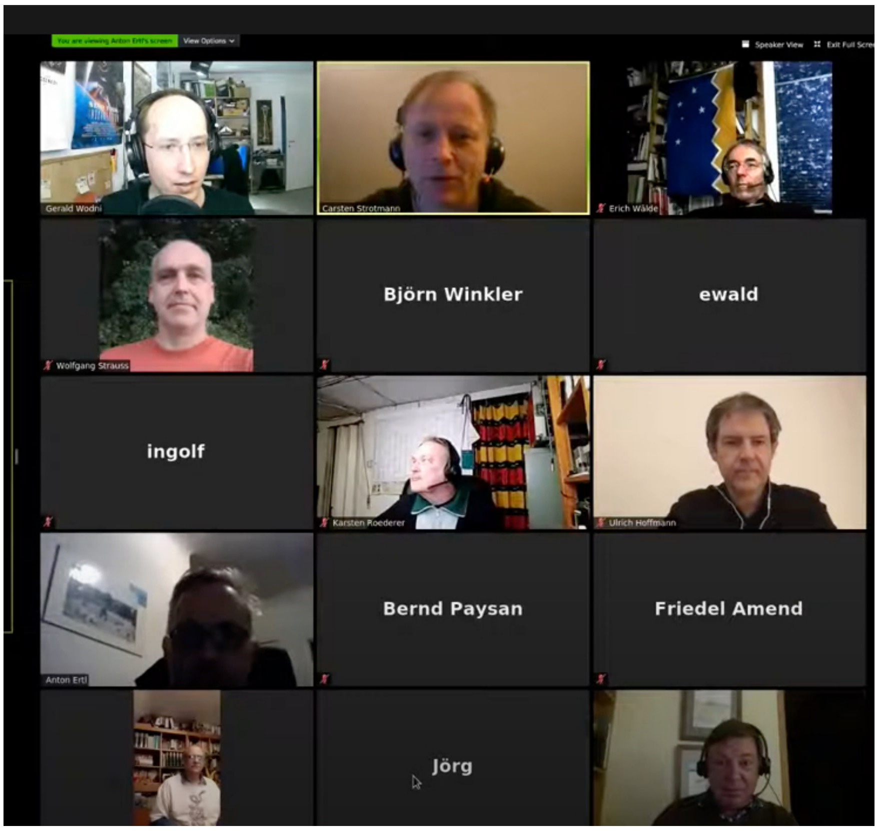


Figure 1: The other day it looked like this photo. Make sure the lighting is good and the background is nice for you.

[Translated to English by Wolfgang Strauß]