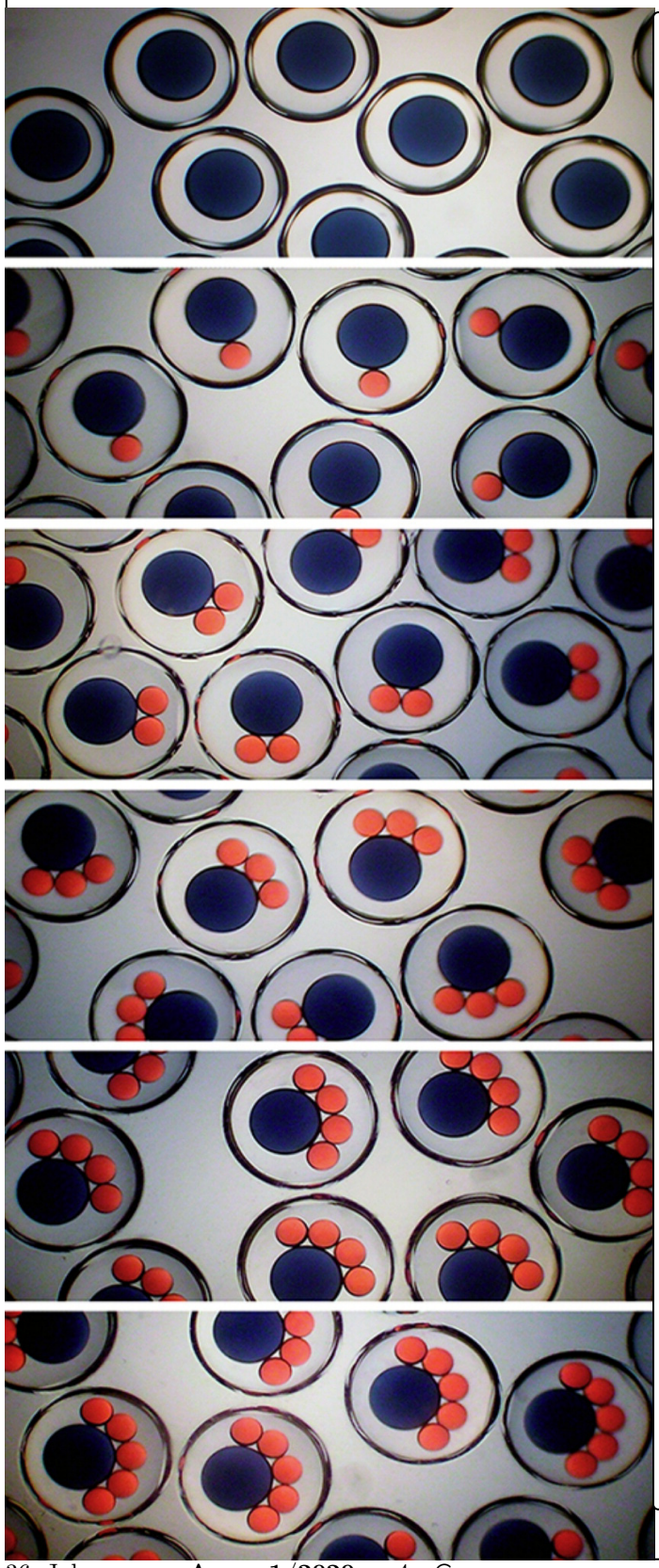
Vierte Dimension

1/2020

Das Forth–Magazin

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



In dieser Ausgabe:

Dienstleistungen und Produkte

Titelbild WeiWang et al, Microfluidic encapsulation.
Current Opinion in Pharmacology Volume 18, October 2014, Pages 35-41

Impressum

Name der Zeitschrift

Vierte Dimension

Herausgeberin

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

Redaktion & Layout

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

Anzeigenverwaltung

Büro der Herausgeberin

Redaktionsschluss

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

Erscheinungsweise

1 Ausgabe / Quartal

Einzelpreis

4,00 € + Porto u. Verpackung

Manuskripte und Rechte

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

Liebe Leser,

kurz vor unserer Jahrestagung ist wieder ein Heft fertig geworden.

28 Seiten stark, immerhin. Liebe Leser, liebe Forthfreunde in aller Welt, nicht nachlassen, tüchtig Forthiges sammeln und herschicken!

Und nun zum Heft. Zustande gekommen ist es wieder durch beharrliches Suchen in allen Ecken der Welt nach Leuten, die sich mit Forth befassen und dann auch was davon hergeben mögen. Als Korrespondenz, als Artikel oder als Kolumne. Ihr braucht keine besonderen Formate zu beachten für euren Text, da machen wir schon was draus. Wir, das sind derzeit Wolfgang Strauß, Bernd Paysan, Ewald Rieger und ich (Michael Kalus), aber das wusstet ihr ja schon. Wolfgang ist der Lektor, Bernd hält die Technik in Gang und unsere Laune hoch und macht den Feinschliff im Satz. Und ich bin der Sammler und Layouter. Ewald schließlich sorgt für den sauberen Druck und Versand der Hefte in alle Welt. Ja, tatsächlich gehen auch Hefte in Papierform über Europas Grenzen hinaus nach Amerika und Asien. Afrika und Australien derzeit (immer noch?) nicht. Nach China geht wohl grad gar nichts raus — Coronavirus Shutdown.

Was haben wir diesmal im Heft? Klaus Schleisiek stellt vor, wie weit der *µCore* inzwischen gediehen ist. Der Name ist Programm. Allerdings finden das auch einige andere Leute aus der Branche, und im Internet sind einige Produkte unter dem Namen „Core“ mit was davor wie „u“ oder so, unterwegs. Ihr wisst aber nun, um was es geht, wenn vom *µCore* die Rede ist: vom Forthprozessor, der schon lange kein bloßes Konzept mehr ist, sondern sich im Praxiseinsatz befindet.

Dann lässt Manfred Mahlow uns wieder Blicke werfen in seinen Werkzeugkasten. *Namespaces* und dann die *Voc–, Item– und Sticky–Words (VIS)* gleich hinterher. Konzepte, die auf der Ebene der sehr kleinen Forthsysteme wie dem *eforth* durchaus Platz finden. Und ja, er würde sich freuen, euch dabei behilflich zu sein, es einzusetzen.

Ganz überraschend hat F. L. van der Markt eine längere Passage Forth–Quellcode beigesteuert. Als Freund der Sensorik hat er sich diesmal dem *BME280* gewidmet. Da kommen gleich drei Messwerte heraus: relative Luftfeuchte, barometrischer Druck und die Umgebungstemperatur — spannend. Ein Novum dabei ist, dass er den erklärenden Text zum Code in die Kommentare der Source verlagert hat. Schaut mal, wie es euch gefällt.

Ulrich Hoffmann und ich selbst haben das Forth–Wort *synonym* untersucht. Anlass war ein Weihnachtsgruß von Willem Ouwerkerk — aber lest selbst.

Martin Bitter schließlich war so freundlich, das, was bei unserem letzten Forth–Treffen im Unperfekthaus in Essen klemmte, aufzuzeigen und gleich auch zu reparieren.

Wir sehen uns dann auf der kommenden Forthtagung — wann und wo die auch immer sein wird! (Siehe Rückseite)

Bis dahin. Und euch allen eine gute Zeit. Euer Michael

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web–Seite des Vereins herunterladen.
http://fossil.forth-ev.de/vd-2020-01

Errata

Unseren aufmerksamen Lesern entgeht nichts:

> “Hello Michael,
>
> Your article gives a good overview of integer calculation in Gforth. I have read it completely and learned a lot.
>
> However, I think that on page 12 in the right column the terms dividend and divisor are exchanged: ud is the dividend and u1 is the divisor.
>
> With the help of Albert Nijhof I succeeded to implement the difficult 32–bit calculations needed for the temperature, pressure and humidity of the BME280–sensor on the MSP430G2553, a 16–bit processor. By comparing the result of pressure and humidity of my noForth program with a 32–bit Gforth program, the difference is very small and can be neglected in practice ...
>
> Regards,
>
> Frans van der Markt, secretary of the Dutch Forth Group.”

Stimmt. Es hätte dort bei der Erklärung von `um/mod` heißen müssen: „`ud` ist ein doppeltgenauer *Dividend*. Der wird durch den einfachgenauen *Divisor* `u1` geteilt ...“

mk

Abbildung 1: Zuiderkapel Bilthoven

HCC!Forth–bijeenkomst

> „... op de bekende locatie in het zaaltje naast de Zuiderkapel aan de Boslaan 1 in Bilthoven.“

Dort trifft man sich regelmäßig, um zu Fortheln. Im Dezember letzten Jahres ging es um:

> „DOM (Domotica met goedkoop verkrijgbare materialen, een vervolg); Klok met weekdagen ... BME280–sensor op egel en dubbele getallen in noForth.“

Um die Einladungen per Email zu bekommen, schau mal dort: `https://forth.hcc.nl`

BME280–sensor

Und in der weiteren Korrespondenz war Frans van der Markt dann so nett, sein Forth–Listing für diesen Sensor freizugeben für die Publikation hier. Da er wenig Worte darum macht, findet ihr den Forth–Code einfach so abgedruckt in unserem Heft. Fragen dazu beantwortet er gern. mk

Das Backtracking–Ballet

Neulich auf YouTube, bei der Suche nach hübschen Beispielen für Sortier–Algorithmen, gab es einen besonderen Treffer: Die *AlgoRythmics*.

> Created at Sapientia University, Targu Mures (Marosvásárhely), Romania. Directed by Kátai Zoltán, Osztián Erika, Osztián Pálma–Rozália, Vekov Géza–Károly. In cooperation with the Hungarian Opera of Cluj–Napoca (Kolozsvár), Romania. Choreographer: Jakab Melinda. Video: Fazakas Csaba–Sándor, Abodi Norbert.

Sie haben sich die Mühe gemacht und tatsächlich 10 Algorithmen getanzt. In passenden Kostümen, ausgeschmückt zu kleinen Geschichten. Da gab es das „Backtracking ballet“, eine Choreographie der „Vier Königinnen“. Die „binäre Suche“ und die „lineare Suche“ kommen als Flamenco daher. „Heapsort“, „Quicksort“, „Shellsort“ und „Bubblesort“ sind ungarische Volkstänze. Transsylvanische Sachsen tanzen einen „Mergesort“, Zigeuner den „Selectsort“ und ein romanisches Volk den „Insertsort“. Ich bin begeistert! mk

BACKTRACKING ballet choreography (The Four Queens)

`https://www.youtube.com/watch?v=R8bM6pxlrLY`

Modern Forth systems are not like their ancestors

> “There is increasing realisation that there are no magic bullets for software development. Interactive languages returned to fashion in the web developer world, which also saw a return to multi–language programming — use each language for what it’s good at. I’m going to look at one of these interactive languages, Forth, and show what people are doing with it, and why it’s a powerful tool.
>
> Modern Forth systems are not like their ancestors. The introduction of the ANS/ISO Forth

> standard in 1994 separated implementation from the language. One result was a proliferation of native–code compiling systems that preserved the traditional Forth interactivity while providing the performance of batch–compiled languages. It is the combination of performance and interactivity that makes modern Forth systems so attractive. (Stephen Pelc in: Dr. Dobb's, Modern Forth, September 11, 2008. Wiederentdeckt von Dirk Brühl, USA)"

Zum Titelbild: Mikropartikel–Emulsionen

Auf der Suche nach einem passenden Aufmacher surfte ich so durchs Internet, um was Hübsches zum *µCore* zu finden. Tante Google erzählt einem dann so Einiges über Kerne. Die ganz großen, wie das Entstehen und Vergehen von Sternen, und auch die ganz kleinen, wie die Mikropartikel.

> "Microparticles (MP) are a heterogeneous group of bioactive small vesicles (100–1000 nm) that can be found in blood and body fluids following activation, necrosis or apoptosis of virtually any eukaryotic cell."

Und dann stößt man auf interessante Forschung, auch ganz Unelektronisches, wie den programmierten Zelltod *Apoptose*, das „Suizidprogramm" einzelner biologischer Zellen oder auch die Doppelemulsionen, um Partikel zu verpacken in Form vieler kleiner Mikrokerne. Spannend!

Das nehme ich dann mal als Aufmacher, dachte ich, sieht hübsch aus, man kann *µCore* assoziieren, und es führte mich ja auch weiter. Hier der Link für andere Neugierige:

`https://www.sciencedirect.com/science/article/pii/S1471489214000952?via%3Dihub`

WeiWang, Mao–JieZhang, Liang–YinChu; Microfluidic approach for encapsulation via double emulsions. In: Current Opinion in Pharmacology Volume 18, October 2014, Pages 35-41

`https://de.wikipedia.org/wiki/Apoptose`

mk

Warum ich mit microCore angefangen habe

Vor 30 Jahren habe ich seismische Datenlogger für den Meeresgrund mit dem RTX2000 entwickelt. Der wurde aber Anfang der 90er abgekündigt, und deshalb musste ich einen anderen Prozessor benutzen. Die (unglückliche) Wahl fiel auf den brandneuen TMS320C31. Beim Testen der AD–Wandler der neuen Hardware fiel auf, dass die beiden DMA–Kanäle sich gegenseitig beeinflussten und Daten vermangelten — ein auch für Texas Instruments neuer Fehler. Ich beschloss, einen eigenen Prozessor — *microCore in VHDL* — für FPGAs zu entwickeln, aufbauend auf den Erfahrungen aus den 80ern, in denen ich bereits zwei Forthprozessoren (FRP1600 und IX1) als ASICs entwickelt hatte. Ende der 90er erschien der XC4013, der für *µCore* groß genug erschien — zu recht, wie sich auf einer von der Forth–Gesellschaft mitfinanzierten Platine erwies. Ende 2004 wurde der *µCore* dann erstmalig als Prozessor in einem Datenlogger eingesetzt und ich habe ihn seitdem in allen Folgeprojekten benutzt. Dabei änderte sich der *µCore* und seine Entwicklungsumgebung jedes Mal auf Grund der Erfahrungen, die ich dabei machte, ein bisschen. Seit Ende 2019 halte ich den *µCore* nun für ausgereift und habe begonnen, ein *microCore–Kochbuch* zu schreiben.

Klaus Schleisiek

`kschleisiek@freenet.de`

WebAssembly and Forth

Hi! Ist das ein Thema in der Szene?

`https://github.com/remko/waforth`

`https://el-tramo.be/blog/waforth/`

`https://github.com/stefano/wasm-forth`

`https://hackaday.com/2018/08/02/web-pages-via-forth/`

Jörg Plewe

Jörg, wie soll ich mit so kurzen Leserbriefen ein Heft füllen? :) Also schaun wir kurz mal rein, was das ist.

Das sind gleich zwei Leute, die Forth in WebAssembly gefasst haben.

Remko Tronçon aus Belgien hat WAForth verfasst.

> "It is entirely written in (raw) WebAssembly, and the compiler generates WebAssembly code on the fly. The only parts for which it relies on external (JavaScript) code is to dynamically load modules (since WebAssembly doesn't support JIT yet), and the I/O primitives to read and write a character to a screen. Parts of the implementation were influenced by jonesforth."

Und er liefert ein *Sieve of Eratosthenes* mit, um zu zeigen, was seine Implementation kann. Sein WAForth[1] ist da mit 6,8 s gegen Gforth mit 3,4 s ziemlich gut.

Stefano aus Italien hat WASM Forth verfasst. Er machts so ähnlich:

> "Interaction with Javascript at the moment is limited to textual input (using WasmForth.source) and output (through the write configuration parameter passed to WasmForth.boot). Using the included (optional) virtual DOM library it's possible to write interactive web apps. See the code in examples/todomvc/ for an example TODO list web app fully implemented in Forth."

mk

[1] 90 million times on MacBook Pro 2017 (3.5 GHz Intel Core i7) running Firefox 60.0.1

microCore

Klaus Schleisiek

µCore (pronounced "micro Core") is a processor written in VHDL for synthesis into FPGAs. It is optimized for embedded real time control. µCore is an embodiment of the virtual machine of the Forth programming language [1] and hence Forth is µCore's assembler.

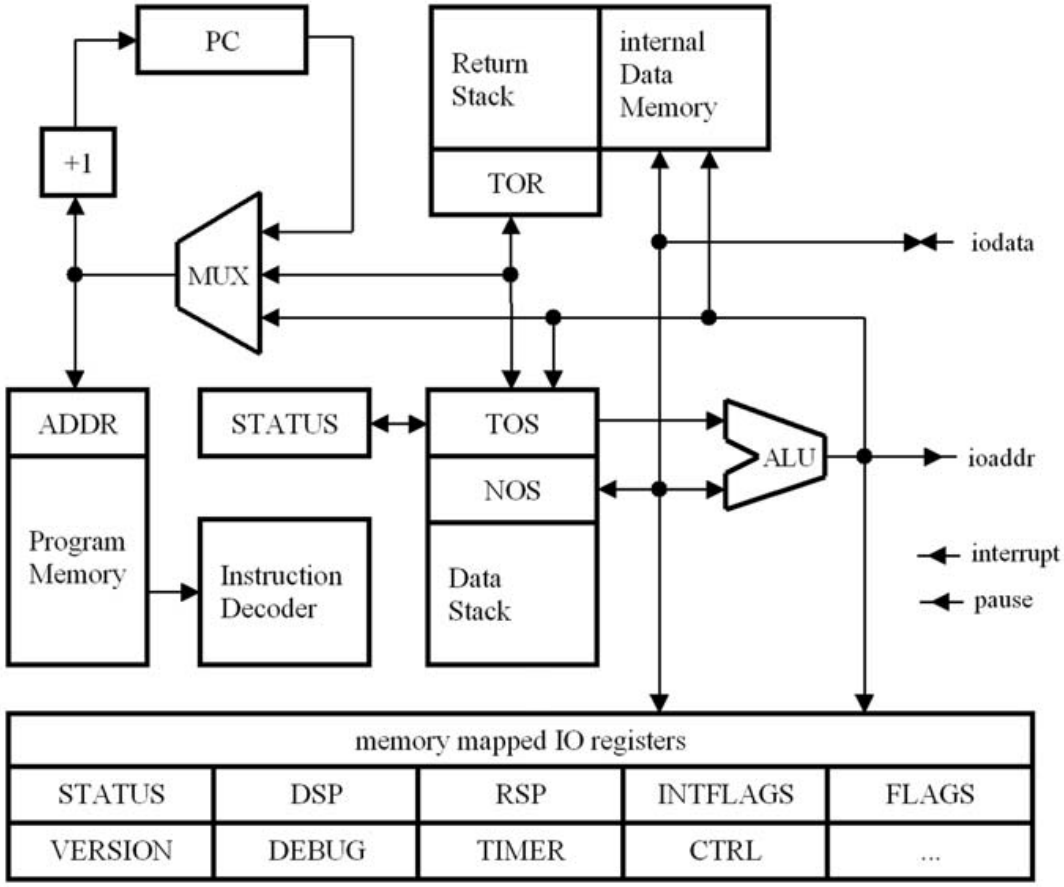


Figure 1: µCore block diagram

Characteristics

1. Dual stack (data stack & return stack)
2. Harvard architecture (8–bit wide program memory & independent data memory)
3. Configurable word width for data memory & stacks
4. Postfix instruction set architecture (literal operands/addresses precede rather than follow consuming instructions in the instruction stream)
5. Single cycle execution, no pipelining
6. Interrupt & pause traps
7. Multiple data and return stack areas for efficient multitasking
8. Deterministic program execution
9. Hardware/Software co–design environment
10. Malleable instruction set

Dual Stack

µCore does not have general purpose registers, it has a "data stack" for numerical computations and a "return stack" for program flow control.

The data stack resides in a private RAM area managed by the DataStackPointer (DSP). Its top two items are held in registers: TopOfStack (TOS) and NextOfStack (NOS).

The return stack is mapped into the data memory managed by the ReturnStackPointer (RSP). Its top item is held in a register: TopOfReturnstack (TOR).

The pros and cons of a stack versus register architecture:

- *Parameter passing.* Both architectures quite often suffer from the fact that the arguments for a subroutine call are not in the registers resp. not in the stack order, in which the subroutine expects them. As a result, registers have to be re–shuffled resp. the stack

has to be re–ordered. The number of registers is finite and hence the number of arguments that may be passed to a subroutine. The stack has a finite depth in hardware as well, but if needed, the bottom part of the stack may be swapped out and in of data memory by a background process at runtime.

- *Interrupt processing.* The stack architecture holds all arguments on stacks already and therefore, no registers need to be saved and restored.

- *Code optimization.* "Register allocation" became a research topic long ago whereas "stack allocation" is an esoteric topic. But three generations of research happened and the results are already of production quality. When µCore's assembler uForth is used, the programmer himself is responsible for stack allocation and stack re–ordering.

- *Instruction set impact.* A register architecture needs address bits in the instruction word whereas a stack architecture does without. A subroutine always takes its arguments from the top of stack downwards and leaves its results on the stack.

Looking at the technical merits, the stack architecture has several advantages especially for real time applications. Its major disadvantage is its unfamiliarity.

Harvard Architecture

µCore's instructions and the program memory are 8 bits wide. The configurable width data memory (see below) is independent of the fixed width program memory. Two instructions, which allow writing and reading program memory, may or may not be instantiated and therefore, µCore can be made safe from corrupting its program during runtime.

During cold boot µCore may always write into program memory and therefore, it may fetch its program from an external, non–volatile memory.

Configurable Data Memory & Stack Word Width

In µCore the word width of the data memory and the stacks are configurable and must be defined before synthesis as needed by the application. µCore does not deal with 8–bit "bytes". That is data memory is always accessed in units of the instantiated data memory and stack width, which simplifies the memory interface considerably.

The instantiated word width may be any odd or even number of bits. 12 bits is a practical minimum, because it limits the program memory's address range to 4096 instructions. The maximum width is only limited by the available FPGA resources. This configurability is rendered possible by µCore's postfix instruction set architecture (see below).

Postfix Instruction Set Architecture

µCore's postfix instruction set extends Forth's postfix syntax into the hardware realm. The principle has been inherited from the Transputer, and it makes the configurable data word width possible.

When instructions include immediate numbers (literal, offset, address etc.), the number of bits needed for these numbers depends on the processor's data word width. In µCore, literals and opcodes are kept separate and literals precede rather than follow opcodes that consume them. Numbers of any magnitude may be constructed by a sequence of literal instructions.

µCore's instructions are 8 bits wide. An instruction's most significant bit (MSB) determines whether it is interpreted as a literal (MSB set) or as opcode (MSB reset). A *literal flag* (LF) in the status register is set by literals and reset by opcodes. This leads to four different LF–MSB–cases for instruction interpretation:

0 0	An opcode that does not need a literal.
0 1	A literal following an opcode. The least significant 7 bits are pushed on the stack as a 2s–complement number in the range -64. . . 63, and the lit flag is set.
1 0	An opcode following a literal. If it is just a noop, the literal remains on the stack as a number. The lit flag is reset.
1 1	A literal following another literal. The literal in TOS is shifted 7 bit positions to the left and the 7 least significant bits of the instruction are appended. This covers the following number ranges: 1 lit -64. . . 63; 2 lits -8192. . . 8191; 3 lits -1048576. . . 1048575 etc. This way numbers of any magnitude can be constructed.

Opcodes do neither include literal information nor register addresses due to the stack. Therefore, each of the remaining 127 opcodes can be used for different semantics. This is plenty. The "core" opcode set has 47, the "extended" and "float" set another 29 opcodes. This leaves room for 42 application specific opcodes.

Opcodes like branch and call do even have different semantics depending on the lit flag. When it is set, they take the number in TOS as a branch offset, when it is not set, they take it as an absolute target address.

Instructions (literals as well as opcodes) are always self-contained and therefore, interrupts can be accepted after each instruction.

Single Cycle Execution, No Pipelining

In general, a single instruction needs one active clock transition to execute. On a hardware level (VHDL), most of µCore's registers have a clock and a clock enable input. The enable input allows embedding µCore into environments with a clock frequency that is above µCore's timing requirements. Constant `cycles`

in `architecture_pkg.vhd` defines the number of clock cycles, which are needed to execute one instruction. In addition, the top–level enable input may be used to temporarily halt µCore. Typically, µCore's worst–case asynchronous signal delay is less than 40 ns (25 MHz) for Xilinc, Altera, and Lattice FPGAs.

Reading the FPGA's internal blockRAM memories requires the execution of a sequence of two uninterruptible instructions. The first instruction latches the memory address inside the blockRAM, the second instruction pushes the memory's data output on the stack.

To this end a general mechanism has been invented, which allows to chain sequences of uninterruptible instructions. It can also be used for read–modify–write instructions.

Interrupt & Pause Traps

Interrupts are a well known concept in computing and almost every existing processor implements it. Hardly known is its Janus–faced sibling, the Pause, another inheritance from the Transputer. A Pause will e.g. be raised by a UART, when the processor intends to read a character, which has not been received yet. The difference between interrupts and pauses are as follows.

Interrupt

An event *did* happen that *was not* expected by the software.

On an interrupt the processor pushes the status register on the data stack, raises the InterruptInService (IIS) status bit that disables further interrupts, and executes a call to the interrupt trap location. The status register is a collection of single bit flags that characterize the processor state besides the PC.

Globally, interrupts may be enabled or disabled via the "InterruptEnable" (IE) status bit. Individual interrupt sources are enabled or disabled writing the Intflags register, which holds a flag for each interrupt source. Reading Intflags allows locating an interrupt's source(s).

Interrupt processing is terminated by the IRET instruction, which returns from the interrupt service routine and restores the status register from the data stack, thereby resetting the IIS bit again.

Pause

An event *did not* happen that *was* expected by the software.

On a pause the processor aborts the current instruction, which caused the pause signal to be raised, does not increment the Program Counter (PC), and executes a call to the pause trap location.

In a single task system, the pause trap would immediately execute an EXIT (return from subroutine), thereby returning to the instruction that caused the pause previously. This loop is repeated until the pause signal will no longer be raised.

In a multitask system, the pause trap would call the scheduler, and another task can be given the opportunity to run. Eventually, the task that caused the pause will be activated again and continue normal execution as soon as the reason for raising the pause signal has vanished, i.e. the missing event did happen.

Using the pause mechanism, resource locking can be completely realized in hardware. E.g. an *ADC* with an integrated 8–channel multiplexer in a multitasking environment can then be programmed in Forth:

```
<channel_number> ADC !
ADC @ Sample !
```

or in C[1] as follows:

```
ADC = <channel_number>;
Sample = ADC;
```

Storing the `<channel_number>` into the memory mapped ADC interface will initiate conversion of that channel and set the ADCs semaphor flag in the flags register. Should the ADC be in use by another task, the semaphor will have been set and a pause will be raised until eventually the semaphor will have been reset. In the next line, the conversion result will be read from the ADC, stored in variable `Sample`, and the semaphor will be reset. Should the ADC be still busy converting when a read attempt is made, pause will be raised until eventually the ADC has finished conversion.

This way the hardware takes care of both mutual exclusion of the ADC resource as well as waiting for the AD–conversion to finish without having to probe flags in software loops.

Multiple Data– and Return–Stack Areas for Efficient Multitasking

The number of data– and return–stack memory areas can be configured so that each task has its own set of stacks. This speeds up task scheduling, because only internal registers TOS, NOS, TOR, DSP and RSP need to be redirected. A task switch @ 25 MHz takes 7 µsec to put the current task to sleep and start another one.

Deterministic Program Execution

µCore's program execution is fully deterministic. There is no pipeline. There is no cache memory that may have to be loaded before the next instruction can be executed. Therefore, µCore's time to execute a certain sequence of instructions is predictable, which is a prerequisite for verifyable real–time systems.

[1] A µCore back–end for the LCC C compiler has been implemented for an earlier 32–bit version of µCore. In addition, LCC itself has been modified for stack– rather than register–allocation. This work has been done at Fachhochschule Aargau, Windisch (CH), supported by the Hasler Foundation.

Hardware/Software Co–Design Environment

µCore's design environment consists of the following elements:

- VHDL source code for µCore on a target system,
- the uForth cross–compiler and debugger running on a host computer,
- a umbilical link that connects host and target,
- an interactive command line interpreter to inspect and control the target,
- a disassembler,
- and a single–step tracer.

Both the VHDL hardware description as well as the uForth cross–compiler share a common file `architecture_pkg.vhd`, which characterizes µCore's architecture and opcodes. Therefore, the cross–compiler will always be in–sync with the hardware description.

The cross–compiler is able to produce program memory initialization code for the VHDL simulator and therefore, uForth programs can be simulated by VHDL simulators.

A umbilical link consisting of a 2–wire UART (RxD, TxD) connects µCore's debugger with the mating debugger on the host and allows to

- load object code into µCore for execution,
- control µCore interactively from a command line (Forth style),
- display the data stack and dump data memory to the host's display,
- upload/download data memory areas without delaying µCore's program execution,
- single step code on a subroutine level displaying the data stack at each step.

Malleable Instruction Set

µCore's instruction set consists of 47 "core", 24 "extended" and 5 "float" instructions. In addition, 26 "software traps" are available, which do a single cycle call to fixed program memory addresses. 42 opcodes are unused or software traps. They may be used for application specific instructions.

In the core version, the extended instructions will be emulated by macros or subroutines. Therefore, a core version has the same functionality as the extended version but it runs slower consuming fewer FPGA resources. Adding a new opcode to µCore is rather simple and consists of three steps:

1. Define the *binary code* for `op_newname` as a constant in `architecture_pkg.vhd`.
2. Add a new *when clause* to the instruction decoder case statement in `uCntrl.vhd` for the semantics of `op_newname`.
3. Define a *name* for `op_newname` in `microcore.fs` to make it known to the uForth cross–compiler.

The core instruction set

Data stack drop, dup, ?dup, swap, over, rot, -rot

Return stack >r, r>, r@

Branches branch, 0=branch, next, call, exit, iret

Data memory ld, st

Unary arithmetic not, 0=, 0<, shift, ashift, c2/, c2*

Binary arithmetic +, +c, -, swap-, and, or, xor, um*, *, um/mod

Flags status–set, ovfl?, carry?, time?, <

Traps reset, interrupt, pause, break, dodoes, data!

Implementation

µCore has been ported to these FPGA families:

- Xilinx (XC2S)
- Lattice (XP2)
- Altera (EP2)
- Actel/Microsemi (A3PE)

Reference implementations on a Lattice XP2–8 prototyping board with minimal external IO and hardware multiplier have been made with different µCore configurations (Tab. 1). Timing constraints in the synthesizer and in the place–and–route tool had been set to 25 MHz.

Literature

[1] Leo Brodie: "Thinking Forth", http://thinking-forth.sourceforge.net/

Appendix: Instruction Set

Opcode Description Format

names `( stack -- effect )` *type* {cycles} text.

names are the opcode names used in the uForth cross–compiler.

stack-effect indicates the effect of each opcode on the stacks. It shows a list of data stack input parameters up to the "`--`" followed by the list of output parameters. Sometimes the return stack is affected as well, which is indicated by a second set of brackets with the `rs:` prefix. In a few cases there are input or output

Instruction Set Variants	Word Width	SLICES	Data Memory	Program Memory	Maximum Clock
core	16	988	6k	8k	33 MHz
extended	16	1199	6k	8k	30 MHz
core	27	1259	4k	8k	33 MHz
extended	27	1608	4k	8k	28 MHz
extended and floating point	27	1808	4k	8k	26 MHz
core	32	1432	3k	8k	33 MHz

Table 1: FPGA resources needed by different implementations

list alternatives (e.g. `?dup`) — these are separated by the | character.

type is either a *core*, an *ext.* (extended), or a *float* (floating point) opcode.

cycles are the number of µCore cycles needed to execute an opcode. The prefix u is used to indicate uninterruptible sequences of cycles. Multicycle opcodes without u are interruptible after each cycle.

text well, that's the opcode description.

The following conventions are used for the input and output list elements:

name	semantics
n	2s–complement number
u	unsigned number
flag	0 = false, true otherwise. When flag is an output, true has all bits set.
addr	data memory address
paddr	program memory address
d	2s–complement double number occupying two stack elements. The most significant word is on top of the least significant word.
ud	unsigned double number. The most significant word is on top of the least significant word.
mask	a bit mask. Usually only one single bit will be set.
op	8–bit instruction
float	Floating point number, `data_width` wide. The mantissa takes the more significant bits, the exponent the less significant bits. Meaningful floating point operations are possible for `data_width` >= 24.
man	2s–complement mantissa.
exp	2s–complement exponent. The significant number of bits is defined by `exp_width` in `architecture_pkg.vhd`.

Opcodes

Miscellaneous

noop `( -- )` *core* {1} no operation

Data Stack

drop `( n -- )` *core* {1} drop top stack item.

dup `( n -- n n )` *core* {1} duplicate top stack item.

?dup `( n -- 0 | n n )` *core* {1} duplicate top stack item if it is not zero.

swap `( n1 n2 -- n2 n1 )` *core* {1} exchange the top two stack items.

over `( n1 n2 -- n1 n2 n1 )` *core* {1} push 2nd stack item on the stack.

rot `( n1 n2 n3 -- n2 n3 n1 )` *core* {1} move 3rd stack item to the top.

-rot `( n1 n2 n3 -- n3 n1 n2 )` *core* {1} move top stack item to 3rd position on the stack.

nip `( n1 n2 -- n2 )` *ext.* {1} drop the 2nd stack item.

tuck `( n1 n2 -- n2 n1 n2 )` *ext.* {1} equivalent to `swap over`.

under `( n1 n2 -- n1 n1 n2 )` *ext.* {1} equivalent to `over swap`.

Return Stack

>r `( n -- ) ( rs: -- n )` *core* {1} pop the top stack item and push it on the return stack.

r> `( -- n ) ( rs: n -- )` *core* {u 2} pop the top return stack item and push it on the stack.

r@ `( -- n ) ( rs: n -- n )` *core* {1} push the top return stack item on the stack.

rdrop `( rs: n -- )` *ext.* {u 2} drop the top return stack item.

I `( -- n2 ) ( rs: n1 u -- n1 u )` *ext.* {u 2} Loop index for `DO ... LOOP`s

Branches

branch (n --) *core* {1} If the lit flag is set, the next instruction is fetched from relative address PC+n, else it is fetched from absolute address n.

0=branch (flag n --) *core* {u 2} If the flag is set, continue execution at the next sequential instruction, else execute the branch instruction.

next (n --) (rs: u -- u-1) | (rs: 0 --) *core* {1 | u 2} Primitive for the FOR ... NEXT loop. If u > 0, u is decremented and the branch instruction is executed, else the top return stack item is dropped and execution continues at the next sequential instruction.

call (n --) (rs: -- paddr) *core* {1} The PC is pushed on the return stack and the branch instruction is executed.

exit (rs: paddr --) *core* {u 2} The top return stack item is dropped and execution continues at paddr.

iret (u --) (rs: paddr --) *core* {u 2} The status register is restored from u, and the exit instruction is executed.

?exit (flag --) (rs: paddr --) | (rs: paddr -- paddr) *ext.* {1 | u 2} Drop the top stack item. When flag is true, execute the exit instruction, else execution continues at the next sequential instruction.

Data Memory

ld (addr -- n addr) *core* {u 2} LoaD pushes the content of the data memory location at addr on the stack as 2nd item.

```
: @ ( addr -- n ) ld drop ;
```

st (n addr -- addr) *core* {1} STore pops the 2nd item of the stack and stores it at data memory location addr.

```
: ! ( n addr -- ) st drop ;
```

@ (addr -- n) ext. {u[2] 2} Fetch the content of the data memory location at addr.

+! (n addr --) *ext.* {u 2} Add n to the content of the data memory location at addr and pop two stack items.

lld (u -- n addr) ext. {u 2} "LocalLoaD". Index u memory cells into the return stack and push its content on the stack as 2nd item. Replace index u by the absolute memory address addr.

```
: l@ ( u -- n ) lld drop ;
```

lst (n u -- addr) *ext.* {1} "LocalSTore". Pop the 2nd item of the stack and store it into the return stack at index u. Replace index u by its equivalent memory address addr.

```
: l! ( n u -- ) lst drop ;
```

Unary Arithmetic

not (u1 -- u2) *core* {1} u2 is the bit–wise not of u1.

0= (u -- flag) *core* {1} flag is true when u equals zero.

0< (n -- flag) *core* {1} flag is true when n is negative.

Shifting with Hardware Multiplier

mshift (u1 n -- u1' u2) *core* {1} Logical single–cycle barrel shift of u1 by n bit positions. u1' is the shift result, and u2 are the remaining bits that have been shifted out of u1.
If n is negative, a right shift is executed, the most significant bit position(s) are filled with zeros, and u2 is filled from the MSB on downwards. The carry flag is set to the MSB of u2.
If n is positive, a left shift is executed, the least significant bit position(s) are filled with zeros, and u2 is filled from the LSB on upwards. The carry flag is set to the LSB of u2.
If n is zero, u1' = u1 and u2 = 0.

```
: shift ( u1 n -- u1' ) mshift drop ;
: u2/ ( u1 -- u1' ) -1 shift ;
: rotate ( u1 n -- u1' ) mshift or ;
```

mashift (n1 n2 -- n1' n3) *core* {1} Arithmetic single–cycle barrel shift of n1 by n2 bit positions. n1' is the shift result, and n3 are the remaining bits that have been shifted out of n1.
If n2 is negative, a right shift is executed, the most significant bit position(s) are filled with the sign bit of n1, and n3 is filled from the MSB on downwards. The carry flag is set to the MSB of n3.
If n2 is positive, a left shift is executed, the least significant bit position(s) are filled with zeros, and n3 is filled from the LSB on upwards. The carry flag is set to the LSB of n3.
If n2 is zero, n1' = n1 and n3 = 0.

```
: ashift ( n1 n2 -- n3 ) mashift drop ;
: 2/ ( n1 -- n2 ) -1 shift ;
```

Shifting without Hardware Multiplier

shift (u n -- u') *core* {|n|} Logical shift of u by n bit positions.
If n is negative, a right shift is executed and the most significant bit position(s) are filled with zeros.
If n is positive, a left shift is executed and the least significant bit positions are filled with zeros.
The carry flag is set to the last bit shifted out of u.

ashift (n1 n2 -- n1') *core* {|n|} Arithmetic shift of n1 by n2 bit positions.
If n2 is negative, a right shift is executed and the most significant bit position(s) of n1' are filled with the sign bit of n1.
If n2 is positive, a left shift is executed and the least

[2] u indicates that this is an uninterruptible read–modify–write instruction.

significant bit position(s) of `n1'` are filled with zeros. The carry flag is set to the last bit shifted out of `n1`.

c2/ `( u -- u' )` *core* {1} Shift right through carry. Used for multi–precision shift operations. `u` is shifted right by one bit position and the content of the carry flag is shifted into the most significant bit position of `u'`. The carry flag is set to the least significant bit of `u`.

c2* `( u -- u' )` *core* {1} Shift left through carry. Used for multi–precision shift operations. `u` is shifted left by one bit position and the content of the carry flag is shifted into the least significant bit position of `u'`. The carry flag is set to the most significant bit of `u`.

Binary Arithmetic

+ `( n1 n2 -- n3 )` *core* {1} `n3` is the 2s–complement sum of `n1` + `n2`.

+c `( n1 n2 -- n3 )` core {1} `n3` is the 2s–complement sum of `n1` + `n2` + carry flag.

- `( n1 n2 -- n3 )` *core* {1} `n3` is the 2s–complement difference of `n1` - `n2`.

swap- `( n1 n2 -- n3 )` *core* {1} `n3` is the 2s–complement difference of `n2` - `n1`.

and `( n1 n2 -- n3 )` *core* {1} `n3` is the logical and of `n1` and `n2`.

or `( n1 n2 -- n3 )` *core* {1} `n3` is the logical or of `n1` or `n2`.

xor `( n1 n2 -- n3 )` *core* {1} `n3` is the logical xor of `n1` xor `n2`.

um* `( u1 u2 -- ud )` *core* {1 | dw+3} `ud` is the double precision unsigned product of `u1` * `u2`. When no hardware multiplier is available, it takes `data_width+3` cycles to execute.

***** `( n1 n2 -- n3 )` core {1 | dw+3} `n3` is the signed product of `n1` * `n2`. The overflow flag is set when the product does not fit into single precision `n3`. When no hardware multiplier is available, it takes `data_width+3` cycles to execute.

um/mod `( ud u -- urem uquot )` *core* {dw+2} Quotient `uquot` and remainder `urem` are the unsigned results of dividing double number `ud` by unsigned divisor `u`. It takes `data_width+2` cycles to execute.

+sat `( n1 n2 -- n3 )` *ext.* {1} `n3` is the 2s–complement sum of `n1` + `n2`. In case of an overflow, `n3` is set to the largest 2s–complement number. In case of an underflow it is set to the smallest 2s–complement number. `+sat` is used to prevent control loops from oscillating in case of over/underflows.

2dup_+ `( n1 n2 -- n1 n2 n3 )` *ext.* {1} See `+`

2dup_+c `( n1 n2 -- n1 n2 n3 )` *ext.* {1} See `+c`

2dup_- `( n1 n2 -- n1 n2 n3 )` *ext.* {1} See `-`

2dup_swap- `( n1 n2 -- n1 n2 n3 )` *ext.* {1} See `swap-`

2dup_and `( n1 n2 -- n1 n2 n3 )` *ext.* {1} See `and`

2dup_or `( n1 n2 -- n1 n2 n3 )` *ext.* {1} See `or`

2dup_xor `( n1 n2 -- n1 n2 n3 )` *ext.* {1} See `xor`

m* `( n1 n2 -- d )` *ext.* {1 | dw+38} `d` is the signed double precision product of `n1` * `n2`. When no hardware multiplier is available, it takes `data_width+38` cycles to execute.

m/mod `( d n -- rem quot )` *ext.* {dw+2} Quotient `quot` and remainder `rem` are the signed results of dividing double number `d` by signed divisor `n`. `quot` is floored and `rem` has the same sign as divisor `n`. It takes `data_width+2` cycles to execute.

sqrt `( u -- urem uroot )` *ext.* {dw/2+4} `uroot` and `urem` are the root and the remainder after taking the square root of `u`. It takes `data_width/2+4` cycles to execute.
Note: Only defined for even `data_width` values.

Flags

status–set `( mask -- )` *core* {1} Status flags Carry, OVerFLow, InterruptEnable, and InterruptInService can be set or reset explicitly. E.g. the carry is set using the phrase `#c status-set`, it is reset using `#c status-reset` without modifying other flags of the status register.

ovfl? `( -- flag )` *core* {1} `flag` is true when the overflow flag is set.

carry? `( -- flag )` *core* {1} `flag` is true when the carry flag is set.

time? `( n -- flag )` *core* {1} `flag` is true when the auto–incrementing time register is larger than `n`. The time register increments every `1/ticks_per_ms` (see: `architecture_pkg.vhd`).
Note: The time register is a wrap–around counter and therefore, the maximum difference between the time register and `n` can be `2**data_width-1`, which is the upper limit for time delays.

< `( n1 n2 -- flag )` *core* {1} `flag` is true when `n1` is less than `n2` in 2s-complement representation.

flag? `( mask -- flag )` *ext.* {1} `flag` is true when the bit selected by `mask` is set in the flags register.

Program Memory

pst `( op paddr -- paddr )` *ext.* {u 2} "ProgramSTore" — `op` is stored into the program memory at `paddr`. This instruction will only be available during the cold boot phase.

```
: p! ( op paddr -- ) pst drop ;
```

pld `( paddr -- op paddr )` ext. {u 2} "ProgramLoaD" — `op` is fetched from program memory address `paddr` and stored as 2nd item on the stack. This instruction may only be available during the cold boot phase.

```
: p@ ( paddr -- op ) pld drop ;
```

Floating Point

***.** `( n1 u -- n2 )` *float* {1} Fractional multiply. 2s–complement `n1` is multiplied by coefficient `u`. `n2` is the most–significant word of the signed double product. This operator is used to compute polynomial expressions using the Horner scheme.

log2 `( u -- u' )` *float* {dw+3} `u` must be in the range `[1..2[ * 2**data_with-1`. `u'` is the logarithm dualis of `u`.

normalize `( man exp -- man' exp' )` *float* {< dw-2} This is an auto–repeat instruction. In each step, `man` is shifted one position to the left and `exp` is decremented by one until (a) the mantissa's sign bit and its second most significant bit have different values, or (b) `exp` has reached its minimum value. In the end, `man'` and `exp'` are normalized versions of `man` and `exp`. It takes at most `data_width-2` cycles to execute.

>float `( man exp -- float )` *float* {2} After normalization the pair `man`, `exp` is packed into floating point number `float`.

float> `( float -- man exp )` *float* {1} Unpack floating point number `float` into `man` and `exp`.

Traps

reset `( -- )` *core* {1} Hardware trap.

interrupt `( -- )` *core* {1} Hardware trap.

pause `( -- )` *core* {1} Hardware trap.

break `( -- )` *core* {1} Soft trap for single–step tracing.

dodoes `( addr -- addr+1 )` *core* {1} Soft trap compiled by `DOES>`.

data! `( addr n -- addr' )` *core* {1} Soft trap used for data memory initialization.

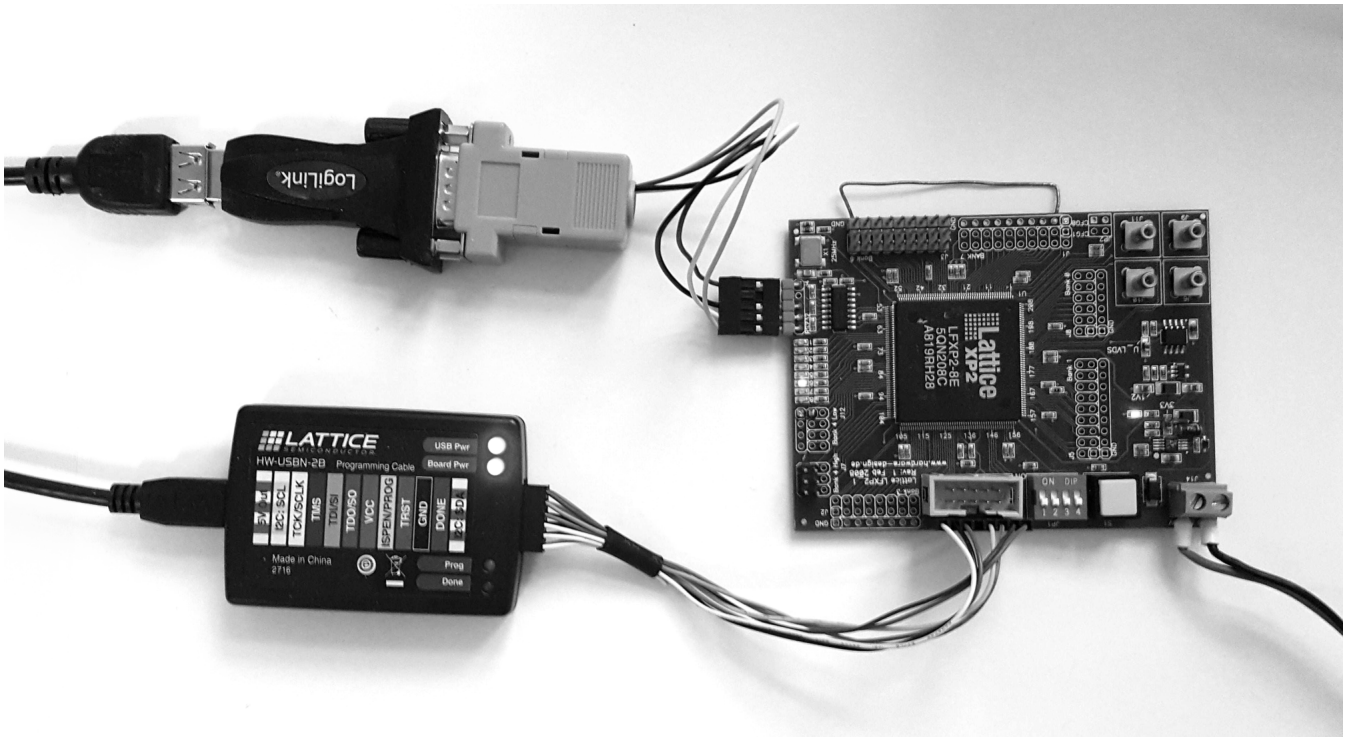


Figure 2: Lattice XP2–8 prototyping board with FPGA programmer (bottom left) and UART umbilical (upper left) connected.

Namespaces and Context Switching for Mecrisp–Stellaris

Manfred Mahlow

Using namespaces is encouraging concerning factoring of code, not only into words but also into well structured source code modules and implicit context switching makes it easy to define data types, data structures, classes and objects. Mecrisp–Stellaris does not support namespaces and context switching on its own. So it became a candidate for an attempt to create a loadable extension that adds both.

Since I discovered Forth and became familiar with the vocabulary concepts of Fig–Forth and F83, I was missing a way to bind a vocabulary (a namespace, a wordlist) to a word representing data, to create a data type or a simple object.

Over the years I found a way to do it without using state–smart words, first for PC Forth Systems and during the last years also for flash based MCU Forth Systems [2]. VOCs, ITEMs and STICKY words are the core elements of the final concept [3][4].

Mecrisp–Stellaris

Mecrisp–Stellaris [1] was a very special candidate to add namespaces and context switching because of its special features and the many different supported targets. So I decided not to add the required modifications and extensions to the Mecrisp–Stellaris core but to concentrate all into an external source code file, to be loaded on demand. Only one hook, called `HOOK-FIND`, added by Matthias Koch, is used, to link the extension into the Mecrisp core.

VOCs, ITEMs and STICKY Words

The extension adds support for wordlists, vocabulary prefixes (VOCs), implicit context switching words (ITEMs), STICKY words and related TOOLS.

The source code file is called
`vis-x.y.z-mecrisp-stellaris.txt`
and can be downloaded from [5].

The extension is compiled to flash. Before loading the file execute `eraseflash` *to reset Mecrisp–Stellaris to its original state.*

The extension should work for most Mecrisp–Stellaris flavors. The only requirement is, that the header of a word compiled into flash (or RAM) is placed without a gap directly after previously comma–compiled numbers (tags).

Wordlists

Wordlists are implemented using tags not separate linked lists.

Two Search Orders

Two search orders are used, a permanent and a temporary one.

The permanent search order is the default search order like the one in Standard Forth Systems. It defaults to wid(forth–wordlist) wid(forth–wordlist) wid(root–wordlist).

The temporary search order is set and activated by vocabulary prefixes.

A vocabulary prefix (VOC) is a wordlist handler, an immediate word, created with an initially empty wordlist.

Explicit Context Switching

The execution semantics of a vocabulary prefix is to activate the temporary search order with its wordlist identifier (wid) on top.

The temporary search order always holds two wordlist identifiers, the wid of the vocabulary prefix that activated the search order and the wid of the root–wordlist.

An active temporary search order is deactivated after the next word from the input stream is found and compiled or executed.

No F83 Vocabularies

The top of an active temporary search order, the wid of the executed vocabulary prefix, can be copied to the permanent search order (see <voc> `FIRST` , <voc> `ALSO`). So vocabulary prefixes can fully replace F83 style vocabularies and `FORTH` and `ROOT` are then vocabulary prefixes which are also members of the permanent search order. [4]

Implicit Context Switching

By assigning the execution semantics of a vocabulary prefix as extra immediate execution semantics to a new word, when it's created, a context switch will be done in interpret or compile mode after the words execution semantics is executed or compiled. May sound complicated but it's easy to use, e.g.:

```
VOC DS1621  DS1621 DEFINITIONS
```

```
... \ DS1621 methods go here

FORTH DEFINITIONS

DS1621 ITEM 0 CONSTANT TS1
```

`TS1` is an implicit context switching word, a combination of a `VOC` (its execution semantics) and a `CONSTANT`.

In interpret/compile mode `TS1` is executed/compiled as `CONSTANT` and afterwards the `DS1621` namespace (the temporary search order) is activated.[1]

The next word, follwing `TS1` in the input stream, is then not looked up in the `FORTH` but in the `DS1621` namespace.

Implementation Notes

This implementation is based on an earlier one published as vocs–0.7.0 in the Mecrisp–Stellaris experimental subdirectory since release 2.4.

Wordlists are implemented by assigning a wordlist identifier (wid) as a tag (wtag) to every created word.

Search Orders The Mecrisp–Stellaris dictionary consists of two linked lists, one for the words in flash and one for the words in RAM. In the compiletoflash mode the words in RAM are hidden. This raised the question, how to implement and handle the search orders. I decided to use two sets of search orders, one for each compile mode. Otherwise all words that manipulate the search orders had to be made compile mode smart and that seemed to be more confusing than having separate search orders for the two modes.

Implicit Context Switching is implemented by assigning a wordlist identifier (wid) as a tag (ctag) to every context switching word (see ITEM).

Glossary of new or modified words

Used symbols

I immediate word

D defining word

S sticky word

<voc> a vocabulary prefix

<voc> ' ("name" – xt)

Find the word `name` in the temporary <voc> search order and return its xt.

<voc> ['] ("name" –) I

Find the word `name` in the temporary <voc> search order and compile its xt as a literal.

[1] The context switching semantics of the `VOC` is never compiled.

\WORDS (–)

Synonym for the Mecrisp–Stellaris word WORDS. Lists all words ignoring the namespaces. May only be useful for some debugging cases.

?? (–) S

A dictionary browser. Shows the words of the first wordlist in the search order, the actual search order, the compiler context, the compiler mode and the data stack. Does not change the actual search order.

@VOC (–) I

Activate the temporary search order with the wordlist identifier of the current compiler context on top.

ALSO (–) I

Double the top of the permanent search order. Abort with an error message if the permanent search order is full. See also ONLY and PREVIOUS.

<voc> ALSO (–) I

Copy the top wid from the temporary <voc> search order on top of the permanent search order. Abort with an error message if the permanent search order is full. See also FIRST , ONLY and PREVIOUS.

<voc> DEFINITIONS (–) I

Make the wordlist of the vocabulary prefix <voc> the new compilation context.

FIND (c-addr len – xt flags)

The new namespace and context switching aware FIND.

Resolve a pending context switch request. Search for the word with name `c-addr,len` in the dictionary. If the word is not found return `xt`=0 and `flags`=0. If the word is found return its `xt` and `flags` and register a context switch request, if it is a context switching one (see ITEM).

<voc> FIRST (–) I

Overwrite the top of the permanent search order with the top of the temporary <voc> search order.

FORTH (–) I

A vocabulary prefix, the wordlist handler for the forth–wordlist.

INSIDE (–)

A vocabulary prefix, the wordlist handler for the inside–wordlist.

It's a wordlist for words used for the implementation but rarely used for applications. Please look at the source code for further information.

ITEM (–)

A prefix for defining words. A prefixed defining word adds the execution semantics of the last executed vocabulary prefix as an extra immediate execution semantics to the next created word, to make it a context switching one. The word is not made state–smart!

ITEMS (–)

List all context switching ITEMs (implicit context switching words) that are defined in the dictionary.

ONLY (–) I

Reset the permanent search order to its default: FORTH FORTH ROOT.

PREVIOUS (–) I

Remove the top wid from the permanent search order.

ROOT (–) I

A vocabulary prefix, the wordlist handler for the root–wordlist. The root–wordlist holds those words that need to be visible in the permanent and in the temporary search order.

STICKY (–)

A prefix for defining words. A prefixed defining word adds an extra immediate execution semantics to the next created word so that the temporary search order, the word is found in later, is hold, until the next word from the input stream is compiled or executed. The word is not made state–smart!

VOC ("name" –) D

Create a vocabulary prefix with a new empty wordlist. Use `name DEFINITIONS` to make the new wordlist the current compilation context.

<voc> VOC ("name" –) D

Create a vocabulary prefix that extends (inherits) the namespace of the vocabulary prefix <voc>.

VOCS (–)

List all VOCs (vocabulary prefixes) that are defined in the dictionary.

WORDS (–)

List all words of the top wordlist of the permanent search order.

<voc> WORDS (–)

Lists all words of the temporary <voc> search order.

.. (–) I

Switch back from a temporary search order to the permanent search order.

Referenzen

[1] *Mecrisp — Native code Forth for MSP430 and ARM Cortex*

http://mecrisp.sourceforge.net/

[2] Manfred Mahlow, *Namespaces and Context Switching for a Tiny Forth*, Forth–Magazin Vierte Dimension 4/2019

[3] Manfred Mahlow, *VOCs ITEMs und STICKY Words*, Forth–Tagung 2019

https://wiki.forth-ev.de/lib/exe/fetch.php/events:ft2019:vocs_items_und_sticky_words.pdf

[4] Manfred Mahlow, *VOC statt VOCABULARY,* Forth–Tagung 2019

https://wiki.forth-ev.de/lib/exe/fetch.php/events:ft2019:voc_statt_vocabulary.pdf

[5] *Forth Namespaces and Context Switching based on VOCs, ITEMs and STICKY Words*

https://wiki.forth-ev.de/doku.php/projects:forth-namespaces:start

VIS HOWTO 01 — Vocabularies and Libraries

Manfred Mahlow

This is the first of a number of HOWTOs showing the potential of the Forth VIS concept [1]. VIS stands for VOCs, ITEMs and STICKY Words. The basic idea of the VIS concept is to handle named wordlists in a less statical way by using vocabulary prefixes (VOCs) instead of F83 style vocabularies (VOCABULARYs) [2]. Then named wordlists can not only be used to implement vocabularies but also to implement libraries. Both are named wordlists but the access to the wordlist members makes the difference.

Vocabularies

Vocabularies are used for tools like assembler, disassembler, editor ... or to hide words required for implementation purposes but not for applications.

VOC versus VOCABULARY

Listing 1 shows how to create a vocabulary using `VOC` instead of `VOCABULARY`.

Listing 1: VOC vs. VOCABULARY

```
1
2   FORTH DEFINITIONS
3
4   \ order:  FORTH FORTH ROOT : FORTH
5
6   \ VOCABULARY ASSEMBLER        \ F83-style
7     VOC ASSEMBLER               \ VOC-style
8
9   \ order:  FORTH FORTH ROOT : FORTH
10
11  \ ASSEMBLER DEFINITIONS        \ F83-style
12    ASSEMBLER FIRST DEFINITIONS  \ VOC-style
13
14  \ order: ASSEMBLER FORTH ROOT : ASSEMBLER
15
16  \ Assembler words go here.
17
18  \ FORTH DEFINITIONS        \ F83-style
19    FORTH FIRST DEFINITIONS  \ VOC-style
20
21  \ order: FORTH FORTH ROOT : FORTH
22
```

The difference between the `VOC` and `VOCABULARY` method is, that a `VOC`–style vocabulary must be explicitly added to the search order, using `FIRST` or `ALSO` and `PREVIOUS` (Listing 1, line 12) [2].

A side effect of using a `VOC` to create a vocabulary is, that any word of the vocabulary can be accessed by prefixing its name with the vocabulary name, no matter whether the vocabulary is a member of the search order or not.

What is a side effect here, makes it very easy to create libraries and that was the leading objective when implementing vocabulary prefixing.

Libraries

Libraries are named wordlists like vocabularies but with a different word access mode.

Listing 2: Creating a Library

```
1
2   FORTH DEFINITIONS
3
4   \ order:  FORTH FORTH ROOT : FORTH
5
6   VOC NAME  NAME DEFINITIONS
7
8   \ order:  FORTH FORTH ROOT : NAME
9
10  \ Library words go here:
11
12    : WORD1 ... ;
13    : WORD2 ... ;
14
15  FORTH DEFINITIONS
16
17  \ order: FORTH FORTH ROOT : FORTH
18
```

The access to the library words is not enabled by adding the library to the search order but by using the library name as a word prefix, e.g. `NAME WORD1` or `NAME WORD2 .`

Like a vocabulary a library may be added to the search order but that creates the risk of hard to find errors when a library word hides a word of another namespace in that search order.

Referenzen

[1] Manfred Mahlow, *VOCs ITEMs und STICKY Words*, Forth–Tagung 2019

https://wiki.forth-ev.de/lib/exe/fetch.php/events:ft2019:vocs_items_und_sticky_words.pdf

[2] Manfred Mahlow, *VOC statt VOCABULARY*, Forth–Tagung 2019

https://wiki.forth-ev.de/lib/exe/fetch.php/events:ft2019:voc_statt_vocabulary.pdf

[3] Manfred Mahlow, *Namespaces and Context Switching for a Tiny Forth*, Forth–Magazin Vierte Dimension 4/2019

[4] Manfred Mahlow, *Namespaces and Context Switching for* Mecrisp-Stellaris, Forth–Magazin Vierte Dimension 1/2020

Synonym

U. Hoffmann, M. Kalus

Im Dezember letzten Jahres erreichten mich Weihnachtsgrüße aus den Niederlanden. Willem Ouwerkerk *schrieb dazu: „Naar een idee van Albert Nijhof een klein noForth programma ter afsluiting van 2019. ... Just run it :)“ Das wurde natürlich sogleich ausprobiert. In noForth auf einem MSP430G2553 Launchpad klappte es sogleich. Auf dem Laptop in Gforth unter Linux Mint wollten wir es aber auch mal laufen lassen. Da gab es dann aber Hindernisse. Und bei genauerer Betrachtung wird es sogar kniffelig. Hier also mal die Wiedergabe der Reise durch die Klippen des Forth.*

Value ist nicht gleich Value

An diesem *defining word* scheiden sich die Geister[1]. Die einen initialisieren den `value` schon bei seiner Definition, andere später irgendwann mal. Dementsprechend haben wir im noForth dieses Verhalten:

```
value ( "name" -- )
```

Doch im Gforth und Forth–94 und Forth–2012 geht es so:

```
value ( n "name" -- )
```

So gibt der noForth–Quellcode `xmas.f` in Gforth sogleich eine Fehlermeldung „Stack underflow“.

Der Workaround an der Stelle ist aber einfach:

```
: value ( n -- ) 0 value ;
```

Das geht, weil `value` nicht immediate ist und immer compiliert, aus einer Definition heraus ebenso wie von der Kommandozeile aus. Also so wie `variable` und `constant` auch.

Parsing words

Den nächsten Fehler erzeugt das `ch` Wörtchen. Es ist ein interpretierendes Wort, statesmart[2], das den *input buffer* scannt und den ASCII–Wert des nächsten Zeichens als Literal compiliert. Klingt kompliziert? Ist es auch. Es ist syntaktischer Zucker, mit dessen Hilfe man im Quellcode erreicht, das Zeichen auf den Stack zu kriegen, um es von `emit` ausgeben zu lassen. Sonst müsste man dafür ja jedesmal solche Klimmzüge machen:

```
[ <ascii-wert> ] literal
```

Also mit `[` den Compiler ausschalten, ASCII–Wert in der Tabelle nachschauen und eintragen, damit er auf den Stack zu liegen kommt, mit `]` Compiler wieder einschalten und den TOP–Stackwert als Literal compilieren. Dazu müsste man das alles aber erst mal wissen, also das System „von innen“ kennen.

Da ist es schon hübscher, eine Sequenz zu haben wie:

```
char x emit
```

um den ASCII–Wert des Zeichens `x` auszugeben. Und noch eine Stufe komplizierter wird es dadurch, dass, je nachdem in welchem „State“ Forth ist, compilieren oder interpretieren, was anderes passiert mit dieser Sequenz. Denn beim Interpretieren genügt es ja, das `x` auf den Stack zu bekommen, von wo es durch `emit` konsumiert wird. Andernfalls muss es ja erst noch in eine Definition gespeichert werden, von der aus es dann später, zur Laufzeit des Wortes, auf den Stack befördert wird, damit `emit` es benutzen kann.

Der State ist also von Bedeutung. Nun kann man solch ein parsendes Wort state–smart anlegen, dann muss sich der Programmierer keine Gedanken um den State machen — so geschehen im noForth, welches dadurch recht komfortabel ist, dafür, dass es ein so kleines System für eine MCU ist.

Oder das Forthsystem stellt für jeden State ein eigenes Wort bereit. So, wie es im Gforth der Fall ist. Da haben wir `char` und `[char]` dafür. Traditionell deutet die eckige Klammer an, dass so ein Wort für den *compiling state* gedacht ist.

Funktionen umbenennen

Nun, solche Probleme sind gar nicht so selten. Das Forth–2012 standardisierte daher ein Wort, mit dem man eine Funktion umbenennen kann[3]:

```
synonym
( "<spaces>newname" "<spaces>oldname" -- )
```

`synonym` erzeugt eines neues Wort mit dem Namen `newname`, das sich so wie das Wort `oldname` verhält. Dabei weiß `synonym`, ob das bisherige Wort *immediate* ist oder nicht und definiert das neue Wort entsprechend. Natürlich muss man da schon wissen, welche Funktion dem neuen Wort zugeschrieben werden soll.

Weiß man das, dann kann man für unseren Fall also

```
synonym ch [char]
```

schreiben, denn `ch` wird im Programm nur im Compile–Mode verwendet.

[1] Der Forth–94–Standard ist hier eindeutig. Doch noForth hält sich an dieser Stelle nicht dran. Kann man machen, aber dann hat man eben diese Probleme hier und der Code ist nicht ohne weiteres portabel.

[2] Nach Auskunft der noForth–Entwickler. Historisch gesehen war `ASCII` mal dafür vorgesehen; es war *immediate* und *statesmart*. `[char]` ist *immediate*, aber *nicht–statesmart*, und `char` ist *nicht–immediate* und *nicht–statesmart*. noForth hat neben `ch` auch `[char]` und `char`, sodass man nach Gusto wählen kann.

[3] In meiner Version enthalten: Gforth 0.7.9_20190919

Weiß man als Programmierer selbst, ob das bisherige Wort im vorliegenden Forth–System immediate ist — `[char]` ist es in Gforth — so kann man auch schreiben:

```
' [char] alias ch immediate
```

Für `[char]` kann das aber je nach System auch anders sein!

Standard–System erhalten

Nach der Value–Redefinition auf diese Weise liegt kein Standard–Forth–System mehr vor. Das ursprüngliche Wort des Systems ist ja nun nicht mehr auffindbar. Das kann man umgehen, indem das Ganze in ein Vokabular verpackt wird. Und nur das Wort `xm` danach außerhalb sichtbar ist[4]. Also so:

```
vocabulary noForth-xmas
noForth-xmas definitions

: value ( -- ) 0 value ;
synonym ch [char]

\ noForth Code
...

only forth also definitions
noForth-xmas
synonym xm xm
Forth
```

Mit diesem Manöver ist `xm` in der Wordlist des Standard-Forth–Systems zu finden. Und alle ursprünglchen Worte des Forth–Systems sind weiterhin zugänglich.

So läuft dann das originale noForth–Programm auch im Gforth.

```
                  ><
                  Oi
                 //\\
                ///\\\
               ////\O\\
              ////O\\i\\
             //////\O\\i\
            ii////i\\\\\\\
           ////////\\\i\\\\
          /////////\\\\\\i\\
         ///i//////\\\\\ii\ii
        ///i///////\\\\\\i\\\\
       //i/i//i//i/\i\\\\\\\\\\
      ///i///////i/i\\\\O\\\\\\\
     ///////////i//\\\\\\\\\\\\\\
    /////ii/////i//\i\i\\\i\\\\\\i
  ////i//////i////\\O\\\\\i\\\\\\\
                  ||
       Many Christmas greetings
```

[4] `xm` : siehe Listing

Pseudo–Random

Eine interessante Beobachtung am Rande: Belässt man `setup-random` im Hauptwort `xm`, wird die Random–Generierung bei jeder Ausführung von `xm` wieder von vorn begonnen — und liefert so immer den gleichem Tannenbaumschmuck! Erst wenn man die Initialisierung ausklammert und nur einmal ausführt, kann man mit `xm` jedem Familienmitglied und den Freunden einen eigens geschmückten Baum übermitteln. Nach erneutem `setup-random` erhält man dann wieder dieselbe Serie.

Woran man sehen kann, dass es sich in der Tat um eine Pseudo–Random–Funktion handelt, die da auf diese Weise implementiert worden ist. Für den Zweck vermutlich ausreichend. Will man mehr, muss man etwas mehr Aufwand treiben.

Links

`http://forth-standard.org/standard/tools/SYNONYM`

`http://home.hccnet.nl/anij/nof/noforthdocumentation.pdf`

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

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

Listings

Listing 1: xmas.f

```
1   \ noForth christmas greetings for 2019
2
3   \ Work location for pseudo random numbers
4   value RND
5
6   decimal
7   \ Make pseudo random number ud
8   : RANDOM        ( -- ud )
9     rnd 31421 *  6927 +  dup to rnd ;
10  : CHOOSE        ( u1 -- u2 )
11    random  um* nip ;
12  : SETUP-RANDOM  ( -- )
13    31414 to rnd ;
14  hex
15
16  : DRESS  ( -- )
17    3 choose if ch i else ch O then ;
18  : DRESS1 ( -- )
19    9 choose 0= if dress else ch / then ;
20  : DRESS2 ( -- )
21    9 choose 0= if dress else ch \ then ;
22
23
24  : XM    ( -- )
25      setup-random  base @  cr
26      dup spaces [char] > emit [char] < emit
27      0 over 1- ?do  cr dup i   dup 1+ spaces
28          - dup   0 do dress1 emit loop
29                  0 do dress2 emit loop
30      -1 +loop   cr spaces ." ||" cr
31      ."      Many Christmas greetings" cr ;
32  xm
33
```

Read the BME280 Pressure, Temperature and Humidity Registers

F. L. van der Markt

[Im Anschluss an einen Chat im Dezember letzten Jahres hat der Autor seinen Forth–Code für den BME280–Sensor veröffentlicht. Zur Darstellung der Messwerte sind einige kniffelige Umrechnungen nötig. Sein Code ist reich kommentiert, sodass wir den geneigten Leser bitten, ohne Umschweife darin einzutauchen. Er ist verfasst in noForth für den MSP430G2553. Die Berechnungen wurden mit C–Code verifiziert, hier nicht abgedruckt, aber erhältlich beim Autor. mk]

Links

f.l.van.der.markt@kader.hcc.nl

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

Listings

```
\ Read the BME280 registers
\ Pressure, Temperature and Humidity.
\
\ Program for the MSP430G2553 launchpad or egelkit
\ F.L. van der Markt
\ 12-12-2019
\ pressure-testen
\ Uservalue for Hsl, with the code of Albert Nijhof
\ If you want to change uservalues, example:
\    11 to Hsl
\    35 to Node-ID  etc.
\    USAVE
\ With USAVE the current values are copied
\ from RAM to flash.
\ When you start the measurements with test,
\ these uservalues are restored to RAM by UREAD
\
\ ?abort, ?error, catch, D>S of Albert Nijhof
\ READ_INT reads now 16 bits,
\ READ_WORD 16 bits >>1
\ P15 is not used anymore, but the full adc_P
\ (20 bits).
\ DIGT*, DIGP* and DIGH* replaced with 3 arrays.
\ Temperature, airpressure and humidity now
\ calculated according to the programs in the
\ datasheet of the BME280/GY-BME280
\ No debug output, model approximation removed!
\ */ replaced by (u)scale routine
\ ------------------------------------------------

\ user words:
\ TEST  displays temperature, pressure and humidity
\       combined with some intermediate results
\       every 5 minutes.
\ TEST1 reads 10 times the ID of the BME280

\ Load first the bitbang I2C routines
\ for MSP430 code variant, without assembler:
\ 'bitbang-i2c p22-p24.f'
\ before loading this file!
\
\ Connect P2.4 to SDA and P2.2 to SCL
\ Remove also the jumper at p2.4 (LED Power)

\ Note:
\ pullup resistors are already integrated
\ in the BME280 sensor.


HEX

EC constant BME280  \ its I2C-address ( hex EC)
BME280 I2C?   \ should print -1
              \ if I2C connections are OK

\ configuration-registers
E0 CONSTANT BME_RESET
F2 CONSTANT BME_HUMI
F3 CONSTANT BME_STATUS
F4 CONSTANT BME_CTRLM
F5 CONSTANT BME_CONFIG

: ARRAY ( n ccc -- )
create cells allot immediate
does> @ char 1- hx 0F and 2* + postpone literal ;

9 array DIGP
3 array DIGT
6 array DIGH

value P15     \ P15 = ADC_P>>5, 15-bit pressure

value T15     \ T15 = ADC_T>>5, 15-bit temperature
value TCEL    \ Temperature in degrees Celsius
value VAR1    \ used for compensation COMP-T
value VAR2    \ ..
value VAR3
value T_FINE  \ ..

value H15     \ H15 = ADC_H/2, 15-bit humidity
value HUM_PM  \ Humidity in promille
value TMP

decimal

\ <><> Albert Nijhof ------>
\ user-values saved in flash
\ with special procedures
value Hsl    6 to Hsl \ Height above sealevel in m
value Node-ID   53 to Node-ID
value Child-ID  2 to Child-ID
: USAVE ( -- )
   [ -1 , ]     \ wordt PROGRAMMA\
   adr Hsl 6 m, \ Inhoud values -> flash (ROM)
   0 ,          \ Afsluiter (nodig voor CHERE
   freeze
;
: UREAD ( -- )
   chere cell-
   dup @ ?abort ( msg from UREAD ) \ beveiliging
   6 - adr Hsl 6 move \ ROM-waarden into RAM
;
\ <------------

```

```
\ create a double variable,
\ name in flash, value in ram.
\ using the name of the variable
\ puts its ram-address onto the stack
: 2variable ( 'name' -- )
  create 2 cells allot  \ reserve 2 cells, 4 bytes
  does> @               \ get ram-address
;

\ store double value in double variable
\ lo = low word, hi = high word
: 2!  ( lo hi adr -- )
  dup >r           \ S: lo hi adr     R: adr
  !                \ S: lo            R: adr
  r> 1 cells + !
;

\ fetch value from double variable
: 2@ ( adr -- lo hi )
  dup              \ adr adr
  @                \ adr hi
  swap 1 cells +   \ hi adr+2
  @ swap           \ lo hi
;

: D-  ( D1 D2 -- D3 ) \ D3 = D1 - D2
  DNEGATE D+
;

: M/ ( d n1 -- n2 ) \ rounded division
  dup >r 2/ s>d d+ r> fm/mod nip ( n2 = quot )
;

\ <><> Albert Nijhof ------>
: D*/ ( dn1 x +y -- dn2 )
    >r dup abs >r   \ lo hi
    ?dnegate     \ teken van dn1 aanpassen als x<0
    tuck            \ teken bewaren
    dabs            \ +lo +hi
    swap r@ um*     \ +hi lo mi
    rot  r> um*     \ lo mi mi hi
    rot 0 d+  \ lo mi hi -- 48bits tussenresultaat
    r@ um/mod       \ lo rest +hi
    r> 2>r          \ lo rest
    r> um/mod       \ rest +lo
    nip r>          \ +lo +hi
    rot ?dnegate ;  \ lo hi -- teken terugzetten

(*
    dn1 maal x/y levert dn2 op.
    dn1 en x mogen negatief zijn,
    y moet positief zijn.
    Voorbeeld:
    -100000. 3000 4000 d*/ d. <rtn> -75000  OK
    Geen afrondingscorrectie.
*)


\ double d to 16-bits number
\ only possible if d fits in a single number !
: D>S ( lo hi -- lo )
    over s>d        \ lo hi lo hi2
    nip             \ lo hi hi2
    <> ?abort
;

: ?ERROR ( throw# -- )
    if ."  Error in " dup count hx 1F and
    type cr then ;
hex
\  alle shifts ( d1 n -- d2 )
code DLSHIFT   4706 ,
4437 ,  9346 ,  2405 ,  54A4 ,
   0 ,  6707 ,  8356 ,  23FB ,   next  end-code

code DRSHIFT   4706 ,
4437 ,  9346 ,  2405 ,  C312 ,
1007 ,  1024 ,  8356 ,  23FB ,   next  end-code

code DARSHIFT   4706 ,
4437 ,  9346 ,  2404 ,  1107 ,
1024 ,  8356 ,  23FC ,   next  end-code

decimal

: scale ( x num den -- result ) \ x, num, den  >0
   >r m*
   r@ 2/ s>d d+ \ x*num + den/2
   r> fm/mod nip
   dup 32767 u>
     if ." SCALE overflow! " dup . cr then
;

(*
: USCALE ( x teller noemer -- y )
\ unsigned */ met afronding
\ alleen gebruiken als alle stackparameters
\ positief zijn !
  >r um*
  r@ 0 d2/ d+  \ heft van deler erbij
  r>  2dup u< 0=
    if ." USCALE overflow!" dup . cr then
  um/mod nip
;
*)

\ <><>  <---  Albert Nijhof

\ variables for humidity and pressure calculations
variable t_finex  \ t_fine /10
2variable adc_H
2variable var_h
2variable vx1
2variable vx2
2variable vx3
2variable vx4
2variable adc_P
2variable p100 \ pressure in hPa*100


hex

: READ-STATUS ( -- status )
    START-BIT
    BME280 BYTE-OUT     \ write-address BME280
    BME_STATUS BYTE-OUT \ select register F3 status
    START-BIT
    BME280 1+ BYTE-OUT  \ read-address BME280
    BYTE-IN             \ read F3 register
    NACK-BIT
    STOP-BIT
;

: WAIT-ADC-READY ( -- ) \ Wait until ready
  BEGIN
     10 MS
     READ-STATUS 8 AND  \ bit 3
     0 = KEY? OR
  UNTIL
;

: WAIT-NOCAL ( -- )    \ Wait until not reading
                       \ calibration
  BEGIN
     10 MS
     READ-STATUS 1 AND  \ BIT 0
     0 = KEY? OR
  UNTIL
;

: CLEAR-DATA ( -- )
  0 TO P15
```

```
    0 TO T15
    0 TO H15
;

: SETUP-BME280 ( -- )
\ reset sensor
    START-BIT
    BME280 BYTE-OUT
    BME_RESET BYTE-OUT
    B6 BYTE-OUT
    STOP-BIT

\ configure humidity oversampling 4x
    START-BIT
    BME280 BYTE-OUT
    BME_HUMI BYTE-OUT
    3 BYTE-OUT
    STOP-BIT

\ configure oversampling 4x pres and 4x temp
\ and normal mode
    START-BIT
    BME280 BYTE-OUT
    BME_CTRLM BYTE-OUT
    6F BYTE-OUT
    STOP-BIT

\ configure for one measurement per 500 ms
\ and filter off
    START-BIT
    BME280 BYTE-OUT
    BME_CONFIG BYTE-OUT
    80 BYTE-OUT
    STOP-BIT
;


: BME280-ID ( -- ) \ should return 0x60
    START-BIT
    BME280 BYTE-OUT
    D0 BYTE-OUT \ select ID register
    START-BIT
    BME280 1+ BYTE-OUT    \ switch to READ-MODE
    BYTE-IN NACK-BIT      \ Read ID
    STOP-BIT
    ." ID = " . cr
;


: READ-ALL ( -- ) \ Read all measured values
    CLEAR-DATA
    WAIT-ADC-READY
    START-BIT
    BME280 BYTE-OUT  \ write address BME280
    F7 BYTE-OUT      \ select register PRES_MSB
    START-BIT
    BME280 1+ BYTE-OUT   \ switch to read-mode
    BYTE-IN ACK-BIT      \ F7 PRES_MSB  8 bits
    BYTE-IN ACK-BIT      \ F8 PRES_LSB  8 bits
    BYTE-IN ACK-BIT  \ F9 PRES_XLSB 4 highest bits
    BYTE-IN ACK-BIT      \ FA TEMP_MSB
    BYTE-IN ACK-BIT      \ FB TEMP_LSB
    BYTE-IN ACK-BIT DROP \ FC TEMP_XLSB not used
    BYTE-IN ACK-BIT      \ FD HUMI_MSB
    BYTE-IN NACK-BIT     \ FE HUMI_LSB
    STOP-BIT
    1 rshift SWAP
      7 lshift + ABS TO H15  \ HUM  >> 1
                             \ ( 15 bits left )
    1 rshift SWAP
      7 lshift + ABS TO T15  \ TEMP >> 5
                             \ ( 15 bits left )
    4 rshift s>d vx1 2! ( pres_xlsb >> 4 )
    4 lshift s>d vx2 2! ( pres_lsb << 4 )
    s>d dm 12 dlshift
    vx2 2@ d+ vx1 2@ d+ ( pres_msb << 12 )
    2dup adc_P 2! ( full 20 bits unsigned adc_P )
\   ." adc_P = " 2dup d. cr
    5 drshift drop to P15 ( 15 bits left )
;

: READ-WORD ( -- n/2) \ Read unsigned short /2
    BYTE-IN ACK-BIT   \ low byte
    BYTE-IN ACK-BIT   \ high byte
    7 LSHIFT SWAP 1 rshift +
;

: READ-INT ( -- n)    \ Read signed short
    BYTE-IN ACK-BIT   \ low byte
    BYTE-IN ACK-BIT   \ high byte
    8 LSHIFT +
;

: READ-PCOMPS ( -- )
  \ all pressure compensation values
    WAIT-NOCAL CR
    START-BIT
    BME280 BYTE-OUT      \ write address BME280
    8E BYTE-OUT          \ select register 0x8E
    START-BIT
    BME280 1+ BYTE-OUT   \ switch to read-mode
    READ-WORD DIGP 1 ! \ 8E/8F  >>1
    READ-INT  DIGP 2 ! \ 90/91
    READ-INT  DIGP 3 ! \ 92/93
    READ-INT  DIGP 4 ! \ 94/95
    READ-INT  DIGP 5 ! \ 96/97
    READ-INT  DIGP 6 ! \ 98/99
    READ-INT  DIGP 7 ! \ 9A/9B
    READ-INT  DIGP 8 ! \ 9C/9D
    READ-INT  DIGP 9 ! \ 9E/9F

    NACK-BIT
    STOP-BIT


    ." DIGP1/2 = " DIGP 1 @ 6 .R
    ."   DIGP2 = " DIGP 2 @ 6 .R
    ."   DIGP3 = " DIGP 3 @ 6 .R  CR
    ."   DIGP4 = " DIGP 4 @ 6 .R
    ."   DIGP5 = " DIGP 5 @ 6 .R
    ."   DIGP6 = " DIGP 6 @ 6 .R  CR
    ."   DIGP7 = " DIGP 7 @ 6 .R
    ."   DIGP8 = " DIGP 8 @ 6 .R
    ."   DIGP9 = " DIGP 9 @ 6 .R  CR
;


: READ-TCOMPS ( -- )
  \ all temperature compensation values

    WAIT-NOCAL CR
    START-BIT
    BME280 BYTE-OUT        \ write address BME280
    88 BYTE-OUT            \ select register 0x88
    START-BIT
    BME280 1+ BYTE-OUT     \ switch to read-mode
    READ-WORD \ 88/89  DIGT1
    READ-INT  \ 8A/8B  DIGT2
    READ-INT  \ 8C/8D  DIGT3
    NACK-BIT
    STOP-BIT

    DIGT 3 !  DIGT 2 !  DIGT 1 !

    ." DIGT1/2 = " DIGT 1 @ 6 .R
    ."   DIGT2 = " DIGT 2 @ 6 .R
    ."   DIGT3 = " DIGT 3 @ 6 .R  CR
;

: READ-HCOMPS ( -- )
  \ all humidity compensation values
    WAIT-NOCAL CR
```

```
    START-BIT
    BME280 BYTE-OUT         \ write address BME280
    HX A1 BYTE-OUT          \ select register 0xA1
    START-BIT
    BME280 1+ BYTE-OUT      \ switch to read-mode
    BYTE-IN DIGH 1 !        \ A1  8-bits
    NACK-BIT
    STOP-BIT

    START-BIT
    BME280 BYTE-OUT         \ write address BME280
    E1 BYTE-OUT             \ select register 0xE1
    START-BIT
    BME280 1+ BYTE-OUT      \ switch to read-mo
    READ-INT DIGH 2 !       \ signed short
    BYTE-IN ACK-BIT DIGH 3 ! \ E3  8-bits unsigned
    BYTE-IN ACK-BIT DIGH 4 ! \ E4  8-bits [11:4]
    BYTE-IN ACK-BIT TO TMP   \ E5  H5[3:0] H4[3:0]
    BYTE-IN DIGH 5 !         \ E6  8-bits H5[11:4]
    BYTE-IN DIGH 6 !        \ E7  8 bits H6 signed
    NACK-BIT
    STOP-BIT
    DIGH 4 @   4 lshift TMP 0F AND OR DIGH 4 !
    DIGH 4 @ 800 AND
      IF DIGH 4 @ 1000 - DIGH 4 !
      THEN  \ signed 12 bit
    DIGH 5 @   4 lshift
      TMP 4 rshift hx 0F AND OR DIGH 5 !
    DIGH 5 @ 800 AND
      IF DIGH 5 @ 1000 - DIGH 5 !
      THEN  \ signed 12 bit
    DIGH 6 @  80 AND
      IF DIGH 6 @  100 - DIGH 6 !
      THEN  \ signed 8 bit

    ."   DIGH1 = " DIGH 1 @ 6 .R
    ."   DIGH2 = " DIGH 2 @ 6 .R
    ."   DIGH3 = " DIGH 3 @ 6 .R CR
    ."   DIGH4 = " DIGH 4 @ 6 .R
    ."   DIGH5 = " DIGH 5 @ 6 .R
    ."   DIGH6 = " DIGH 6 @ 6 .R CR
;


decimal


: COMP-T ( -- ) \ calculate 10 * T-Celcius
\ with the formula of pdf :
\ T15 = adc_T >> 5
\ dig_T1 is divided by 2
\ TCEL = 10X temperature in Celsius
  T15 DIGT 1 @ - DIGT 2 @ 5120 scale
    TO VAR1  \ now divided by 10
  T15 DIGT 1 @ - DUP 1024 ( >>10 ) scale
  DIGT 3 @ 16384 scale ( >> 14) 10 /
    TO VAR2 \ now divided by 10
  VAR1 VAR2 +
    TO T_FINE  \ T_FINE now divided by 10
  T_FINE 5 128 scale 1+ 2/
    TO TCEL
;


decimal
: .temp  ( -- )
\ T15 dup u. .  CR
   COMP-T
   TCEL
   \ calc. T_fine,
   \ TCEL 10 x temperature in Celsius
   ." Temperature: "
   s>d tuck dabs <# #  ch . hold #s rot sign #>
   \ string_adr length
   type ."  C" cr
;


: COMP-P
  \ calculate airpressure formula for BME280
  \ here tcel is temp_in-Celsius *10,
  \ but needed X 100
  \ only digp1 value is /2
   tcel 10 * 256 m* -103. d+ ( -128+25 )
   50 m/ dup t_finex !
\  ." t_finex = " t_finex @ . cr
\  ." adc_P = " adc_P 2@ d. cr
   ( t_finex @ )  5 m* ( /10 --> /2 )  64000. d-
   2dup vx1 2!
\  ." vx1 = " vx1 2@ d. cr
   4 M/ dup M* 1024. d+ DIGP 6 @ 2048 d*/
   2dup vx2 2!
\  ." vx2 = " vx2 2@ d. cr
   vx1 2@ digp 5 @ 2* 1 D*/ d+
   2dup vx2 2!
\  ." vx2 = " vx2 2@ d. cr
   d2/ d2/  DIGP 4 @ 8192 ( 2^13 ) m*  3 dlshift
   d+ vx2 2!
\  ." vx2 = " vx2 2@ d. cr
   vx1 2@ 4 M/ dup M* DIGP 3 @ 16384 d*/
   d2/ d2/ vx3 2!
\  ." vx3 = " vx3 2@ d. cr
   vx1 2@ DIGP 2 @ 2 d*/ vx3 2@ d+ 18 darshift
   2dup vx1 2!
\  ." vx1 = " vx1 2@ d. cr
   32768. d+ DIGP 1 @ 16384 d*/ vx1 2!
\   ." vx1 = " vx1 2@ d. cr
   vx1 2@ or 0=
     if exit then \ hi and lo or-ed should be <> 0
   1048576. adc_P 2@ d-
   vx2 2@ 12 darshift d- ( lo hi )
   3125 2 d*/ p100 2!  \ actually P100/2
\  ." P100/2 = " p100 2@ d. cr
   vx1 2@ d2/ d2/ ( corrrectie P100 )
   vx3 2!
   p100 2@ vx3 2@
   1 darshift d+ ( denominator/2 added )
   vx3 2@  d>s dup >r abs
   1 swap d*/
     r> ?dnegate
     2dup p100 2! ( just in case denom vx1<0 )
\  ." P100 = " p100 2@ d. cr
   8 ( >>3 ) M/ dup M* 4096. D+ 13 darshift
   DIGP 9 @ 4096 ( >>12 ) d*/ vx1 2!
\  ." vx1 = " vx1 2@ d. cr
   p100 2@ d2/ d2/ DIGP 8 @ 8192 ( >>13 ) d*/
   2dup vx2 2!
\  ." vx2 = " vx2 2@ d. cr
   vx1 2@ D+ DIGP 7 @ s>d d+
   4 darshift p100 2@ d+ p100 2!
\  ." P100 = " p100 2@ d. cr
\   ." luchtdruk = "
\   p100 2@ <# # # ch . hold #s #>
\   type ."  hPa " cr
; \ COMP-P


: COMP-H ( -- )
  \ calculate relative humidity in promille,
  \ approximation
   H15 2 M* adc_H 2!
\   ." adc_H = " adc_H 2@ d. cr
   t_finex @ 10 m* 76800. d- var_h 2!
\   ." t_finex = " t_finex @ . cr
\   ." var_h a = " var_h 2@ d. cr

   adc_H 2@  digh 4 @ 64 m* d-
   var_h 2@  digh 5 @ 1 d*/ 14 darshift d-
   1. d+ d2/ vx1 2!
\   ." vx1 b = " vx1 2@ d. cr

   var_h 2@ digh 6 @ 1024 d*/ vx3 2!
\   ." vx3 d = " vx3 2@ d. cr
```

```
561
562     var_h 2@ digh 3 @ 2048 d*/ vx3 2@ d>s 1 d*/
563         32768. d+ 10 darshift 2097152. d+
564     2dup vx4 2!
565   \   ." vx4 e = " vx4 2@ d. cr
566
567     7 darshift digh 2 @ 1 d*/ 64. d+ 7 darshift
568     2dup vx2 2!
569   \   ." vx2 f = " vx2 2@ d. cr
570
571     vx1 2@  d>s 16384 d*/ d2/
572     2dup vx1 2!
573   \   ." vx1/2^15 g = " vx1 2@ d. cr
574      \ 15 bits shifted right
575
576     3 darshift d>s dup m* d2/ digh 1 @ 16 d*/
577     vx2 2!
578   \  ." vx2 h = " vx2 2@ d. cr
579
580     vx1 2@ 15 dlshift vx2 2@ d-
581     2dup vx3 2!
582   \   ." vx3 i = " vx3 2@ d. cr
583
584     15 darshift  d>s  25 32 scale to hum_pm
585     \ 15 >> ipv 22 >>
586     \ ( oude return was in %, dus *100 /128 )
587   \   ." HUM_PM *10 = " HUM_PM  . cr
588     HUM_PM 100 < if 100 to HUM_PM then    \ >= 1%
589     HUM_PM 9900 > if 9900 to HUM_PM then  \ <= 99%
590     hum_pm 10 / to hum_pm ( promille )
591
592   ; \ COMP-H
593
594   decimal
595   : .pres ( -- )
596     [’] COMP-P catch dup ?error
597       ?exit \ i.p.v. COMP-P
598     ." Pressure:    "
599     p100 2@
600     110. d- ( onnauwkeurigheid, afronding d*/ ?  )
601     Hsl 12 M* d+ ( hoogte in m boven zee )
602     10 M/  s>d
603     tuck dabs
604       <# # ch . hold #s rot sign #>
605       \ string_adr length
606     TYPE ."  hPa "
607   (*
608    p100 2@ 100 m/
609    dup  1033 >                 IF ." very dry"  THEN
610    dup  1022 > over 1034 < AND IF ." sunny"     THEN
611    dup  1006 > over 1023 < AND IF ." variable"  THEN
612    dup   987 > over 1007 < AND IF ." wind/rain" THEN
613          988 <                 IF ." storm"     THEN
614   *)
615     CR
616   ;
617
618
619   : .humi ( -- )
620
621     [’] COMP-H catch dup ?error
622       ?exit \ i.p.v COMP-H
623     HUM_PM 5 + 10 / ( round to % )
624     ." Humidity:    " . ch % emit cr
625   ;
626
627   : wait ( #min -- )
628     60 * begin
629            1000 ms
630            1 -
631            dup 0= key? or
632          until
633          drop
634   ;
635
636
637   : test ( -- )
638      uread \ ROM values ==> to RAM
639      decimal cr
640      ." Height in m above sealevel (Hsl) = "
641      Hsl . CR
642
643      SETUP-I2C
644
645      SETUP-BME280
646      READ-PCOMPS
647      READ-TCOMPS
648      READ-HCOMPS
649      begin
650         read-all
651
652         ." T15:" T15 . ." adc_P:" adc_P 2@ d.
653         ." H15:" H15 . CR
654         .temp
655         .pres
656         .humi  cr
657         5 wait
658         key?
659      until
660      key drop .s
661   ;
662
663   : test1 ( -- ) \ test reading the sensor-ID
664     decimal
665     SETUP-I2C
666     SETUP-BME280
667     10 0 do BME280-id space dm 500 ms loop cr
668   ;
669
670   CR
671   CHERE ’ tools\ - u.  \ bytes flash used :
672   IVECS CHERE - u.     \ bytes flash free :
673   HERE HX 200 - .      \ bytes ram used :
674   TIB HERE - .         \ bytes ram free :
675
676   shield BME280\
677   ’ BME280\ ’ usave cell+ rom!
678   \ specify if needed defaultvalues
679   \ for these uservalues:
680   2   to Hsl \ 2 m above sea-level
681   53  to Node-ID
682   1   to Child-ID
683   USAVE
684   \ --- end of program
685
686
```

Find Baud — oder, wenn man nicht rechnen kann!

Martin Bitter

Während des Treffens der Rhein–Ruhr–Forthgruppe gab es einen Workschop „Einführung in RISC–V mit Mecrisp–Quintus und Longan Nano“. Bei einem Teilnehmer, Carsten, wollte die Kommunikation zwischen seinem Laptop mit einem speziellen USB–Seriell–Wandler und dem Longan nicht gelingen. Andere Wandler an diesem Laptop und andere Laptops mit diesem Wandler funktionierten hingegen einwandfrei. Die Symptome zeigten sich in einer nur halbwegs stabilen Kommunikation. Die meisten Zeichen waren lesbar, jedoch in unregelmäßigen Abständen, alle 5 bis 10 Zeichen, tauchten unsinnige Zeichen auf dem Terminal auf, bzw. empfing der Mikroprozessor Unsinn. So war ein Arbeiten nicht möglich. Ein Datenblatt war nicht zur Hand, nur die Adresse des Baudratenregisters war bekannt.

Baudratenmodul und Bitratenregister

Viele MCUs bieten die Gelegenheit, zur seriellen Kommunikation ein Hardwaremodul zu benutzen. In der Regel ist das bequemer beim Programmieren und schneller in der Ausführung, bzw. lässt mehr Zeit für anderes im Parallelbetrieb, als eine Softwarelösung (Bitbanging). Um dann die Baudrate festzulegen, wird in ein bestimmtes Register, manchmal sind es zwei, ein Wert geschrieben, der festlegt, wie abhängig von einer Quellfrequenz die gewünschte Baudrate erreicht wird. Welcher Wert das jeweils ist, kann dem Datenblatt entnommen werden. Einige Datenblätter zeigen Tabellen, aus denen man sich die richtige Kombination von MCU– bzw. Busfrequenz und Baudrate heraussuchen kann, andere geben eine Formel an, mit der man diesen Wert errechnen soll. Nun kann es aus verschiedenen Gründen sein, dass man diese Berechnung nicht durchführen kann oder die Tabelle nicht zur Hand hat — wie bei unserem Treffen eben auch.

Eine Abhilfe

Was tun? Nun, wenn man nicht rechnen kann, weil wichtige Daten fehlen oder weil die „Normdaten“ nicht helfen (Abweichungen in den Toleranzen?), kann man ausprobieren! Ich habe in einer Schleife das Baudregister mit verschiedenen Werten beschrieben und dann die MCU den String „Klappt!“ und den Wert ausgeben lassen. Vorher habe ich den vorhandenen Wert des Baudratenregisters gerettet und im Anschluss wieder restauriert. Nach dem Start des Wortes `find-baud` erschien wie erwartet Schrott auf dem Bildschirm, aber auch knapp 40 mal der lesbare Text „Klappt!“, gefolgt von den Zahlen `392` bis `438`. Den Mittelwert habe ich dann in das Baudratenregister geschrieben. Das Ganze hat ca. 15 Minuten gedauert, einschließlich an- und umstöpseln von Wandler, MCU und Laptops. Dumm nur, dass das Problem 20 Minuten auftauchte, bevor Carsten den Workshop verlassen musste. So konnte `find-baud` zwar auf meinem Laptop seine Funktionalität beweisen, ob Carsten damit geholfen wurde, weiß ich heute noch nicht. Michael Kalus war ebenfalls zugegen und wollte die Geschichte unbedingt noch in der VD haben (Deadline heute!)[1].

[1] Das war schnell, Martin! Bist noch drin im Heft, wie du siehst. :)

Generalisierbar? Caveat!

Ich denke, der Ansatz ist generalisierbar. Er hilft bei unsauberen Quarzen etc. Das beigelegte Beispiel ist für Mecrisp–Quintus und einen *GD32VF103xx* geschrieben. Wichtig ist, dass man (hier) das Baudratenregister nicht mit Null füllt — das mag der GD32VF103xx nicht! Eine Einschränkung liegt darin, dass man irgendwie eine funktionierende Verbindung braucht (anderer Laptop). Dort kann man dann `find-baud` starten und muss nun dafür sorgen, dass beim Anschließen an den Ziellaptop die MCU nicht stirbt. Ist das gelungen, startet jeder Tastendruck (vgl. `key`) die Finderoutine. Nebenbei ist es der Routine egal, mit welcher Baudrate das Terminal angeschlossen ist, sie findet für jede technisch mögliche Baudrate den richtigen Wert. Zudem kann man überprüfen, wie die offiziellen Werte (aus Formel oder Tabelle) zu den tatsächlichen Gegebenheiten passen. Falls man die Gegenseite (Laptop) so programmiert, dass sie aktiv ein Codewort sendet, kann man die Antworten des Laptops analysieren und so das Eintragen des richtigen Wertes in das Baudratenregister automatisieren. Es kann vorkommen, dass so fürchterlich ungültige Werte in das Baudratenregister geschrieben werden, dass die Kommunikation einfriert. Bei mir ist das ab einer MCU–Frequenz von 104 MHz der Fall gewesen.

```
: find-baud ( -- )
  $40013808 @ >r \ Baudrate retten
  key drop \ auf Tastendruck warten
  ." Originaler Baudratenwert ist: "
  dup . cr \ Baudratenwert ausgeben
  1000 0 do \ und 1000 mal
  $10 I + $40013808 ! \ I > 10 wg fraction
  cr ." Klappt! " 10 I + . \ Meldung
  loop
  r> $40013808 ! \ Baudrate restaurieren
;
```

Carsten: Gültige Werte gefunden im Bereich von 411 bis 438.

Martin: Gültige Werte gefunden im Bereich von 392 bis 438.

```
: init ( -- )   411 438 + 2/ $40013808 ! ;
```

Forth–Gruppen regional

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

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

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

Ruhrgebiet **Carsten Strotmann**
`ruhrpott-forth@strotmann.de`
Treffen alle 1–2 Monate im Unperfekthaus Essen
`http://unperfekthaus.de`.
Termine unter: `http://forth-ev.de`

Gruppengründungen, Kontakte

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

μP–Controller Verleih

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

Spezielle Fachgebiete

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

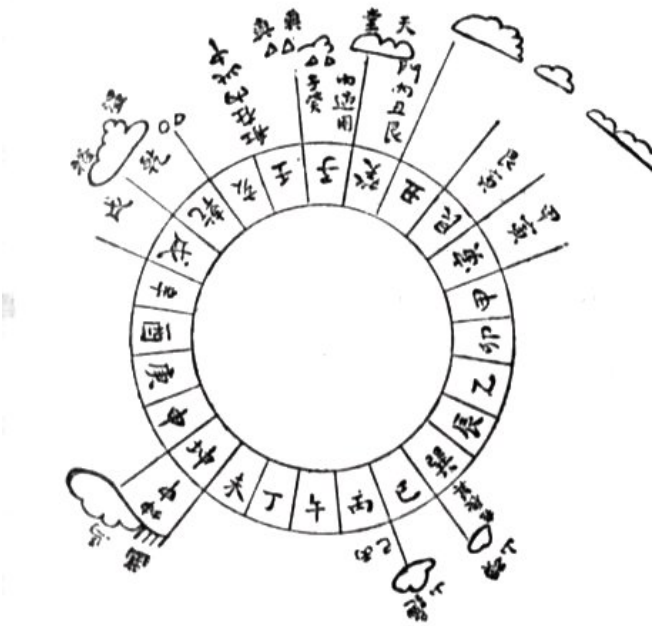
Termine

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

26.–29.03.2020 +++ abgesagt +++
Forth–Tagung in Klein–Glien
`https://tagung.forth-ev.de`

12.–13.09.2020
Maker Faire Hannover, HCC
`http://www.makerfairehannover.com`

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



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

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

Offizielle Absage der Forth–Tagung wegen Coronavirus

Bernd Paysan

Die Tagung im Jahr 2020 sollte uns vom 26. bis 29. März 2020 in das CocoNAT Workation Reatreat *in Klein–Glien bei Bad Belzig führen. Leider hat sich das neuartige Corona–Virus SARS–CoV–19 auch in Deutschland breitgemacht, weshalb wir, das Direktorium der Forth–Gesellschaft, nicht in der Lage sind, die Tagung wie geplant durchzuführen.*

Seit Anfang März steigen die Fallzahlen von COVID–19, wie die WHO die vom Erreger SARS–CoV–19 ausgelöste Krankheit nennt, auch in Deutschland rasant exponentiell an — exponentiell ist das, wofür die wenigsten Menschen ein intuitives Gefühl haben. Wir denken linear und extrapolieren von der Vergangheit linear weiter. „3 oder 4 Tote bisher, das ist doch nix." (jetzt sind es schon 16). Italien ist dabei derzeit ziemlich genau 9 Tage voraus (Stand ca. 7000 gemeldete Infektionen). Alle Faktoren, die in Italien vielleicht anders sind als bei uns, sind vor dem Exponenten (mehr/weniger testen, Verteilung der Infektion über die Altersgruppen, vorhandene Intensiv–Betten bis zum Zusammenbruch der Aufnahmekapazitäten), bedeuten also lediglich eine Verschiebung um wenige Tage nach vorn oder hinten.

Entsprechend lief in den vergangenen Tagen eine ansteckende[1] Welle Aktionismus durch die bislang bräsig abwartende Politik, und auch die rechtliche Situation hat sich, wie vor Tagen schon vermutet, erheblich geändert. Ausgangssperren wie in Frankreich oder Italien haben wir hier aber noch nicht.

Das Virus ist bekämpfbar, aber nur mit harten Maßnahmen — nicht über konstante Faktoren, sondern indem man den multiplikativen Faktor von >1 auf <1 (und zwar möglichst deutlich kleiner — je kleiner, desto kürzer die harte Maßnahme) drückt. Alle Maßnahmen werden mit einer Verzögerung von 10 Tagen oder mehr wirken, weil die gemeldeten Zahlen den tatsächlichen Ansteckungen etwa um 10 Tage hinterherhinken. Es ist also jetzt schon klar, dass die Fallzahlen von Italien (Stand heute) erreicht werden, egal, welche Maßnahmen jetzt ergriffen werden. Es ist schon jetzt klar, dass Italien die Zahlen in Wuhan übertreffen wird, und zwar erheblich.

Einige Maßnahmen wirken nur, solange die Fallzahlen klein sind, weil z. B. das Testen und Verfolgen aller Kontakte einfach ab einer gewissen Fallzahl nicht mehr möglich ist. Und einige Maßnahmen sind erst durchsetzbar, wenn die Fallzahlen groß genug sind, damit alle Skeptiker überzeugt werden. Obwohl die Maßnahmen dann länger dauern und härter sein müssen. Der Beschluss von Bundesregierung und Ländern vom 16. März untersagt deutschlandweit schon sehr viele Arten von Veranstaltungen.

Was bedeutet das konkret?

Es gibt im Moment erhebliche Einschränkungen der Versammlungsfreiheit. Unsere Veranstaltung ist nicht direkt wörtlich erwähnt, aber das ist alles mit „oder ähnliches" ergänzt. Übernachtungen dürfen nur noch zu „notwendigen" Zwecken genutzt werden, und definitiv nicht zu touristischen Zwecken. Unsere Tagung ist ganz sicher nicht systemrelevant, und damit im Moment nicht nur verzichtbar, sondern damit eben auch nicht zulässig. Selbstverständlich versuchen wir, Stornogebühren zu vermeiden, um das Vereinsvermögen nicht zu belasten.

Wir überlegen daher, ob und wie wir die Tagung im März als „virtuelle Forth–Tagung" abhalten können, bei der wir uns per Video–Konferenz zusammenschalten. Außerdem sind wir derzeit auch mit dem Tagungshotel CocoNAT über eine mögliche Verschiebung der Tagung etwa in den (Spät)–Sommer in Verhandlungen.

Bitte prüft also, unter welchen Bedingungen Ihr Eure jetzigen Reisetickets stornieren könnt. Bahntickets können bei offiziellen Absagen aufgrund COVID–19 nach Informationen der Deutschen Bahn kostenfrei storniert werden. Verschiedene Grenzen ins Ausland sind derzeit weitgehend geschlossen, die abgesagten Flüge oder Bahnfahrten von dort werden wohl auch erstattet.

Bleibt gesund, oder übersteht, falls es nicht klappt, diese Infektion unbeschadet.

[1] Auch das Verhalten in Epidemien lässt sich epidemiologisch erklären. Die Information, dass man Klopapier horten muss, hat z. B. ihren Ursprung in Hong Kong, das selbst keine Klopapierproduktion hat, weshalb die Leute dort einen Versorgungsengpass befürchteten. Und von dort aus wurde der Rest der Welt damit angesteckt. Das ist ein Virus für das Gehirn. Unsere OP–Masken kommen aus China, aber unser Klopapier, das machen wir selber.