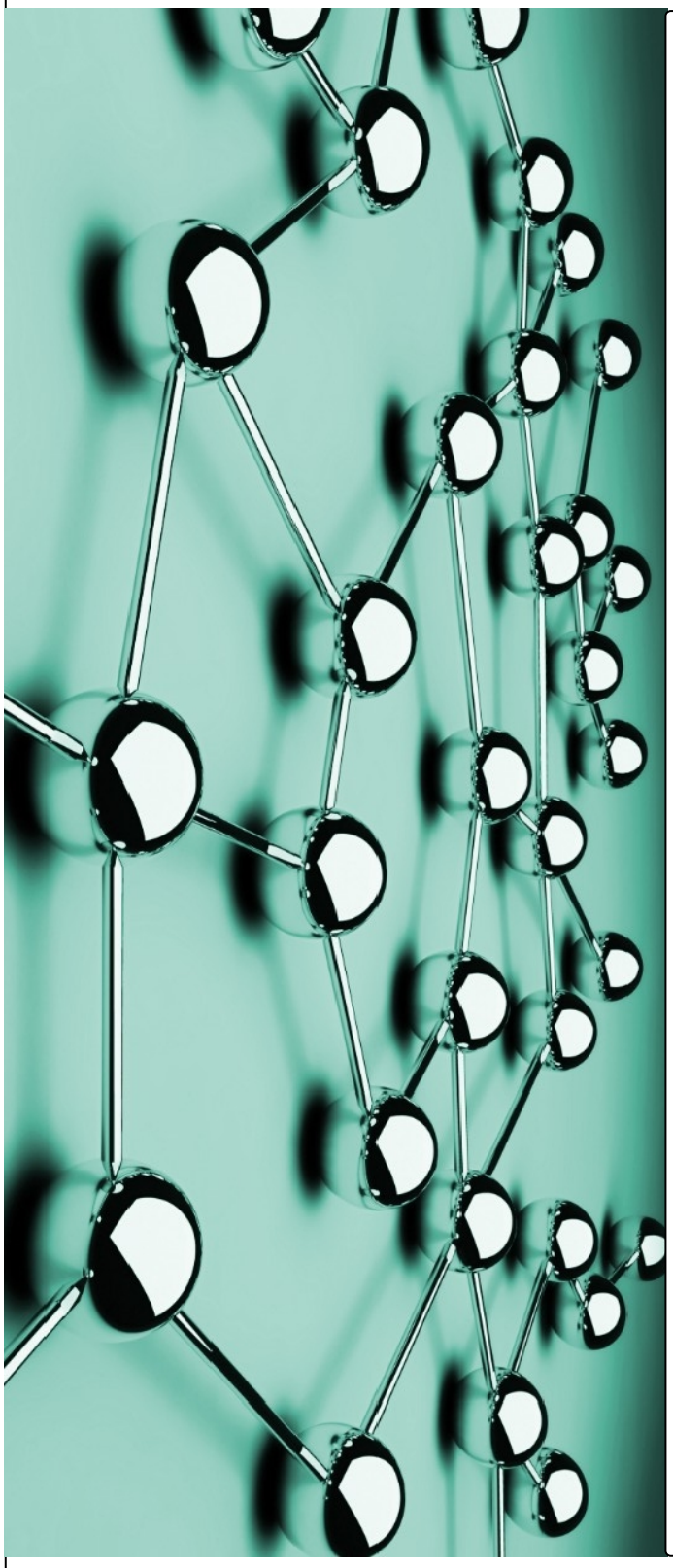
Vierte Dimension

1/2016

Das Forth–Magazin

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

In dieser Ausgabe:



Dienstleistungen und Produkte fördernder Mitglieder des Vereins

tematik GmbH
Technische
Informatik

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

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

LEGO RCX–Verleih

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

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

Inhalt:

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

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

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

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

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

Ingenieurbüro
Klaus Kohl–Schöpe

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

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

KIMA Echtzeitsysteme GmbH

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

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

FORTecH Software GmbH
Entwicklungsbüro Dr.–Ing. Egmont Woitzel

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

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

Liebe Leser,

Größere Netze sind mir erst bei der Seefahrt so richtig begegnet. Mit dem Kutter auf der Nordsee. Denen, die noch aus Holz waren, gebaut von Hinni Bülcher in Ditzum. Der Geruch von Modder, Tang und gekochten Krabben war in der Luft, und auf dem Grill fangfrische Scholle. Hm, lecker, sag ich euch. Die Netze hingen im Hafen, teils gebündelt, teils aufgespannt. Wunderbar geknüpft. Wie man solche Netze macht war mir ein Rätsel. Zu der Zeit wurden auch Geschichten geknüpft, die waren aus Seemansgarn, und handelten von Ebbe und Flut und dem Klabautermann.

Später hatte ich mit sozialen Netzen zu tun. Darin kam Forth schon vor. Klaus Schleisiek kam gerade aus den USA zurück nach Hamburg, also zurück zu den Netzen. Er hatte eine „Forth Interest Group, local chapter“ mitgebracht. In Wuppertal hatten wir auch eine kleine Gruppe Forther und wollten auch ein FIG–Chapter werden. Damals hatte ich noch die Angewohnheit, Leute persönlich aufzusuchen. In Kontakt kam man noch per Brief. Und so traf man sich bei Horst–Günter Lynche und seiner ersten „Vierte Dimension“. Später wurde die Forth–Gesellschaft daraus.

Heute werden Netze elektromagnetisch und digital geknüpft. Wunderbar, was so ein Smartphone damit aus der Luft fischen kann. Aus Italien sogar, von der Universität Palermo, erreichte mich die Kunde, dass auch dort solche Netze kunstvoll geknüpft werden, für Sensoren, mit AmForth. Das sind Zauberer.

Ein anderer Magier ist sogar grad dabei, das Internet selbst anders zu knüpfen. Im Chat mit net2o könnt ihr es bereits selbst erproben. Dem Vernehmen nach läuft alles schon stabil.

Selbst altes Forth kann wieder herausgefischt und in neue Program–Netze gewoben werden mit einer kunstvollen Technik, die Systemvariablen wieder benutzbar macht. Diese variablenartigen defining words sind neu für mich.

Dass neben humaner vernetzter Interaktion auch die Maschinen dazu übergehen, das Internet der Dinge, das ahnt man ja. Doch der Teufel steckt bekantlich im Detail. Solch ein Netz im Allgemeinen will kontrolliert sein und der Wechselstrom, es zu betreiben, im Speziellen auch. Optokoppler sind die Augen dafür.

Vernetzt, wie ihr heute alle seid, habt ihr es inzwischen vermutlich längst mitbekommen, dass die kommende Jahrestagung unserer Forthgesellschaft Mitte April in der Hochschule Augsburg im Fachbereich Informatik stattfindet. Ich hoffe, ihr habt alle eine Unterkunft ergattert und es wird lauschig in dem hübschen Städtchen. Also treffen wir uns schon bald im schönen Augsburg. Bis dann, 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-2016-01`

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

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

go forth

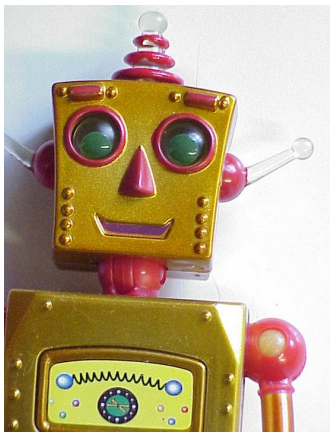
Abbildung 1: Wandmalerei von Alexandre Farto in Berlin

Die *Go–Forth–Kampagne* von Levi's aus dem Jahre 2009 war Auslöser für die Wandmalereien in Berlin. Der portugiesische Straßenmaler Alexandre Farto, aka Vhils, schuf, davon inspiriert, eine Portrait–Serie lokaler „moderner Pioniere des Alltags". Das wiederum war Vorbild für das Titelblatt unseres Forth–Magazins „Vierte Dimension", Heft 3+4/2015. Chuck Moores Gesicht wurde dafür auf eine Häuserwand in Dortmund gemalt. Michael Ostermann heißt der Künstler, der das Bild geschaffen hat — mit Photoshop. (Foto: David Yates, Creative Commons, `http://andberlin.com/2012/03/28/vhils-go-forth/`)

Kings Go Forth (Delmer Daves) war ein Schwarz–Weiß–Film aus dem Jahre 1958. Ich war da 8 Jahre alt. Hat etwas gedauert, mir den Film heute mal anzusehen. Wohin Forth einen aber auch so führen kann ...

> After having now seen it a couple of times through showings on the UK TCM, I have found it growing on me and especially like Sinatra's delicate, understated central performance. Tony Curtis and Natalie Wood both do their best with difficult roles, and the sweeping black–and–white views of the French Riviera are memorable, as is the melancholy Elmer Bernstein score.

(`https://movieclassics.wordpress.com/2014/01/17/kings-go-forth-delmer-daves-1958/`)



Last but not least ist *GoForth* der Titel eines Baukastens der Firma ST Robotics. Die Hardware ist lose um Meccano und andere Metallbauteile herum konstruiert, mit einem ARM–Microcontroller für die Motorsteuerung und Forth als steuernde Sprache. Der Baukasten kann von Schulen günstig bezogen werden von deren *Schools Robotics Initiative*. `www.strobotics.com` mk

:–)

```
: Forth ( -- )
  is the only language
  that is usefull for digestion
  ( because it's the only one   )
  ( that have colon definitions ) ;
```

Bernard Geyer and Stephen Walters, in FORTH PROGRAMMING LANGUAGE, Facebook; 07.01.2016 db

De Saint–Exupery's Law of Design

A designer knows that he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away. (`http://spacecraft.ssl.umd.edu/akins_laws.html`) mk

Forth — not quite dead

In seinem lesenswerten Beitrag rudert Michael Will (scidata) zurück: „So, while I do not aspire to revive Forth or advocate it generally, I may have been in error in describing it as dead. I come to praise Forth, not to bury it." Vor dieser Schlussfolgerung fasste er in 12 Punkten zusammen, warum Forth eine gute Wahl für die (Programmier) Ausbildung ist. Und gibt auch Quellen an: „There are several efforts at delivering basic, classic Forth to beginning programmers. One that I particularly like is *4E4th* which runs on a tiny, inexpensive prototyping card. The Raspberry Pi can easily run a bigger, more capable Forth. Several commercial enterprises continue to promote Forth. Some can be found towards the end of the excellent and extensive Forth page on Wikipedia." Und wie wir wissen, bieten MPE (`http://www.mpeforth.com/`) und Forth inc. (`http://www.forth.com/`) exellente Forth–Systeme an. mk

Der Swap

Der ist froh, bei Messi noch einen Platz im Regal ergattert zu haben, und freut sich schon auf die nächste Forth–Tagung der Forth–Gesellschaft. Außerdem schaut er kritisch, wie unsereiner immer weiter in die „arbeitslose“ Faulheit abdriftet. Mehrere Projekte wurden angefangen und harren der weiteren Bearbeitung. Der Wert für Projekteanfangen/sec ist sehr hoch. Zur Zeit versuch ich einen Einstieg in die FPGA–Welt zu finden. Ein Gameduino wurde schon angeschafft, Quartus II (für den Gameduino unbrauchbar) und ISE installiert. Angeregt durch den Artikel in der letzen 4d, ist N.I.G.E. nun in der neueren Version auch schon angeschafft. Leider hat der Entwickler keinerlei Abwärtskompatibilitäten eingebaut, sodass die Benamsung komplett überarbeitet werden muss, um N.I.G.E.–konform zu kompilieren. Bis zur Aufstellung der Unterschiede zum Vorgänger–Board bin ich schon gekommen. Das Ashling–Equipment möchte auch noch ein Forth auf den Tricore von Infineon bringen — das scheint noch die größte Hürde zu sein. Aber ihr seht, es geht ihm gut, dem Swap.

Worauf der SWAP von seinem Platz aus sonst noch so blicken konnte, die Artefakte, die belegen, wie ich zu Forth kam, das ist dann eine eigene Geschichte. Karsten

SWAP und seine Freunde

SWAP hat Freunde in aller Welt, z. B. dieses brückenbewachende Geschwisterpaar hier:

Frage: **Wo befinden sich diese beiden Swap–Freunde?**

Antworten bitte an die Redaktion unter `vd@forth-ev.de`.

Die Antwortgeber der richtigen Antworten werden im kommenden Forth–Magazin veröffentlicht. uho

Variably — die Variablen–artige

Ulrich Hoffmann

Bei der Portierung eines umfangreichen Forth–Pakets war es die Aufgabe, den Original–Quellcode so wenig wie möglich zu verändern. Dieser verwendete System–Variablen die Forth–typisch mit `@` *ausgelesen und mit* `!` *geschrieben wurden. Der vorliegende Artikel beschreibt, wie dieses Verhalten nachgebildet werden kann, dabei aber für* `@` *und* `!` *geeignete Worte zum Lesen und Schreiben aufgerufen werden.*

Motivation und Grundlagen

In altem Forth–Code finden wir immer wieder system-spezifische Variablen, die in der Form vvv `@` oder vvv `!` benutzt werden, um das Verhalten des Programms oder des zugrundeliegenden Forth–Systems zu beeinflussen. `STATE`, `BASE`, `CURRENT`, `CONTEXT`, `DP`, `>IN`, `SCR`, `BLK`, `DPL`, ... sind solche Variablen. Moderne Systeme müssen dabei möglicherweise viel mehr Aspekte als das einfache Auslesen und Setzen einer Variablen durchführen und verwenden daher explizite Worte, um entsprechende Aktion auszuführen.

Beispielweise reichte es traditionell aus, den Wert von `STATE` zu ändern, um den Zustand des Forth–Outer–Interpreters zu beeinflussen:

- `-1 STATE !` schaltete in den Kompilations–Modus.
- `0 STATE !` schaltete in den Interpretations–Modus.

Dieses Vorgehen ist Standardprogrammen (Forth–94 oder Forth–2012) nicht erlaubt. Sie dürfen `STATE` nur noch lesen. Den Modus des Outer–Interpreters dürfen sie nur durch die Worte `[` und `]` setzen:

- `]` schaltet in den Kompilations–Modus.
- `[` schaltet in den Interpretations–Modus.

Ziel ist es nun, alten Forth–Code soweit wie möglich unverändert zu lassen, und dabei den Phrasen vvv `@` und vvv `!` eine neue Bedeutung geben zu können: nämlich den Aufruf eines beliebigen Wortes (natürlich mit passendem Stack–Effekt).

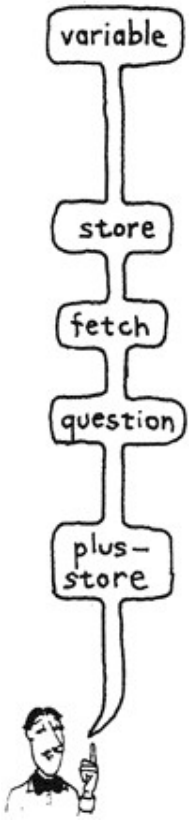
Implementierung

Wir wollen also Worte definieren können, die sich im Wesentlichen so wie Variablen verhalten. Wenn richtige Variablen alleine stehen, legen sie ihre Adresse auf den Datenstack, mit dem `@` und `!` dann arbeiten können. Unsere neuen variablen–artigen Worte, nennen wir sie *Variablys* (kleine Variablen oder Variablen–artige), sollen nicht alleine vorkommen dürfen, sondern eben nur im Zusammenhang mit `@` und `!`, was häufig gegeben ist:

- vvv `@` (-- x) liest eine Variably aus.
- vvv `!` (x --) schreibt eine Variable.

Um Variablys zu erzeugen, dient das Defining–Word `Variably`. Siehe Listing 1 auf der nächsten Seite.

`Variably` erwartet zur Definitionszeit des neuen Variablys zwei Execution–Tokens auf dem Stack: Zunächst das Execution–Token für die `@`–Aktion mit dem Stack–Effekt `( -- x )` und dann das Execution–Token für die `!`–Aktion mit dem Stack–Effekt `( x -- )`. `Variably` speichert sie im Parameterfeld (Body) der neu definierten Variably. Variablys selbst sind Compiling–Words und damit immediate.



Wird die Variably später in der Form vvv `@` benutzt, wird die `@`–Aktion ausgeführt, bei vvv `!` die `!`–Aktion. Beide können tun, was immer nötig ist, um den Wert auf dem Stack zu erzeugen bzw. zu verarbeiten. Um die `@`– bzw. `!`–Aktion zu ermitteln, liest das Variably das nächste Wort aus dem Eingabestrom. Ist es `@` wird die `@`–Aktion ausgeführt bzw. kompiliert, ist es `!` die `!`–Aktion, deren Execution–Token sich ja im Parameterfeld der Variably finden. Ansonsten ist die Variably falsch verwendet worden und eine passende Fehlermeldung wird ausgegeben.

Anwendung

Ein Standard–System ist nicht verpflichtet, die klassischen Variablen `CURRENT` (Vokabular, zu dem neue Definitionen hinzugefügt werden) und `CONTEXT` (Vokabular, das durchsucht wird) zu definieren. Stattdessen gibt es

```
1  \ Variably - words that behave in a variable manner.   uh 2015-10-24
2
3  \ vvv @  and  vvv !  call an appropriate word specified
4  \ when defining the variably.
5
6  : Variably ( <name> xt-@ xt-! -- )
7     CREATE , , IMMEDIATE
8     DOES> ( i*x -- j*x )
9       ' DUP [’] @ = IF DROP CELL+ ELSE
10            [’] ! - ABORT" must be used with @ or !" THEN
11      @  STATE @ IF COMPILE, ELSE EXECUTE THEN ;
```

Listing 1: Die Implementierung von `Variably`

die Worte `GET-CURRENT`, `SET-CURRENT` zum Auslesen und Setzen der Definitions–Wortliste bzw. `GET-ORDER` und `SET-ORDER`, um die Suchordnung auszulesen und zu beeinflussen.

Alte Programm lesen und manipulieren aber möglicherweise `CURRENT` und `CONTEXT` direkt.

Definiert man `CURRENT` und `CONTEXT` nun wie folgt:

```
1  \ example: old vocabulary variables
2
3  ’ GET-CURRENT   ’ SET-CURRENT   Variably CURRENT
4
5  :NONAME ( -- wid )
6     GET-ORDER OVER >R SET-ORDER R> ;
7
8  :NONAME ( wid -- )
9     >R GET-ORDER NIP R> SWAP SET-ORDER ;
10
11 Variably CONTEXT
```

dann wird für `CURRENT @` das Wort `GET-CURRENT` und für `CURRENT !` das Wort `SET-CURRENT` ausgeführt bzw. kompiliert. Analog wird für `CONTEXT @` das erste mit `:NONAME` anonym definierte Wort aufgerufen/kompiliert, das die erste Wortliste der Suchreihenfolge liefert, bzw. für `CONTEXT !` die zweite `:NONAME`–Definition zum Setzen der ersten Wortliste der Suchreihenfolge.

Das alte Programm kann also weiter mit `CURRENT` und `CONTEXT` arbeiten, verwendet dabei aber die neuen Features des zugrundeliegenden Systems.

Ausblick

Variablys sind nicht nur ein Werkzeug zum Portieren alter Programme auf neue Systeme. Sie lassen sich auch gut für das Debuggen einsetzen, da sie den schreibenden und lesenden Zugriff auf Variablen zu protokollieren erlauben:

```
1  Variable logme
2  :noname cr ." logme-fetch "    logme @ .s ;
3  :noname cr ." logme-store " .s logme ! ;
4  Variably logme
```

Definiert man zu einer Variablen ein passendes, gleichnamiges Variably, wie `logme` im obigen Quellcode, dann wird jedes `logme @` und `logme !` auf der Konsole protokolliert.

Mögliche Erweiterungen

Variablen werden zuweilen auch zusammen mit den Worten `ON`, `OFF`, `+!` oder `?` (und anderen wie `C@` oder `C!`) verwendet. Eine passende Erweiterung von `Variably` ist problemlos möglich, indem einfach in `Variably` auf die zusätzlichen Worte geprüft wird und zusätzliche Aktionen mit ihren Execution–Tokens in der Variably abgelegt werden. Eine solche Erweiterung zu versuchen, überlassen wir dem geneigten Leser als Fingerübung.

May the Forth be with you.

Bildquelle: Starting-Forth, Leo Brodie, © Forth Inc. http://www.forth.com/starting-forth/sf8/sf8.html

Referenzen

1. http://forth-standard.org/

Use of Forth to Enable Distributed Processing on Wireless Sensor Networks

Salvatore Gaglio, Giuseppe Lo Re, Gloria Martorella and Daniele Peri
DICGIM, University of Palermo - Viale delle Scienze, Ed. 6 - 90128 Palermo, Italy
{salvatore.gaglio, giuseppe.lore, gloria.martorella, daniele.peri}@unipa.it

***Abstract*—Wireless Sensor Networks (WSNs) are composed of tiny sensor nodes able to monitor environmental conditions. Existing applications for WSNs usually adopt a centralized approach that exploit sensor nodes just for sensing, while data processing takes place on more powerful base stations. This can be considered a consequence of the common WSN programming practice that proves too rigid to support development based on distributed processing. In fact, local processing of complex data, such as symbolic information and rules, is an under explored aspect. The adoption of high level interpreters above general purpose operating systems is often unpractical since it implies the saturation of the available resources. In this paper, we detail the implementation of an alternative Forth-based approach that implements a minimal but extensible operating system featuring common WSN functionalities as well as advanced skills such as symbolic distributed processing. We show the definition of words and syntactic constructs that enable collaborative processing on WSNs and ease the development of complex applications even on resource constrained WSN nodes. To this purpose, our approach is based on an abstract mechanism enabling nodes to exchange directly Forth code. Cooperative behaviors, introducing dynamic computation into the network, are thus easily implemented, as we show in a few applicative examples. Moreover, using the same mechanism, remote nodes can be effortlessly reprogrammed even after their deployment. Finally, we show how our approach proves to be feasible and advantageous through a comparison, in terms of memory usage, with relevant interpreter-based software platforms for WSNs.**

I. Introduction

Wireless Sensor Networks (WSNs) are composed of tiny wirelessly interconnected sensor nodes that are equipped with a microcontroller, a radio interface subsystem, some sensor devices and an autonomous power supply, usually consisting in batteries [1]. Generally, such devices are characterized by quite constrained resources in terms of energy, communication and processing capabilities.

WSNs represent a very active research area as several applications have been proposed in literature in several contexts such as biomedical, healthcare, military, industrial and environmental fields [2].

The development of high level applications is typically supported by general purpose operating systems for WSNs such as Contiki and TinyOS [3], which primarily focus on reducing power consumptions while optimizing resource usage [4].

Mainstream programming practices involve the cross-compilation of specialized code with the thin layer operating system of choice, and the subsequent code uploading to the on-board ROM memory. Any modifications in the source code lead to retrace the same steps afresh.

Such practice strongly limits the development of more advanced applications than the static acquisition and transmission of sensory data that is then to be processed by a base station [5].

Sophisticated applications, such as those concerning Ambient Intelligence (AmI) scenarios, could instead be developed if the nodes were able to process cooperatively more complex data –e.g. symbolic data and rules– than the numerical values in rigid representations resulting from sensing. Such applications may in fact implement intelligent, autonomic, and self-organizing behaviors by distributed processing of symbolic and qualitative description of the observed phenomena. However, due to memory constraints of the available development methodologies, such kind of applications are too complex to be implemented on WSNs without recurring to centralized or Cloud-based infrastructures [6].

In order to give the network some adaptivity to changes of the environment as well as of the application goals after the nodes have been deployed, alternative WSN application development tools are thus strongly required [7].

To overcome the inflexibility of conventional programming methodologies, several interpreters targeting resource constrained Wireless Sensor Network (WSN) nodes have been presented in literature [8], [9], [6], [10]. Their primary goal is to support the application development as well as the retasking of already deployed nodes. However, node reprogramming affects just the application code, while the hardware-abstraction layer modifications require to upload the whole binary image or to replace just the modules to be updated [11].

In general, high-level language interpreters are designed as applications running atop the chosen general purpose operating system. Unfortunately, such a strategy dramatically increases the processing load on the on-board microcontrollers, and detaches the application from the hardware. Moreover, this solution often leads to high memory occupation that leaves insufficient memory resources to develop not trivial applications [8].

The choice of Forth in WSN AmI applications, which are characterized by realtime and resource constraints, seems thus quite natural and desirable [12]. Moreover, the interactive nature of Forth makes it easy to face the challenges of AmI development with experimental programming.

In this paper, we detail our experimentation on the use of Forth on WSN nodes as an operating system and development tool. We describe our ongoing implementation of a

Forth-based software platform that provides nodes with basic WSN capabilities such as networking, sensing, and actuation, accessible through expressive words, and easily extensible to support complex functionalities.

Distributed processing is one of the key goals of our platform that we addressed with a simple abstract mechanism based on the transmission of Forth code among nodes, even already deployed ones. In the next sections, we show how we have been able to implement this abstraction in a few dozen words, with a remarkably low resource usage with respect to other available interpreters.

The remainder of the paper is organized as follows. Section II details the wordset we have implemented to use Forth as an operating system for WSNs. In Section III, the primitives supporting distributed processing are presented. Section IV describes some working applications running on WSN nodes in order to demonstrate the feasibility of our approach and finally Section V reports our conclusions.

II. Forth as an Operating System for WSNs

Forth naturally provides an interactive environment with most of the functionalities of an operating system for common computers. In the case of WSN nodes the OS responsibilities include the management of networking as well as all the various on-board and optional sensors and devices.

Most WSN nodes –referred to as *motes* in the specific literature– are based on MCU with Harvard architecture with separate memories for data and programs. Several interfaces, e.g. digital I/O, analog inputs, I2C, SPI and UARTs enable the connection with external modules, such as the radio subsystem, sensing boards, and so forth. For instance, the IrisMote platform that we used as a testbed, which is one of the most adopted, especially for research purposes [13], is equipped with an IEEE 802.15.4 compliant radio transceiver, 128 KB of Flash memory, 8 KB of static RAM and a 4 KB EEPROM, and can be expanded with sensing and prototyping boards.

At the beginning of our experimentation, we sorted out all the available Forth environments targeting the AVR microcontroller used in the IrisMote platform. We chose AmForth [14], a simple indirect threaded code interpreter, as it proved mature enough to be used as a development tool, and as it also provided a usable interactive shell through a serial terminal. However, AmForth did not include natively the support for any WSN platform. This required us to patch AmForth for the IrisMote to include specific configuration settings, such as those concerning ports, clock generators, on-board radio registers, timers and so on.

In our efforts to build an operating system for WSNs we defined an essential collection of definitions for the basic functionalities needed by WSN applications, such as networking and sensing.

Not all the definitions are strictly related to the hardware. Instead, we defined some more generic and platform-independent words that are not tied to specific hardware implementation and can thus easily work on different platforms.

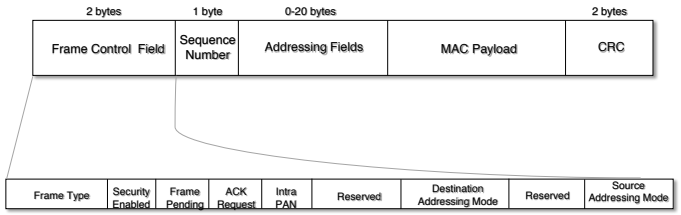


Fig. 1: Format of a valid MAC layer frame according to the 802.15.4 standard. The frame control field is detailed in the bottom of the figure.

As an example, hardware independent words are those used to create valid data frames according to the 802.15.4 standard. In our implementation, transmission and reception of valid 802.15.4 frames is based on two buffers:

- outbound: a memory area where the outgoing frame is stored before downloading it to the radio frame buffer for the transmission;
- inbound: a memory area where the received frame is stored after it is uploaded from the radio frame buffer.

The buffers are 128 bytes long. According to the 802.15.4-2003 standard (see Figure 1), we defined the words to create valid data frames and to set the frame fields appropriately, e.g. short/long destination addressing mode field, frame type, frame length, and so on. In particular, Listing 1 shows the word definition to create a default frame with the following settings:

- short addressing mode (source and destination);
- intra-pan bit set to 1;
- 0xabcd pan address;

Listing 1: Forth word to create a valid 802.15.4 data frame with fixed settings

```
: default-pkt ( -- packet )
outbound dup erase
data frame_type
pan_compr
dest short mode! src short mode!
dest pan $abcd s_addr! src addr id @ s_addr! ;
```

A. Forth Words for Input Redirection to the Radio Module

Our Forth-based implementation supports interactive development on already deployed devices. This feature permits adding new words on remote nodes even if they are not physically connected to a serial terminal. Interactivity, symbolic processing and executable code exchange are the pivotal characteristics of our system.

The code exchanged among nodes and received from the radio channel is interpreted by the system, provided that the default input –the USART, at boot– has been redirected to the radio transceiver. Each incoming frame triggers the interrupt invoking the text interpreter on the frame payload.

Listing 2 shows the word definitions to enable the interpretation of incoming frames from the radio channel, by switching the input from the USART to the radio.

Listing 2: Forth words to redirect the standard input device to the radio

```
variable old-key
variable old-key?
variable payld-addr
variable payld-size
variable payld-in
$200 constant usart_rx_in
$201 constant usart_rx_out
$202 constant usart_rx_data

: payld-reset
    0 payld-size !
    0 payld-in ! ;

: payld-set ( addr n -- )
    payld-size !
    payld-addr !
    0 payld-in ! ;

: radio-key?
    payld-size @ dup 0 > swap
    payld-in @ > and ;

: radio-keyin
    payld-in @ dup payld-addr @ +
    c@ swap 1 + payld-in ! ;

: radio-key
  begin pause radio-key? until radio-keyin ;


: +radio-input
  payld-reset
  ['] key defer@ old-key !
  ['] key? defer@ old-key? !
  ['] radio-key is key
  ['] radio-key? is key? ;

: -radio-input
  old-key @ is key
  old-key? @ is key? ;

: usart_inject
  usart_rx_in c@ usart_rx_data + !
  1 usart_rx_in +! ;
```

Essentially, the input redirection makes the deferred words `key?` and `key` point to `radio-key?` and `radio-key` respectively. The word `radio-key?` is used to assess if there are unread characters in the frame payload by checking either if the variable `payld-size` is greater than 0 and the current pointer to the payload `payld-in` is lower than `payld-size`. The word `radio-keyin` fetches the next character in the frame payload and advances the current payload pointer `payld-in`. Finally, the word `radio-key` executes `radio-key?` and `radio-keyin` until all the characters in the frame payload have been read. To redirect the input to the radio, the word `+radio-input` is typed in the node shell. The execution causes the AmForth shell to be lost, until a data frame containing the word `-radio-input` is received. This word restores the input to the USART.

Code processing takes place directly in the interpreter loop as the last character of each incoming frame is required to be a carriage return. Such an event triggers the interpretation of the payload. However, in real use, interacting with networked devices through a wired line is unpractical. In fact, to redirect the input to the radio system without any wired connection, we defined a special frame containing just the character $17, which is the ASCII code for the non-printable character ETB.

Once a frame is received, the node uploads it from the radio frame buffer and checks whether the frame payload is equal to ETB. If so, it executes `+radio-input`. Actually, to switch the input, the word `usart_inject` must be executed to exit the system blocking loop waiting for characters from the USART that in the current AmForth implementation cannot be preempted in other ways.

B. Support to the Radio Operations

In order to support the communication among nodes, we defined a number of words to drive the radio of IrisMotes. The low-power AT86RF230 transceiver [15] is connected to the master SPI interface of the microcontroller and to additional control signals, i.e. IRQ and GPIO signals. Essentially, the SPI is used for frame buffer and register access operations, according to the SPI protocol. Although AmForth already provides the words `spi!` and `spi@` for writing and reading a character on the SPI bus, further efforts were needed to configure ports for the specific target device.

We also defined word sets to support the functional specification of the radio device. For instance, in Listing 3 the words `reg_rd` and `reg_wr` specify the operations to be undertaken for reading and writing the radio registers. Similarly, we defined the words `to_framebuf` and `from_framebuf` to upload incoming frames, and download outgoing frames, respectively. Word choices reflect the nomenclature of the radio datasheet.

Uninterruptible code, such as that implementing SPI operations, is enclosed within critical sections. The words `ss_l` and `ss_h` set the SS line of the SPI interface respectively low and high.

Listing 3: Some words of the radio driver

```
: reg_rd ( register_address -- register_value )
      reg_addr_mask and reg_rd_command or
      critical[
      ss_h ss_l spi! spi@ ss_h
      ]critical
;

: reg_wr ( register_value register_address -- )
      reg_addr_mask and reg_wr_command or
      critical[
      ss_h ss_l spi! spi! ss_h
      ]critical
;

: to_framebuf ( packet_to_send -- packet )
      dup critical[
      ss_h ss_l framebuf_wr_command spi! length spi!
      dup length 0 ?do dup I + 1 + c@ spi! loop
      ss_h ]critical
;

: from_framebuf ( packet -- packet )
      critical[
      ss_h ss_l framebuf_rd_command spi!
      spi@ over c! dup length 0 ?do
      spi@ over I + 1 + c! loop ss_h ]critical
;
```

The radio transceiver operating modes and its transitions can be represented by the state diagram in Figure 2.

To permit a plain alignment between specifications and implementation, the same diagram can be completely ported into Forth definitions as shown in Listing 4, which includes just a restricted number of defined words.

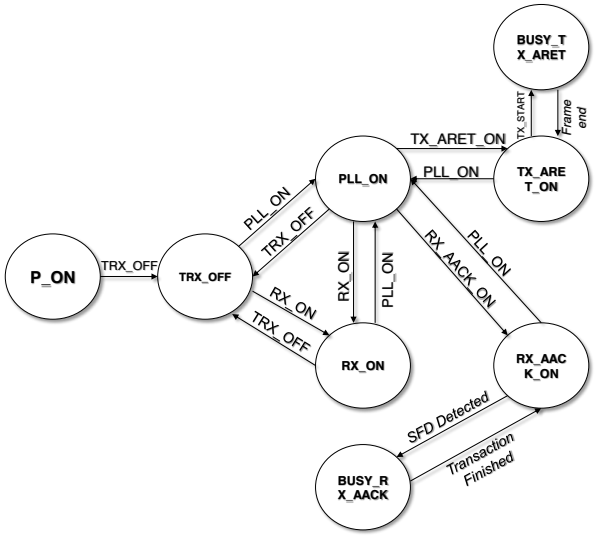


Fig. 2: Part of the state diagram representing the set of operating modes of the AT86RF230 according to the datasheet [15]. The label on the arrows represents the command that writes to the TRX_STATUS register causing the transaction to another state. Events are indicated with labels in italics.

Listing 4: Definitions for radio operating modes and transitions

```
: pll_on ( -- )
      pll_on_state cmd-wr  ;

: trx_off ( -- )
      st-reset ;

: tx_start ( -- )
      tx_start_cmd cmd-wr ;

: rx_on ( -- )
      rx_on_state cmd-wr  ;

: rx_aack_on ( -- )
      pll_on rx_aack_on_state cmd-wr ;

: tx_aret_on ( -- )
      pll_on tx_aret_on_state cmd-wr ;

: transmit ( packet -- )
     -int IRQ low
      idle? if green led blink else
      trx_off outbound process-tx drop to_framebuf drop
      tx_aret_on tx_start
      then +int
;

: switch-input? ( inbound -- )
      dup payld addr nip c@ $17 = if
      +radio-input $17 usart_inject
      then
;

: received ( inbound -- )
      trx_off
      from_framebuf dup process-rx switch-input?
      payld addr nip swap payld size payld-set
;
```

In particular the words `process-rx` and `process-tx` are deferred words that may be used to process `inbound` and `outbound` buffers. A possible use may be for encryption and decryption purposes. The last three definitions implement frame transmission, input redirection and frame reception. Transmission and reception of frames are signaled by interrupts on the Timer1 Input Capture Trigger. The Interrupt Service Routine in AmForth is also a defined word. Therefore, we defined a word acting as the handler routine and we stored its address as interrupt vector. Our interrupt handler routine reads the IRQ STATUS register and acts as a dispatcher. Since the AT86RF230 differentiates between six interrupt events, it calls the appropriate interrupt handler, according to the interrupt source. For instance, the interrupt generated by either a frame transmission/reception causes the execution of the word `trx_end_isr`. If a correct transmission triggers the interrupt, the radio enters the `rx_aack_on` state, otherwise the frame is downloaded to the `inbound` buffer.

```
: trx_end_isr
      red led blink state?
      tx_aret_on_state = if
      else inbound received
      then trac_status trac !
      rx_aack_on ;
```

C. Supporting Sensing and Actuating Tasks

The acquisition of sensory data is the main functionality of WSN nodes. Typically, expansion boards are required to provide the nodes with several sensors simultaneously. As in the case of the radio transceiver driver implementation, we have extended the WSN node dictionary with a number of words to drive sensor boards. Moreover, word sets composed of high level words enable the data sensory acquisition through the different available sensors. For instance, a program to make a node sense the temperature may consist of the single word `temperature` that leaves at the top of the stack the required sensory value. Similarly, the word `luminosity` activates the light sensor, puts the sensory reading atop the stack, and finally disables the sensor. Although the code is concise and expressive, the execution of these words involves low level aspects as reading from the ADC and returning the raw data on the stack. However, we choose high level word names to make the description of a task in natural language and the implementation as similar as possible. The words we defined to support sensing tasks are summarized in Table I.

WSNs may also include some actuator nodes to change the environmental conditions. An IrisMote can behave as an actuator when connected, for instance, to the MDA300 expansion board that includes two relays, one of which normally opened and the other one normally closed. We defined words to drive the relays and developed a light control application by connecting a LED to the expansion board, as detailed in Section IV.

III. A Forth-based Approach to Enable Symbolic Distributed Processing for WSNs

To implement sophisticated AmI applications, even resource constrained nodes may need to exchange complex information that is not rigidly structured and that may differ from numerical values such as symbolic descriptions and rules. Conventional programming methodologies impose to define in advance the format of the message to be exchanged as well

TABLE I: Summary table of words used for sensing tasks. Words are indicated together with their “stack effect”

Word name	Description
temperature (-- temp_value)	Measure temperature and push the numeric value onto the stack
luminosity (-- light_value)	Measure light and push the numeric value onto the stack
mic (-- mic_value)	Measure sound level and push the numeric value onto the stack
accx (-- accx_value)	Measure the acceleration along the X axis and push the numeric value onto the stack
accy (-- accx_value)	Measure the acceleration along the Y axis and push the numeric value onto the stack
+sounder (--)	Activate the buzzer
-sounder (--)	Disable the buzzer

as to fix the packet fields where given information must be placed. To overcome this rigidity, interpreters targeting WSN nodes, such as Maté [9], T-RES [6] and TakaTuka [10], have been presented. However, such solutions are based on bytecode transmission and interpretation. Not only the source code expressivity gets lost as the source code is translated into bytecode but also the translation process is, in all effects, a cross-compilation.

In order to retain expressiveness without sacrificing compactness, we let our nodes able to directly exchange and execute Forth code. Indeed, we implemented an abstract mechanism to handle the transmission of code among nodes, and from the terminal shell to nodes. The implementation cost of such an abstraction is quite low in Forth and, at the same time, the support to distributed applications is straightforward, as shown in Listing 5.

Listing 5: Forth words for executable code exchange

```
variable current_pay
variable nest
variable current_buf
variable buf $80 cells allot

: 2dup over over ;
: buf-reset buf current_buf ! ;
: pay-reset outbound payld addr nip current_pay ! ;

: (write) \ i*x addr len  dest_addr -- j*y
 swap cmove
;


: num>str ( number -- string_addr string_len )
      hex 0 <# #s [char] $ hold #> ;

: space+ ( pay_ptr -- pay_ptr+1)
      bl p+
;

: cr+ ( pay_ptr -- pay_ptr+1)
      $0d p+
;

: nest+ ( -- )
      nest @ 1 + nest ! ;
: nest- ( -- )
      nest @ 1 - nest ! ;

: [tell:]? ( addr len -- f )
      s" [tell:]" icompare ;

: [:tell]? ( addr len -- f )
      s" [:tell]" icompare ;

: tell:? ( addr len -- f )
      s" tell:" icompare ;

: :tell? ( addr len -- f )
      s" :tell" icompare ;


: >buf ( addr1 n -- )
      dup >R current_buf @ dup >R (write)
      R> R> + space+ current_buf ! ;

: >pkt ( addr1 n -- )
      dup >R current_pay @ dup >R (write)
      R> R> + space+ current_pay ! ;

: c>pkt ( value -- )
      current_pay @ swap ( c@ -- ) p+ current_pay ! ;

: char>buf ( value -- )
      current_buf @ swap over c! 1 + current_buf !
;


: subst
      0 do buf I + c@ dup ( c@ -- )
      case
      $0e of drop num>str >pkt endof
      $0f of drop num>str >pkt endof
      $10 of drop >pkt endof
      c>pkt
      endcase loop
;

: [endtell] ( flash-addr flash-count -- )
      dup >r buf imove r>
      subst outbound dup current_pay @ cr+
      endpayld transmit
      pay-reset
;

: endtell ( buf buf-len -- )
      nip subst outbound dup current_pay @ cr+
      endpayld transmit
      pay-reset
;

: subst? nest @ 0 = if
         2dup s" ~" icompare if drop drop $0e true else
         2dup s" ~~" icompare if drop drop $0f true else
         2dup s" ~s" icompare if drop drop $10 true else
         drop drop 0 then then then
         else drop drop 0 then
;


: parse-tell ( -- buf buf-len )
   buf-reset
   begin bl word count
         2dup :tell? if
           nest @ 0 > if nest- >buf 0
                    else true
                    then
         else
         2dup tell:? if nest+ >buf 0
         else 2dup [tell:]? if nest @ 3 + nest ! >buf 0
         else 2dup [:tell]? if nest @ 3 - nest ! >buf 0
         else 2dup subst? if char>buf drop drop 0
         else >buf 0
         then then then then then until drop drop
         buf current_buf @ over -
```

```
;

: [parse-tell]
   buf-reset
   begin bl word count
         2dup [:tell]? if
           nest @ 0 > if nest @ 3 - nest ! >buf 0
                   else true
                   then
         else
         2dup [tell:]? if nest @ 3 + nest ! >buf 0
         else 2dup tell:? if nest+ >buf 0
         else 2dup :tell? if nest- >buf 0
         else 2dup subst? if char>buf drop drop 0
         else >buf 0
         then then then then then until drop drop
         buf current_buf @ over -
;

: reply ( -- dest_addr)
      inbound src addr @ nip ;
;

: pkt-init 0 nest ! outbound erase
      default-pkt dest addr rot s_addr! drop
      pay-reset
;

: tell:
     pkt-init parse-tell endtell
;


: [tell:]
      postpone pkt-init
      [parse-tell] postpone sliteral
      postpone [endtell]
; immediate
```

Our programming environment and experimental setup is composed of some nodes wirelessly deployed and a wired node that behaves as a bridge to send user inputs to the network.

The syntactic construct for the code exchange among nodes is based on the word `tell:` that parses the input until `:tell` is encountered and sends a default data frame, according to IEEE 802.15.4 standard, to the node holding the MAC address placed on top of the stack.

To tell all the nodes in the radio range to turn their green LED on, a simple line of code is all that it needs to be typed on the bridge node shell:

```
bcst tell: green led on :tell
```

As a consequence, the microcontroller on the bridge node interprets the text typed by the user and creates a default data frame with the broadcast address as destination, containing the program to be sent, `green led on`, as payload.

A recursive usage of code exchange, through nested `tell:` *<code>* `:tell` constructs, permits commands to hop form one device to another before reaching the final destination, as shown in Figure 3. From a mere semantic standpoint, the sense is *"to tell a node to tell another node to do something"*.

The code to be remotely executed may contain syntactic placeholders that are substituted at runtime with the content of the top of the stack using a hexadecimal representation. For the sake of clarity, our implementation consists in a two pass parsing process. An intermediate substitution of such special markers takes place in the first pass, while the items on top of the stack definitively replaces placeholders during the second pass.

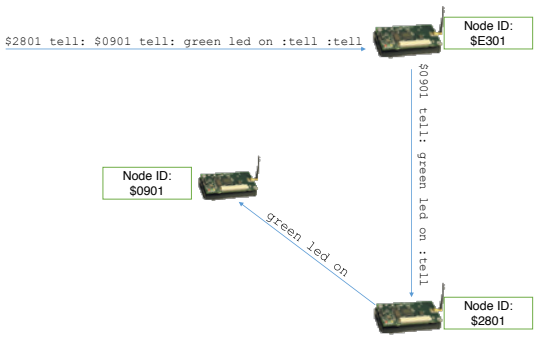


Fig. 3: Recursive employment of the `tell: <code> :tell` primitive. Once the node with ID E301 encounters the first `tell:` it parses all the following symbols until the last `:tell` and a frame containing `0901 tell: green led on :tell` is sent to node with ID 2801. The payload interpretation of the payload on the receiving side leads to the sending of a new frame destinated to node 0901 with `green led on` as payload. Once received, node 0901 turn its green LED on.

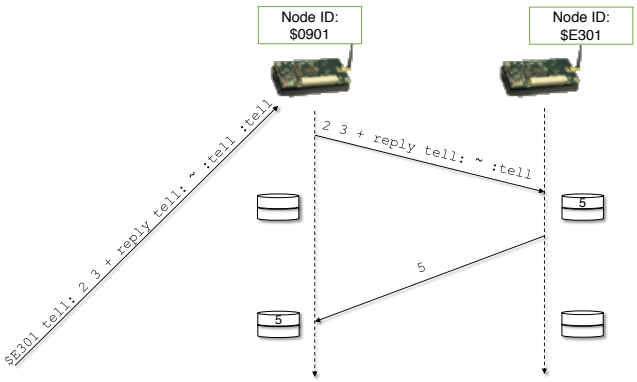


Fig. 4: The node with address $0901 interprets the code it receives and tells the node with address $E301 to perform the sum between 2 and 3, and then to reply with the value on top of its stack. Even though the reply message consists only in a literal value, it is interpretable Forth code and it is simply executed by node $0901 leaving 5 on top of its stack.

Such special markers are:

- ~ for a single cell value
- ~ ~ for a double cell value
- ~s for strings

Instead of implementing state-smart words for code exchange [16], we defined the compile-time construct `[tell:]`*<code>*`[:tell]`.

An example of code exchange between two nodes is described in Figure 4. Incorporating such high level abstraction on resource constrained devices leaves plenty of room for the development of WSN applications that natively support distributed processing. The word sets composing our software platform are reported in Figure 5 along with their size in terms

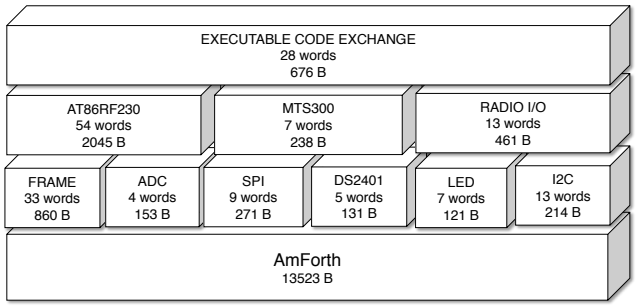


Fig. 5: A comprehensive view of the main word sets we defined and that compose our software platform. For each word set, the number of words and the Flash usage in bytes are indicated.

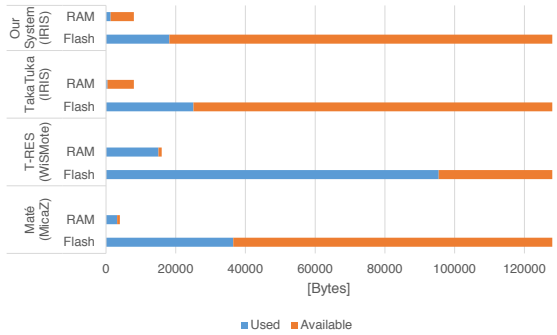


Fig. 6: Memory footprint of our software platform along with some representative interpreter-based architectures.

of number of words and Flash memory occupation.

Besides providing an on-board interpreter that does not need cross-compilation, our approach compares favorably with the aforementioned interpreter-based architectures with respect to memory usage, as we assessed with tests in our experimental setup. Where possible, as for TakaTuka and Maté, we compiled the software platforms for the IrisMote or for the quite similar MicaZ hardware. T-RES, instead, only runs on the WiSMote hardware platform.

Results, reported in Figure 6, confirm that the implementation of the interpreter above a general purpose operating system occupies much of the available memory, as in the case of Maté and T-RES. As scripts are stored in RAM, not enough space is left for the development of complex applications, even in the presence of the double-sized RAM of WiSMotes. Our Forth-based approach, instead, compactly keeps application code in the relatively abundant Flash, while RAM just holds temporary data as variable values, buffers and stacks.

More in detail, the memory footprint of all the platform word sets is 5170 bytes of Flash, as reported in Figure 6, and 1026 bytes of RAM. Including the underlying AmForth, the overall footprint of our platform is 18693 bytes of Flash memory and 1321 bytes of RAM memory.

IV. Application Development on WSNs

We have developed different applications for WSNs to test both our approach and our software platform. As a first step, we designed and implemented a working telnet-like remote shell on the bridge node to be actually used as a development tool [17]. Using the remote shell application on the bridge node through a serial terminal, the programmer can interact with a remote node that is reachable by the bridge node.

Besides debugging and node reprogramming, this application can serve different purposes such as the inspection of the state of a remote node or the acquisition of sensory readings as if the remote node were physically connected through the serial line.

The code is fully functional, and has been extensively used in our experimentations. We defined a number of words to redirect the output to the outgoing message, to display incoming messages from the inspected node, and to implement the remote shell loop. The resulting implementation is quite readable and understandable. The almost complete remote shell application code is shown in Listing 6. Although few additional words are omitted, their description can be found in Table II.

Listing 6: Code for a simple remote shell application

```
80 constant cmd-maxlen
variable cmd cmd-maxlen cells allot
variable cmd-len
variable node_id
variable timeout

: input-send ( -- )
    cmd cmd-len @
    node_id @ [tell:] ~s [:tell] ;

: rshell-task (  -- )
    payld-reset
    input-send
    timeout @ wait-answer if
    payld-print then ;

: user-input ( -- )
    cmd cmd-maxlen accept ( -- len )
    cmd-len ! ;

: close ( -- )
    node_id @ [tell:] -radio-output
    [:tell] quit ;

: rshell-loop ( -- )
    begin
     cr ." rsh>" user-input
     close?
     if close
     else rshell-task
     then
    again ;

: on-timeout ( -- )
    ." Connection timeout." cr ;

: welcome-msg
    ." Welcome to the remote shell
 application!" cr
    ." Enter 'close' to close the
 application" cr ;

: rshell ( id -- )
    welcome-msg
    2000 timeout  !
    dup node_id !
    [tell:] +radio-output [:tell]
```

```
    rshell-loop  ;
```

A desirable use of WSN nodes is monitoring the environment to react to undesired events. A role-based working implementation differentiates the words defined on remote nodes on the basis of their role in the network.

For instance, the actuator node dictionary could include words to trigger an alarm if the luminosity value exceeds a predefined threshold. A node provided with a light sensor may regularly perform the luminosity measurement and tell its neighbor to forward it to the actuator.

Such words explicitly make use of the syntactic construct for distributed code exchange. However, the designer may interactively set the topology, the threshold, the actuator node and may start the event detection application by interacting with the bridge node shell.

The code to implement such an application is provided in Listing 7 and consists in few words defined on the three kinds of nodes. With respect to the baseline of our platform, the increment of RAM usage on the sensor node for the application is just 6 bytes, while additional 156 Flash bytes are required to store the word definitions for timer3 management and the application. Since the routines for timer3 are not needed on the forwarder and actuator nodes, Flash and RAM increments for both are just 86 and 4 bytes respectively.

More importantly, the overall Flash and RAM memory footprint of the application and the software platform, even considering 21 additional bytes for the turnkey definitions, is 18714 bytes of Flash memory and 1321 bytes of RAM memory. Such result is even lower than the baselines of the other platforms –that is without any application– as can be deduced from Figure 6.

Listing 7: Distributed event detection application on WSN nodes

```
\ Defined on the sensor node
variable neighbor
variable threshold

: luminosity-check
        luminosity
        threshold @ > if
        neighbor @ [tell:] alarm [:tell]
        then ;

: light-monitoring
        ['] luminosity-check
        5seconds timer3.init timer3.start ;

\ Defined on the forwarder node
\ and on the actuator node

variable actuator
variable neighbor

: same ( -- outbound)
        outbound inbound over length copy ;

: message ( dest_addr outbound --outbound)
         dest addr rot s_addr! ;

: propagate
        transmit ;

: actuator?
        actuator @ 1 = ;

: alarm
        actuator? not if
        neighbor @ same message propagate
        else +sounder 1000 ms -sounder
        then ;
```

Furthermore, instead of an alarm, the actuator may directly switch the light off once the luminosity exceeds the threshold value through a redefinition of the word `alarm` (Listing 8).

Listing 8: Redefinition on the actuator node of the word that triggers the alarm

```
: alarm
        actuator? not if
        neighbor @ same message propagate
        else     light off then ;
```

In previous work [18] we have also showed how to support smart applications that exploit symbolic reasoning. We enriched a Forth formalism for Fuzzy Logic by VanNorman [19] with the possibility to exchange definitions and evaluations among nodes. Instead of reasoning about crisp values, resource constrained nodes process the fuzzy variables `temp` and `lightexp` that can be easily defined on deployed nodes. Further words such as `fvar` define the related membership functions. By exploiting executable code exchange, a fuzzy variable definition can be easily distributed among nodes even after their deployment. After defining the membership functions `lightexp.low`, `lightexp.medium`, `lightexp.high` and `temp.low`, `temp.medium`, `temp.high`, the following code makes a node measure and fuzzify light exposure:

```
lightexp measure apply
```

while the code:

```
lightexp.low @
```

pushes onto the stack the truth value resulting from the fuzzification phase. For instance, rather than through a thresholding process, a device can establish if it is close to the window through the evaluation of fuzzy rules in the form:

```
temp.high @ lightexp.high @ &
=> close-to-window
```

A node can request the others to update their fuzzy temperature values as follows:

```
bcst temp fvar-remote-update
```

The word `fvar-remote-update` evaluates `temp` and broadcasts a message containing the Forth code to update the three membership values. The frame payload includes a repetition of the structure:

```
<truth> <membership func> fvar-update
```

for each membership function of the argument fuzzy variable. When a node receives the message, it interprets the command updating the truth values of its local membership functions. The combination of symbolic reasoning with executable code exchange makes even resource constrained devices able to process and exchange qualitative information about the physical phenomenon.

TABLE II: Summary table of additional words used in the remote shell application

Word (before -- after)	**Description**
+radio-output (--)	Redirect the output to the radio. This word is part of the Radio I/O word set
-radio-output (--)	Redirect the output to the UART. This word is part of the Radio I/O word set
radio-input? (timeout -- flag)	Check for incoming radio messages. If no message arrives before `timeout` milliseconds leave false on the stack, otherwise leave true
payld-reset (--)	Set to 0 the incoming payload length and its current pointer
payld-print (--)	Display the incoming frame content (i.e. the payload)
wait-answer (timeout -- flag)	Wait for incoming radio frame for a predefined period of time specified by the `timeout` variable. If the timeout expires without receiving any answer message, an exception handled by `on-timeout` occurs
user-input (--)	Wait for user input and store its content in the `cmd` buffer and its length in the `cmd-len` variable for further processing by `close?` and `input-send`

V. Conclusions

As remarked in literature, common programming methodologies for WSNs lack proper programming abstractions for the development of distributed applications. The standard practice consists in linking the application, written in C-derived programming languages, with a general-purpose operating system at the end of a cross-compilation process. All this proves rigid and time consuming. To overcome these limitations, the adoption of interpreters for high-level languages to be run on established operating systems has been proposed. Nevertheless, existing approaches consist in several software layer implementations that collide with the resource constraints of nodes.

In this paper, we detailed the implementation of an alternative Forth-based approach that implements a minimal but extensible operating system featuring common WSN functionalities along with symbolic distributed processing through executable code exchange. The Forth-based software platform we have implemented is quite compact. Indeed, we showed how a symbolic distributed AmI event detection application can be implemented with a total memory usage that is less than the mere baselines of relevant interpreter-based software platform for WSNs. In further experimentations we will compare our Forth environment to other existing interpreter-based platforms for WSNs in terms of efficiency, interpretation overhead and energy consumption.

References

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communication Magazine*, vol. 40, no. 8, pp. 102–114, 2002.

[2] E. Gilbert, K. Baskaran, and E. E. Blessing, "Research Issues in Wireless Sensor Network Applications: A Survey," *International Journal of Information and Electronics Engineering*, vol. 2, no. 5, pp. 702–706, 2012.

[3] M. O. Farooq and T. Kunz, "Operating systems for wireless sensor networks: A survey," *Sensors*, vol. 11, no. 6, pp. 5900–5930, 2011.

[4] A. M. V. Reddy, A. P. Kumar, D. Janakiram, and G. A. Kumar, "Wireless sensor network operating systems: a survey," *Int. J. Sen. Netw.*, vol. 5, no. 4, pp. 236–255, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1504/IJSNET.2009.027631

[5] L. M. Oliveira and J. J. Rodrigues, "Wireless Sensor Networks: a Survey on Environmental Monitoring," *Journal of communications*, vol. 6, no. 2, pp. 143–151, 2011.

[6] D. Alessandrelli, M. Petracca, and P. Pagano, "T-Res: Enabling Reconfigurable In-network Processing in IoT-based WSNs," in *Distributed Computing in Sensor Systems (DCOSS), 2013 IEEE International Conference on*, May 2013, pp. 337–344.

[7] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1922649.1922656

[8] L. Evers, P. Havinga, J. Kuper, M. Lijding, and N. Meratnia, "SensorScheme: Supply chain management automation using Wireless Sensor Networks," in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, Sept 2007, pp. 448–455.

[9] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," in *ACM Sigplan Notices*, vol. 37, no. 10. ACM, 2002, pp. 85–95.

[10] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rhrup, and Z. Uzmi, "Optimized Java Binary and Virtual Machine for Tiny Motes," in *Distributed Computing in Sensor Systems*, ser. Lecture Notes in Computer Science, R. Rajaraman, T. Moscibroda, A. Dunkels, and A. Scaglione, Eds. Springer Berlin Heidelberg, 2010, vol. 6131, pp. 15–30.

[11] W. Munawar, M. Alizai, O. Landsiedel, and K. Wehrle, "Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks," in *Communications (ICC), 2010 IEEE International Conference on*, May 2010, pp. 1–6.

[12] B. Watts, "FORTH, a Software Solution to Real-time Computing Problems," *Behavior Research Methods, Instruments, & Computers*, vol. 18, no. 2, pp. 228–235, 1986.

[13] "Iris Datasheet," 2013, available online at http://www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf.

[14] "Amforth documentation," 2013, available online at http://amforth.sourceforge.net/amforth.pdf.

[15] "At86rf230 datasheet," 2013, available online at http://www.atmel.com/images/doc5131.pdf.

[16] M. A. Ertl, "State-smartness— Why it is Evil and How to Exorcise it," *EuroForth98*, 1998.

[17] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri, "A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, Sept 2014, pp. 1–8.

[18] S. Gaglio, G. L. Re, G. Martorella, and D. Peri, "High-level Programming and Symbolic Reasoning on IoT Resource Constrained Devices," in *Accepted at The First International Conference on Cognitive Internet of Things (COIOTE 2014)*, October 2014.

[19] R. VanNorman, "Fuzzy Forth," *Forth Dimensions*, vol. 18, pp. 6–13, Mar. 1997.

Chat mit net2o

Bernd Paysan

Wir halten unseren wöchentlichen Forth–Chat seit letztem Sommer statt im IRC in net2o ab, und beseitigen dabei stückweise verbliebene Probleme. Ebenfalls soll die Einstiegshürde sinken, denn net2o soll ja auch einfach zu benutzen sein. Dieser Artikel erklärt, wie man in net2o einsteigt, und wie man dann am Donnerstag–Abend–Chat teilnehmen kann.

net2o installieren

Net2o ist noch in einer relativ frühen Testphase, also kann sich jederzeit etwas am Protokoll ändern, das dann nicht mit der Vorgängerversion kompatibel ist. Das bedeutet dann auch, dass man für den wöchentlichen Chat „up to date“ sein muss. Manchmal ändern sich sogar in den Minuten vor dem Chat noch ein paar wichtige Dinge. net2o hängt von Gforth ab, das heißt natürlich, wenn ich Fehler finde, deren Ursache im Gforth sind, dann baue ich da keinen Workaround, sondern fixe das Problem in Gforth direkt.

Natürlich versuche ich, net2o einfacher installierbar zu machen, es soll ja nicht noch ein weiteres schwer zu installierendes, kryptisch zu bedienendes Programm sein, das dann seinen Zweck verfehlt, die Kommunikation sicherer zu machen, weil es niemand benutzt. Deshalb soll net2o auch einfach zu installieren, aufzusetzen und zu benutzen sein: Key erzeugen, sich bekannt machen, loslegen. Im Moment ist die Benutzeroberfläche noch rein textbasiert, das wird sich auch noch ändern.

Fertig zum Installieren und Updaten ohne groß nachzudenken gibt es net2o zur Zeit für Debian–basierte Linuxe und Android. Für andere Betriebssysteme muss (noch) aus dem Quellcode selber compiliert werden. Windows steht als nächstes an, für eine setup.exe. Fangen wir aber mit den einfachen Varianten an:

Installieren auf Android

net2o ist als Beispielanwendung in das Gforth.apk auf Google Play eingebaut. Es wird gestartet, indem man auf das net2o–Icon klickt (das wird nicht automatisch auf den Homescreen geklatscht, aber in der kompletten Übersicht ist es zu finden). Das ist also ganz einfach: Gforth auf Google suchen, installieren, fertig. Auto–Update einschalten, dann bekommt man automatisch die neuste Version (mit der Google–üblichen Verzögerung).

Wer aus diesem oder jenem Grund keinen Google Play Store auf seinem Android hat, kann die jeweils aktuelle Gforth.apk direkt von meiner Homepage herunterladen: `https://net2o.de/Gforth.apk`. Da ist das Update schneller als bei Google Play. Das ist jetzt mit allem, was Gforth mitbringt; in Zukunft wird es wohl auch eine eigene net2o–App geben, die nur das Nötigste enthält (nicht alle Forth–Quelltexte), und damit schlanker ist.

Installieren auf Debian/Ubuntu

Für Debian habe ich ein Repository aufgesetzt, damit ist das Installieren von net2o und einem aktuellen Gforth recht einfach. Unterstützt werden aktuell die Architekturen amd64, i386 und armhf.

Um das Repository einzubinden, öffnet man eine Root–Shell, und gibt hier ein:

```
cat >/etc/apt/sources.list.d/net2o.list <<EOF
deb [arch=...,all] http://net2o.de/debian testing main
EOF
wget -O - https://net2o.de/bernd@net2o.de.gpg.asc \
| apt-key add -
aptitude update
aptitude install net2o
```

Auf älteren Debian–Versionen als auf dem aktuellen Testing muss ich nach `deb` in der `net2o.list` noch die Architekturen mit `[arch=i386,all]`, `[arch=amd64,i386,all]` oder `[arch=armhf,all]` eingeben, sonst wird das Common–Paket nicht gefunden.

Darauf erscheint dann diese Meldung:

```
The following NEW packages will be installed:
  ed25519-prim{a} gforth{a} gforth-bin{a}
  gforth-common{a} gforth-lib{a} keccak{a}
  net2o threefish{a}
0 packages upgraded, 8 newly installed, 0 to
remove and 0 not upgraded. Need to get 720 kB/
2.320 kB of archives. After unpacking 11,6 MB will
be used.
Do you want to continue? [Y/n/?]
```

Diese Frage beantworten wir mit Return. Eigentlich sind alle Pakete signiert, aber ältere Debian–Versionen meckern gelegentlich das eine oder andere Paket an. Damit haben wir dann ein aktuelles Gforth, alle Crypto–Libraries, und den net2o–Code. Alle Updates gehen dann automatisch mit `aptitude update && aptitude upgrade`.

Installieren aus den Quellen

Das funktioniert grundsätzlich auf allen möglichen Linuxen, Cygwin/Cygwin64 und Mac OS X, ist aber natürlich nicht ganz so einfach wie das Installieren aus einem Repository. Man braucht die Build–Tools, libtool und bei neueren Debians libtool-bin, yodl (für swig), und libltdl7. Wer Gforth mit Manual haben möchte, braucht noch texinfo und texlife-texinfo (OpenSuSE) und ggf. a2ps (Debian); für net2o ist das nicht nötig.

Dann legt man sich ein net2o–Verzeichnis an, und holt sich `https://fossil.net2o.de/net2o/doc/trunk/do` (geht direkt mit wget). Diese Datei ausfürbar machen, und ausführen. Wenn alles klappt, hat man eine lauffähige Version von net2o.

Für das wöchentliche Update kann man auch einfach wieder `./do` aufrufen, oder, falls sich nur die Forth–Quellen geändert haben,

```
fossil up
make && sudo make install
```

Das reicht aus, wenn sich nichts Wichtiges am Gforth geändert hat. Das `./do` versucht aber auch, nichts Unnötiges neu zu kompilieren, also kann man auch einfach jedesmal ein `./do` ausführen.

net2o einrichten

net2o installiert ein Kommandozeilen–Tool `n2o`, das Kommandos entweder direkt aus den Argumenten interpretiert, oder mit `n2o cmd` in einen Kommandozeilen–Modus startet. Alle folgenden Kommandos also entweder mit `n2o` vorne dran von der Shell aus aufrufen, oder mit `n2o cmd` die Kommandozeile starten, und dort eingeben. Die Android–App macht genau das: net2o im Kommandozeilen–Modus starten. Hilfe bekommt man mit `help` [<*cmd*>], ohne Kommando einfach die Liste aller Kommandos, mit Kommando etwas ausführlicher zum spezifischen Kommando.

Key erzeugen

Wenn man net2o zum ersten Mal frisch installiert hat, muss man sich erst einmal eine Kennung, also einen Key geben. Die Android–App fordert einen dazu direkt auf, ohne dass man ein Kommando eingeben muss.

`keygen` <*nick*> erzeugt einen Key mit einem symbolischen Namen, und exportiert diesen Key als Datei <*nick*>`.n2o`. Es wird auch ein Revocation Key erzeugt, den man sich am besten sofort aufschreibt. Hat man keinen Stift zur Hand, kann net2o diesen Revocation Key speichern, er ist aber dann genauso verletzlich wie der eigentliche Secret Key, und das ist schlimmer als gar kein Revocation Key: Damit kann ein Angreifer dann eine Revocation durchführen, und man hat selbst keinen Zugang mehr zu dem glaubwürdig verteilten neuen Key.

`keygen` fragt auch nach einem Passwort, dieses Passwort schützt den Secret Key und alle damit verschlüsselten Dateien vor einem Angreifer, der Zugang zum Gerät hat. Nehmen wir an, das ist bereits verschlüsselt mit einem guten Passwort, dann besteht keine Notwendigkeit, sich hier nochmal ein starkes Passwort zu geben.

Man kann beliebig viele Keys (und unterschiedliche zugehörige Nicks) erzeugen, und die Passwörter auch nachträglich ändern, sowie das „Passwort–Level“, das die Zahl der Hashrunden angibt (in $256*4^n$), die das Passwort wiederholt gehasht wird, um einen Brute–Force–Angreifer auszubremsen. Default ist 2, sinnvolles Maximum ist 4, mehr wird auf aktuellen Maschinen zu langsam. Beim Öffnen gibt man keinen User–Namen ein, sondern nur das zum Key gehörige Passwort; falls mehrere damit verschlüsselt sind, wählt man das gewünschte aus der Liste.

Keys importieren

Um mit jemandem zu chatten, braucht man erst mal dessen Key. Beim Group–Chat reicht, den Key des direkt angesprochenen Knotens zu kennen, die Keys der anderen Teilnehmer erfährt man, wenn sie sich bemerkbar machen. Wichtig ist jetzt erst mal mein Key. Um den zu bekommen, sucht man ihn einfach:

```
keysearch kQusJ
```

Keys werden als base85–Zahl codiert, jeweils 5 Buchstaben sind 32 Bits. Den erfolgreich importierten Key kann man nun mit

```
keylist
```

ausgeben lassen, und neben dem eigenen Key sollten nun

```
 0 (T!y^QYGM7x~WuVP2P-9{WxlsZTE_Jv2Z|l*x6v1
2015-04-07T19:29Z->never c-dmr--h- net2o-dhtroot
 1 kQusJzA;7*?t=uy@X}1GWr!+0qqp_Cn176t4(dQ*
2015-11-23T19:38Z->never c-dmr--h- bernd
```

aufgelistet sein (nicht unbedingt in der Reihenfolge, aber mit diesen base85–Daten). Die beim Erzeugen des Keys angelegte Datei kann man auch z.B. per E–Mail verschicken und mit `n2o keyin` <*nick*>`.n2o` einlesen, aber es gibt auch eine reine Net2o–Variante, um sich bekannt zu machen.

Einladung abschicken

Wenn der Partner seinen Chat vermutlich bereits offen hat, kann man ihm eine Einladung schicken:

```
invite @<nick>
```

schickt ihm eine Einladung. Er kann sie dann aus dem Chat mit dem Chat–Kommando `/invitations` annehmen, ignorieren, oder einen in die Block–Liste werfen. Ob und wann er das gemacht hat, erfährt man nicht vom Protokoll, man kann aber schon mal einen Chat aufmachen, vielleicht spricht er einen ja an.

Chat starten

Es gibt zwei Sorten von Chats: 1:1–Chats und Gruppenchats. Bei den Gruppenchats gibt es noch den Unterschied, ob man sie initiiert, oder sich mit anderen verbindet. Gruppen–Chats werden über eine baumartige Verbindung verteilt, man kann sich deshalb bei jedem Gruppenmitglied anmelden, und wird Teil des Baumes. Der Baum kann dynamisch restrukturiert werden, wenn Mitglieder den Chat verlassen oder zu viele an einem Knoten hängen. Die drei Formen sehen so aus:

```
chat @<nick> \ 1:1-Chat
chat <group> \ Gruppen-Chat initiieren
chat <group>@<nick> \ Gruppen-Chat beitreten
```

Im Chat kann man diese Operationen als Chat–Kommando (mit / oder \ vorne dran) auch ausführen, und damit Chat–Kanäle wechseln und Verbindungen aufbauen. Bauen wir also eine Verbindung zum Forth–Chat auf:

```
chat forth@bernd
```

Es ist grundsätzlich auch möglich, mit sich selbst über mehrere Rechner zu chatten, dazu gibt man dann noch den Rechnernamen an, also sagen wir, Rechner 1 heißt „foo“ und Rechner 2 „bar“ (Rechnernamen am Prompt):

```
foo% n2o chat test
bar% n2o chat test@foo.mynick
```

Der erste startet den Chat ohne direktes Ziel (das wäre ja noch nicht online), der zweite gibt den Rechnernamen an. Man kann den Rechnernamen auch erfragen mit

```
lookup @<nick>
```

Das listet dann alle öffentlich gemachten Adressen (die über den DHT–Server geroutet werden, d.h. die eigenen IP–Adressen sind nicht sichtbar, wohl aber die Rechnernamen). Wenn ich online bin, sollte sich folgendes Bild ergeben:

```
% lookup @bernd
bernd:
10#"daiyu"[2A03:4000:2:188::1]37.221.194.73:4242|71
20:19:18Z->never
```

Die Uhrzeit der Signatur enthält auch ein Ablaufdatum, das hier auf „unendlich“ gestellt ist.

Mit

```
chatlog forth
```

kann man den Chatlog der Gruppe Forth ausgeben lassen; da gibt es noch die Optionen `-bw` (ohne Farbmarkierungen) und `-date` <*n*> für den Detailgrad der Zeitangabe, diese Optionen sind Optionen für das n2o–Commando, also gibt man sie vor dem `chatlog` an.

Chatten

Wenn wir jetzt im Chat sind, gibt es auch wieder Kommandos. Kommandos werden mit / oder \ eingeleitet (damit sowohl IRC– als auch LaTeX–Nutzer auf ihre Kosten kommen). `/help` gibt gleich mal aus, was für Kommandos es gibt. Alle Eingaben, die nicht mit / oder \ anfangen, werden als Message weitergesendet; fängt man eine Message mit @<nick> oder @<nick>: an, so wird der Nick in einen Key verwandelt, und beim Empfänger wieder in einen Nick, damit kann jeder seine eigenen Nicks pflegen, ohne dass diese nach außen bekannt werden. Man sollte sich dabei nur nicht vertippen, die Suche ist weder Case–insensitiv noch fängt sie Tippfehler ab. Wenn es kein bekannter Nick nach dem @ ist, wird der String so gesendet, wie er ist.

Diese Kommandos sind im Moment definiert, einige davon helfen, net2o–Internas zu verstehen und zu debuggen, andere sind für den Chat selbst nützlich

/me <***action***> Kennt man aus IRC, es wird eine Message angezeigt, dass der User eine bestimmte Aktion durchführt. Beispiel am Chatende „bernd macht das Licht aus“

/bye Beendet den Chat, ctrl–D funktioniert auch

/peers Listet die Peers des aktiven Chats auf

/here Sendet die GPS–Koordinaten als „ich bin hier“–Message (wo GPS vorhanden)

/help Die Hilfefunktion für den Chat

/invitations Zum Bearbeiten von Einladungen, die während des Chats angekommen sind. Man kann eine Einladung annehmen, ignorieren oder den Einlader dauerhaft blockieren.

/chats Listet eine kurze Übersicht über alle Chats und die Anzahl Peers in jedem

/nat Zeigt Informationen zum NAT–Traversal der Peers. Da net2o auch hinterm Firewall oder sogar in einem Mobilnetzwerk (mit Carrier–Grade–NAT) gehen soll, sind diese Informationen hilfreich. Hat es nicht optimal geklappt, kann man mit

/renat versuchen, den NAT–Traversal erneut durchzuführen

/notify Einstellung für die Benachrichtigungen beim Eintreffen von Messages

always Immer, unabhängig vom Bildschirm

on Nur an, wenn der Bildschirm aus

off Aus

led <***rgb***> <***on–ms***> <***off–ms***> LED–Einstellung am Smartphone, die Farbe sind 6 Hex–Digits, die An– und Auszeiten sind in Millisekunden, dezimal

interval <***time***>**[smh]** Benachrichtigungen sollen nicht zu sehr nerven, nach jeder Benachrichtigung kann man eine Pause einstellen, die in Sekunden, Minuten oder Stunden bemessen wird

mode 0–3 Stellt den Benachrichtigungsmodus ein, also Ton und Vibration. 1 ist Ton, 2 ist Vibration, beides zusammen ist 3.

/beacons Listet alle „Beacons“, die zu verbundenen Peers alle Minute gesendet werden, damit der Firewall die Verbindung offen hält

/n2o <***cmd***> führt ein net2o–Commando aus dem Chat–Modus aus (z.B. Dateien übertragen oder Einladungen versenden).

/chat @<***user***>**/**<***group***>**[@**<***user***>**]** Wechselt den Chat–Room, oder legt einen neuen an (inklusive Verbindungsaufbau, falls ein User angegeben ist)

Referenzen

[1] Bernd Paysan, *„net2o: Get it“,* Fundstelle `https://fossil.net2o.de/net2o/doc/trunk/wiki/get-it.md`

„Entprellung“ von AC–Optokopplern

Rafael Deliano

Die Überwachung, ob mehrere 220V–Leitungen geschaltet sind, kann einfach mit Optokopplern efolgen (Abb.1). Das Signal hat im Nulldurchgang einen Aussetzer (Abb.2 und 3).

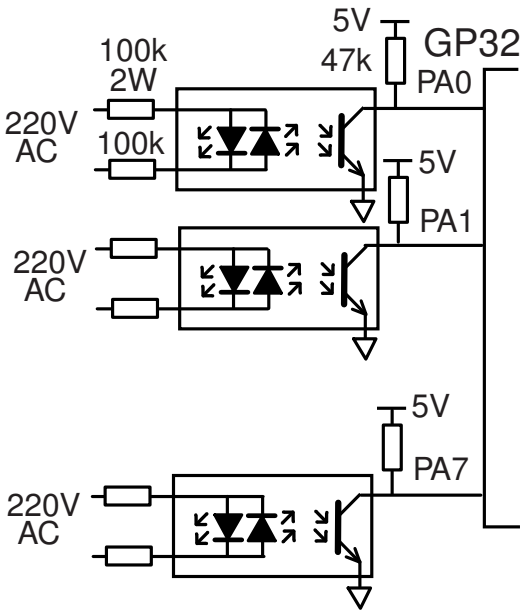


Abbildung 1: Schaltung

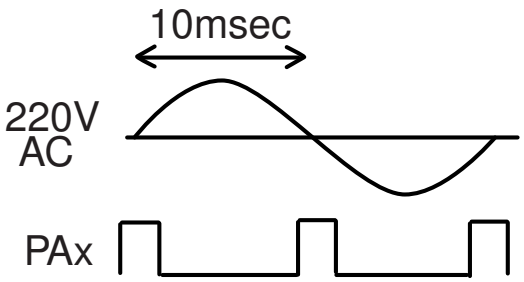


Abbildung 2: Signal

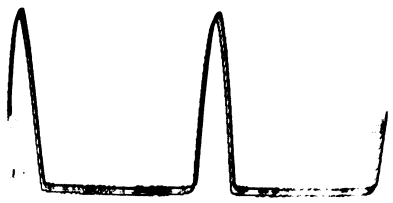


Abbildung 3: Oszilloskopbild des Signals an PAx

Die Netzspannung variiert nur um ±20% (Abb.4). Der Koppelfaktor des Optokopplers LTV814 streut mit 20% ... 100% ... 300% deutlich stärker. Man sollte den pullup–Widerstand nicht zu klein wählen (Abb.5,6,7), weil man sonst im ungünstigen Fall nicht durchschaltet, jedenfalls aber nur einen kurzen Puls erhält.

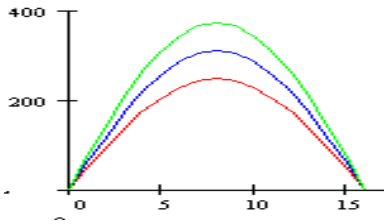


Abbildung 4: Netz ±20% ; [Zeit]/[Volt]

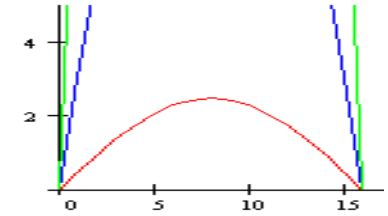


Abbildung 5: 10k pullup

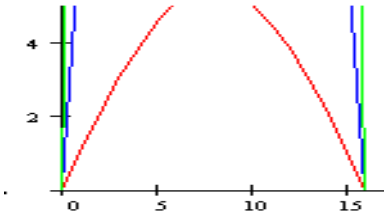


Abbildung 6: 22k pullup

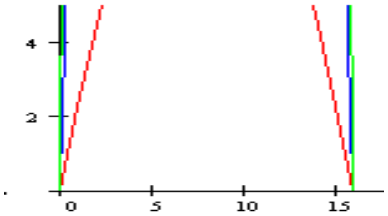


Abbildung 7: 47k pullup

Passende Software ist stabiler als ein simpler Kondensator. Wie implementiert wird, hängt von der Anwendung ab. Der bequemste Fall ist eine Hauptschleife, die asthmatisch langsam läuft. Man kann dann bei Abfrage eines Pins für etwas über 10msec stoppen, um den Puls abzuwarten (Listing1):

```
Startwert = 0
Port lesen, invertieren, verodern, 1msec warten
...
Port lesen, invertieren, verodern, 1msec warten
Bit mit AND ausmaskieren.
```

Die meisten Systeme müssen schneller sein und haben einen Systemtick der im Interrupt läuft. Aus dem kann man alle 1msec eine Routine aufrufen, die den Portzustand kontinuierlich in eine Tabelle im RAM loggt:

`Port lesen, über Index in Tabelle schreiben, Index modulo inkrementieren.`

Das würde man typisch in Assembler codieren (Listing2a). Die Leseroutine im Hauptprogramm (Listing2b) wertet das gleitende Fenster aus:

`Alle 12 Bytes lesen, invertieren, verodern, Bit mit AND ausmaskieren.`

Die Variante ist auf den Fall, dass man bis zu 8 Eingänge auswerten muss, zugeschnitten (Abb.1).

Listings

```
1   Listing 1: Langsames Lesen eines Pins
2   : PA0?    \ (  --- Flag ) 1 = power
3   00 D% 11 0 DO
4   PA C@ NOT      \ get port & invert
5   OR             \ merge
6   1 MSEC LOOP    \ delay
7   B% 01 AND ;    \ mask
8
9   Listing 2a: Interruptroutine Pseudocode
10  D% 12 ZVARIABLE OPTO-BUFFER
11      1 ZVARIABLE OPTO-INDEX
12
13  : 1msec-IRQ    \ (  ---  )
14  PA C@ OPTO-BUFFER OPTO-INDEX c@ + C!
15  OPTO-INDEX C@
16  DUP D% 11 = IF DROP 0 ELSE 1 + THEN
17  OPTO-INDEX C! ;
18
19  Listing 2b: Lesen des Pins im Hauptprogramm
20  : PA0?    \ (  ---  Flag ) 1 = power
21  00 D% 11 0 DO
22  OPTO-BUFFER I + C@ \ get "port"
23  NOT OR             \ invert & merge
24  LOOP B% 01 AND ;   \ mask
25
```

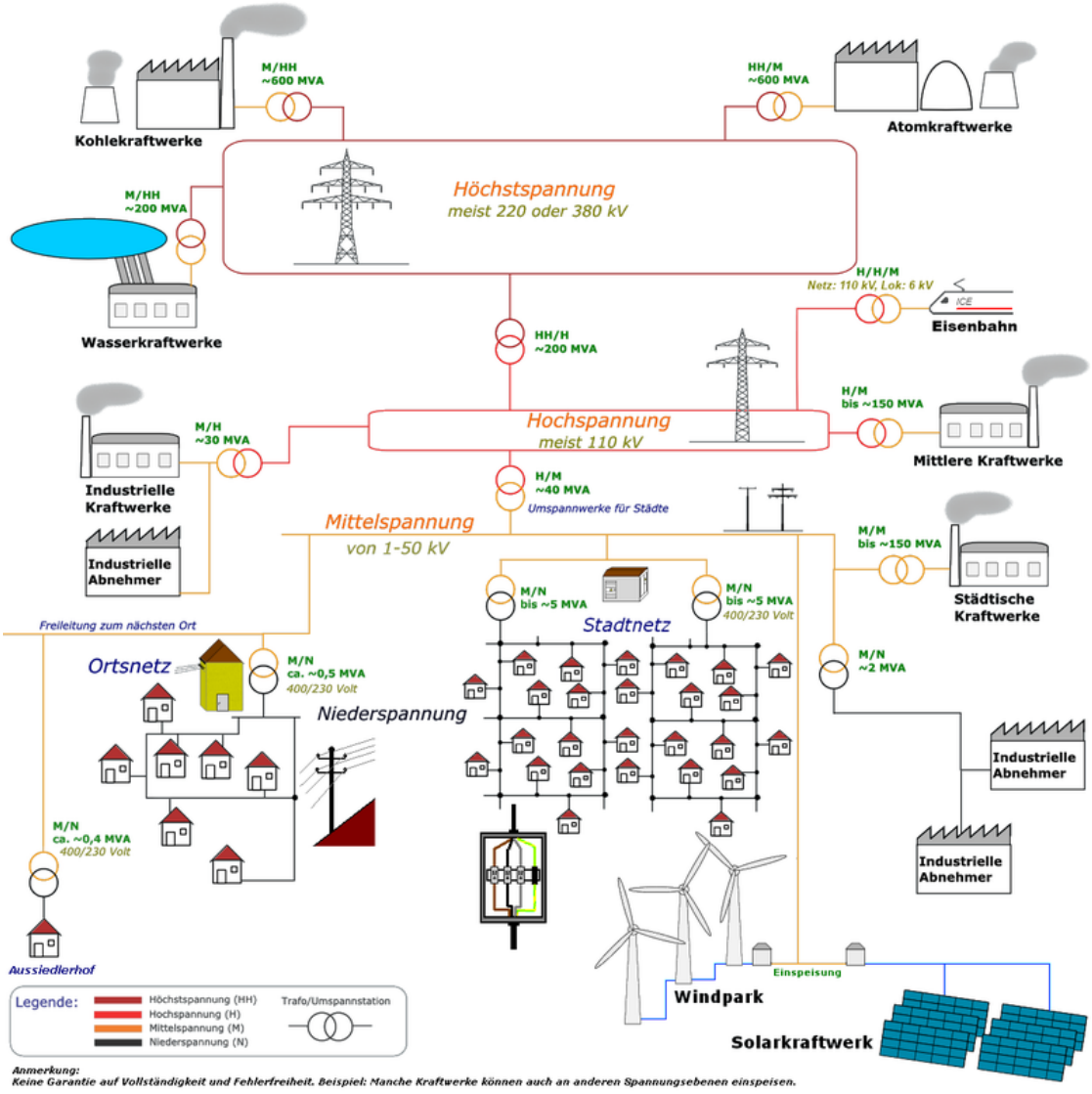


Abbildung 8: Grobe Struktur eines Stromnetzes (wikipedia, Bild: Stefan Riepl, Leon)

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

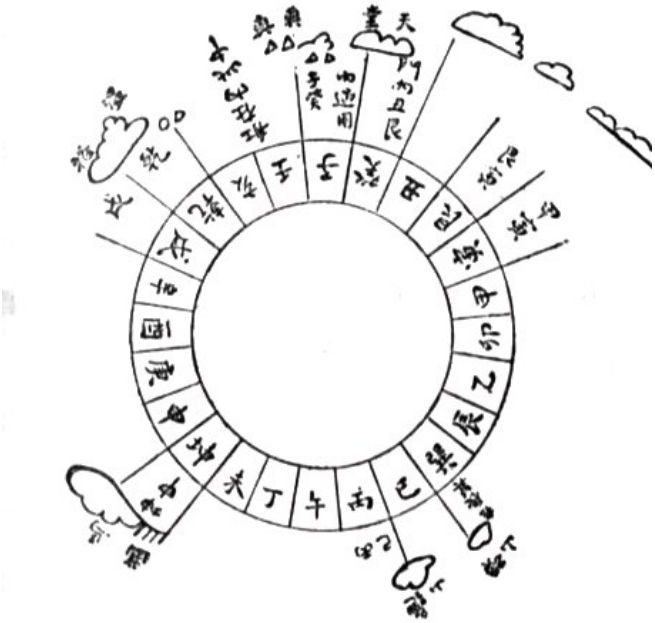
Termine

Donnerstags ab 20:00 Uhr
Forth–Chat net2o `forth@bernd` mit dem Key `keysearch kQusJ`

14.–17. April 2016
Forth–Tagung in Augsburg
`https://tagung.forth-ev.de`

27.–29. Mai 2016:
Maker Faire Hannover
`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.

Einladung zur

Forth–Tagung 2016 vom 14. bis 17. April in Augsburg

Tagung ist in der Hochschule Augsburg, Gebäude J, Friedberger Str. 2A, · 86161 Augsburg, Anfahrt über die Schülestraße. Unterbringung bitte selbst (z.B. über AirBnB) organisieren.

Anreise

Augsburg ist über die Autobahn A8 erreichbar, mit dem ICE und hat nur einen kleinen Flughafen (München benutzen). Vom Hauptbahnhof kommt man mit der Tram 3 oder 6 zur Haltestelle Rotes Tor (unweit der Puppenkiste).

Anmeldung

Auf `http://tagung.forth-ev.de`

Programm

Donnerstag

ab 14:00	Vorträge für Studenten

Freitag

vormittags	Vorträge für Studenten
14:00	Begin der Tagung,
	Vorträge und Workshops

Samstag

vormittags	Vorträge und Workshops
nachmittags	Noch mehr Vorträge und Workshops

Sonntag

09:00	Mitgliederversammlung
12:00	Ende der Tagung