Vierte Dimension

1/2018

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

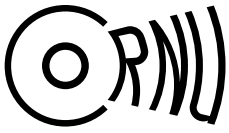
Güstener Straße 72 52428 Jülich
Tel.: 02463/9967-0 Fax: 02463/9967–99
`www.kimaE.de` info@kimaE.de

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.



Cornu GmbH
Ingenieurdienstleistungen
Elektrotechnik

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

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

Hier könnte Ihre Anzeige stehen!

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

`Secretary@forth-ev.de`

Titelbild Autor Text

Quelle

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,

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

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-2018-01

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

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

Liebe Redaktion,

Gibt's auch was anderes als Blindtext????

Generft

Troll

Developing a minimal UDP/IP stack for the N.I.G.E. Machine

Andrew Read

In this article I would like to walk you through the second half of my project to develop Ethernet capability on the N.I.G.E. Machine. The first article in Forth Dimension 2/2016 described the development of the hardware layer; this article describes the development of the software layer. Although this software package, "Minimal IP", has been written especially for the N.I.G.E. Machine, it is ANSI Forth compatible and suitable for any system with the capability to read and write Ethernet frames from a network peripheral. The code is open source [4].

Introduction

We briefly revisit the model Internet protocol stack (fig. 1). Starting from the lowest level, a peripheral integrated circuit on the Digilent Nexys 4 board, a LAN8720 device, provides the functionality of the physical layer, handling the physical transmission of signals over an Ethernet cable, including such matters as appropriate voltages, bit encodings and transmission rates.

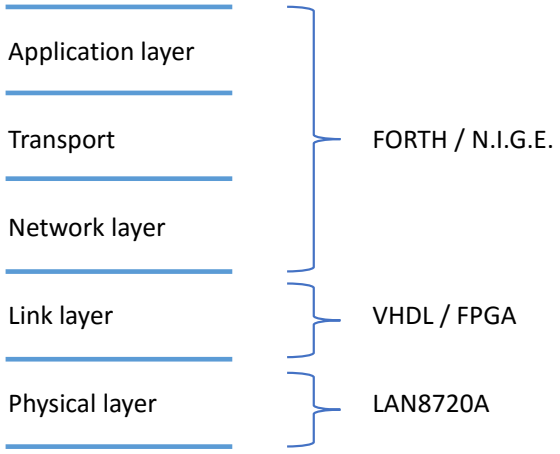


Figure 1: Ethernet technology stack model, an informal representation

A softcore MAC[1] controller, written in VHDL, is incorporated into the design of the N.I.G.E. Machine. The MAC controller handles the next layer in the Internet protocol stack, the "Link" layer, which is responsible for point-to-point communications across a single link, and which also includes the definition of link layer address formats ("MAC addresses"). From the MAC controller, datapackets are passed up the protocol stack to the Network layer which provides IP address functionality, then to the Transport layer that typically offers TCP[2] and/or UDP[3]. Finally the application layer communicates with the network via one or more abstract TCP or UDP ports. Bridging the gap between the MAC addresses of the Link layer and the IP addresses of the Network layer is the ARP[4] (not illustrated in fig. 1).

At the conclusion of the previous article I reported that the hardware MAC controller was only partially working and that there was a problem concerning the early termination of transmitted Ethernet frames. Shortly after writing the article I found the cause: I had incorrectly arranged TXD0 and TXD1 (fig. 11 of the previous article) in the serialization of Ethernet frames. Once the connections from the output shift register to TXD0 and TXD1 had been reversed, the MAC controller became fully functional.

Scope of the system

The Internet protocols taken in general are large and complex by the standards of application software that might typically reside on a small, embedded Forth system. Often such embedded systems requiring Internet connectivity make use of a separate networking module, such as those in the Wiznet range [6], and exchange with it data and simple commands over a serial line. But the spirit of Forth is to simply and solve for only what you need, and so I have limited the scope of this Internet protocol stack to a suitably "minimal" set.

Firstly, I have included within scope UDP but not TCP. TCP is a reliable, byte-stream protocol that performs services such as reordering frames, re-requesting lost frames, and error correction. UDP is a simple, datagram-oriented protocol that offers nothing beyond checksum-based error detection. UDP is much easier and simpler to implement than TCP. To convey some quantification of the difference, in Stevens's classic text book on Internet protocols [1], UDP, which itself has not changed for more than 30

[1] Media Access Control
[2] Transmission Control Protocol
[3] User Datagram Protocol
[4] Address Resolution Protocol
[5] Internet Control Message Protocol

years, is fully described in only 36 pages while TCP is discussed over 225 pages.

Secondly, I have included only what is essential from the IP protocol: IPv6, ICMP[5] and IP fragmentation are all excluded. DHCP[6] is also excluded, with the consequence that the system depends on manually configured, static IP addresses.

Design choices

Client/server or follow-the-data?

As scoped, the system needs to implement three different internet protocols: IP, ARP and UDP. Each protocol can be considered as having input and output sides (table 1). For example, an Ethernet frame is received from the network by the MAC controller and passed to IP as an IP datagram. IP analyses the datagram to determine whether it contains a UDP datagram or some other protocol. If it contains a UDP datagram then that datagram should be passed to UDP for further handling and subsequently to be handed over to the application software. A similar process happens in the reverse order when the application software hands a message to UDP which is then packaged as a UDP datagram and handed to IP, then packaged as an IP datagram and passed to the MAC controller, and finally packaged as an Ethernet frame and passed onto the network.

Two distinct software models present themselves for handling these flows (fig. 2). A client/server model might implement each protocol as a separate process or group of processes. An interprocess communication technique, for example shared memory, would be employed for passing datagrams in an efficient matter between protocols. Each protocol would present no more to the rest of the system than a well-defined interface for accepting datagrams from another protocol. In this approach the separate protocols have been fully abstracted, and the system enjoys the usual benefits of such abstraction, such as independence of implementation at the protocol level.

The client/server model entails certain costs in design and operation: by separating the protocols into separate servers it becomes necessary to create interfaces to those servers. The interfaces must be efficient, so some kind of arrangement must be made to transfer data without copying. And since the individual protocol servers are not necessarily synchronized, buffers must be created at the interfaces to hold pending datagrams. Finally, each process requires a separate execution thread with the attendant process overheads.

Another software model can be characterized as "follow-the-data". In this model a message makes its way from the application through UDP and IP into the MAC controller by a series of function calls. The application calls a function to transmit a block of data in memory as a UDP datagram. That function in turn calls a function to transmit a UDP datagram as an IP datagram, which in turn calls a function to load the IP datagram onto the MAC controller for network transmission. The thread of execution follows the data sequentially from application message through to Ethernet frame. No protocol interfaces are necessary, thus neither are the attendant buffers, and so overheads are reduced.

The principal limitation of this approach is that because of the rigid connection between the protocols and the absence of interposing buffers, it is harder to scale it to systems with higher volumes of datagram communication, especially in a multitasking environment. So the design trade-off is to enjoy simpler code with lower design and execution overheads at the expense of limited scalability. With embedded Forth systems in mind, I have chosen the "follow-the-data" approach.

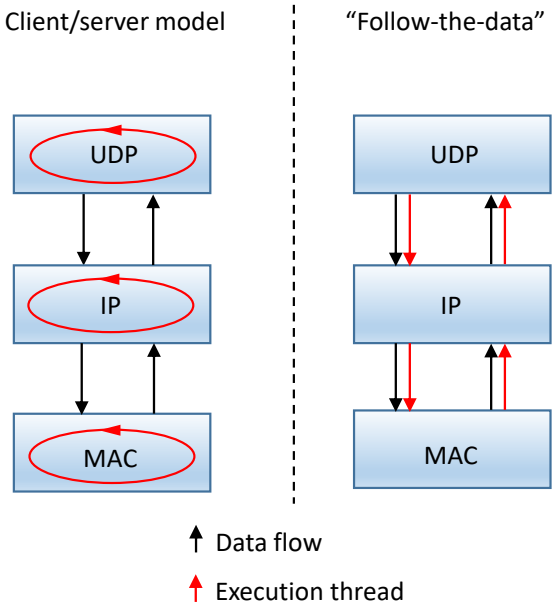


Figure 2: Design choices considered for implementation of the UDP/IP protocol stack

Buffering

Buffering is a consideration because the system may be receiving frames from the network faster than the application can handle them, and vice-versa. Let's quantify the issue. The PHY[7] Ethernet layer operating in "100BASE-TX" mode synchronously transmits and receives data at 100 Mbits/s. An Ethernet frame with an example payload of 80 bytes (say a line of text from an ASCII terminal) will be 160 bytes in length (see fig. 9 of the previous article) with an inter-frame gap of another 12 bytes, in total equivalent to 1,376 bits. The N.I.G.E. Machine is clocked at 100 MHz and has a CPU throughput of just under one instruction per clock cycle (most instructions complete in exactly 1 clock cycle, but branches, calls and returns take 2 or 3 clock cycles). So there will be just over 1,000

[6] Dynamic Host Configuration Protocol

[7] Physical

Protocol	Input side	Output side
IP	Accept an Ethernet frame from the MAC controller	Pass the datagram to UDP or discard
IP	Accept a datagram from UDP	Pass an Ethernet frame to the MAC controller
ARP	Accept an ARP request Ethernet frame from the network	Pass the ARP request response to the network
APR	(As needed by the ARP cache)	Make an ARP request to the network
UDP	Accept a UDP datagram from IP	Pass the message content to the applicaton
UDP	Accept a message for transmission from the application	Pass a UDP datagram to IP

Table 1: Overview of the input and output sides of the three protocols: IP, ARP and UDP

instructions executing in the time it takes that 80 byte message to arrive or depart from the MAC controller, or just over 12 instructions per byte of message. This suggests that buffering is unlikely to be necessary in the transmit direction, but that buffering (or flow control) is highly likely to be necessary in the receive direction.

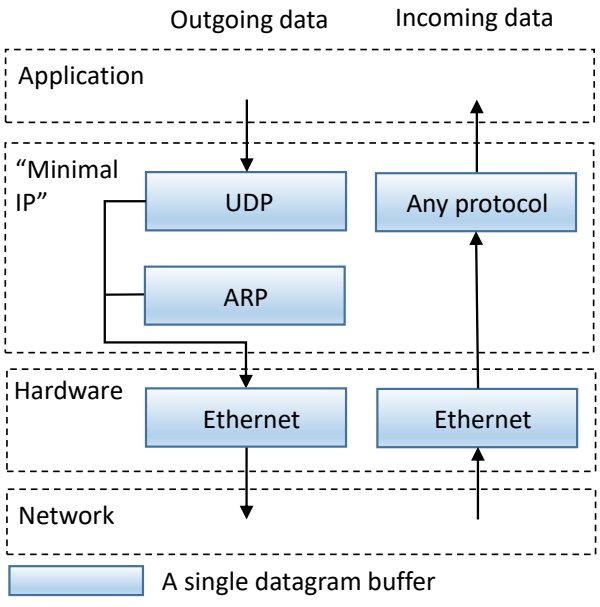


Figure 3: Location of the Ethernet datagram buffers in the hardware and software layers

[8] Application Programming Interface

The current version of the minimal IP package adopts the simplest possible approach to buffering (fig. 3). We assume that the MAC controller is able to buffer at least one received Ethernet frame and one Ethernet frame in transmission. The minimal IP package has a buffer of one frame for any received datagram and and a buffer of one frame each for outgoing ARP and UDP datagrams. Should additional Ethernet frames arrive from the network whilst the processing of an incoming Ethernet frame is taking place, then the MAC controller will be able to buffer one frame but while that buffer remains unread, further frames will be dropped. Should the application software generate messages faster than the MAC controller is able to convey them to the network, which the above analysis suggests is unlikely, then the minimal IP package will block until it has the capacity to accept the next message.

Sockets or a simpler API?

Typical TCP/UDP/IP stack implementation offers an API[8] based on sockets. Sockets are simply virtual ports for reading or writing network data. They are opened with a suitable function call and subsequently referenced by a unique handle (often an integer pointer). Sockets solve the general problem of multiple applications (likely multitasking) needing multiple network access ports. Minimal IP does not offer sockets; instead I have solved only for the simpler problem of a single application needing a single network port.

Why this design choice? Similar considerations apply as for the previous discussion on the client/server model: the socket approach is scalable but requires more initial overhead. The minimal IP approach is a simple and effective starting point, which is an especially relevant consideration on an embedded device that may only need to run a single application. If more capability is required then the code is small and clear enough to be adapted on a case by case basis.

Implementation walk-through

The full listing of Minimal IP [4], with substantial code comments, is slightly less than 600 lines: too long for a complete printing with this article. I would like to walk you through some aspects of the code with edited highlights from the full repository.

Data structures

We reserve space for three Ethernet frames with simple allotments of bytes from the Forth dictionary. There is one incoming frame of any type and one each outgoing ARP frame and UDP frame (Listing 1 - data_structures.f). A maximum payload of 1,500 bytes is supported. Combined with the "follow-the-data" implementation, this approach is fast because there is no hand-off of data between protocols - each protocol accesses its own datagram within the overall Ethernet frame with fetch and store instructions at the appropriate byte offsets.

We store MAC addresses and IP addresses in memory in byte-by-byte format with the most significant byte in the lowest memory address. This is network endian format, consistent with how MAC and IP addresses appear within Ethernet frames.

MAC controller interface

Minimal IP relies on two functions being available to receive from and transmit frames to the hardware MAC controller: `getFrame` and `putFrame`. This is necessarily machine dependent, and the code for the N.I.G.E. Machine is shown in Listing 2 - netNIGE.f.

The API

The API requires a multitasking Forth system and two processes: one for the transmission of datagrams and one for the reception of datagrams.

Firstly the transmit process: typically the transmit process is the process in which the main thread of the application software is executing. An application wishing to send a UDP datagram over the network first confirms the destination IP address, UDP destination port and UDP source port by calling `setUDPdestIP` and storing the word-length integer ports numbers (network endian format) in the variables `UDPdestPort` and `UDPsrcPort` (Listing 3 - API.f). The send is initiated by providing the address and number of bytes of the relevant data buffer on the parameter stack and calling `UDP.send`. This function firstly completes the necessary fields of the UDP datagram, then computes the UDP checksum (done in parallel with copying the data from the buffer into the UDP datagram, for efficiency), before finally calling IP to dispatch the datagram.

Incoming datagrams from the network are handled in a super-loop running in a second, dedicated process. This is the word `EthernetReceiveTask`. Once incoming UDP frames emerge from the top of the protocol stack they are "hardwired" to the Forth word `UDP.in`, which is called from the process receiving incoming frames. Each application should rewrite `UDP.in` to handle incoming UDP data as needed. As presently written, `UDP.in` simply types the incoming UDP data to the screen.

As an alternative to this dual process approach, the code could likely be adapted around interrupts on systems where multitasking is not available. (I chose the dual process approach because multitasking on the N.I.G.E. Machine is built-in and supported by hardware),

Checksums

An Ethernet frame carrying a UDP datagram will contain three error-check fields. Firstly there is an Internet checksum calculated over the UDP header and UDP payload. (The checksum is actually calculated over a UDP pseudo-header that includes parts of the IP header). Secondly there is another Internet checksum calculated over the 20 byte IP header, and lastly there are 4 CRC[9] bytes at the end of the Ethernet frame. The CRC bytes of the Ethernet frame are calculated in hardware by the MAC controller for both incoming and outgoing Ethernet frames, as described in the previous article. The Internet checksums much be calculated in software by the IP and UDP protocols.

The Internet checksum is a 16-bit mathematical sum used to determine, with reasonably high probability, whether a message that has been received matches the original message that was sent [1]. The Internet checksum proceeds as follows. The 16-bit one's complement sum is computed over the data range in question by considering it to be a sequence of 16-bit words. In both cases (IP checksum and UDP checksum), this range includes the checksum field itself, which is initialized to zero before the checksum calculation begins. After the calculation the checksum is stored in the checksum field.

The Internet checksum method is not an arbitrary procedure. It is based on the mathematics of Group Theory and is chosen, among other reasons, for the following remarkable property: when the Internet checksum algorithm is calculated for a second time over the requisite data range including the checksum field, then if there have been no changes to the data (and no pairs of compensating errors) then the checksum calculated this second

[9] cyclic redundancy check

time will be equal to zero. Hence we have a simple error detection indicator: if the checksum of an incoming datagram does not equal zero then it should be discarded. An explanation of alternative checksum algorithms and further background on the Internet checksum itself is given in Plummer's original 1978 paper [8].

For performance reasons the Internet checksum should be as computationally efficient as possible, yet as described it relies on 16-bit one's complement arithmetic, which is not a commonly available CPU instruction. Braden et al. [7] provide a C algorithm for efficient calculation of the Internet checksum using 32-bit addition with carry (which compiles to a single machine language operation, standard on all 32-bit CPU's). Their method relies on calculating a 32-bit two's complement sum over the data range, and then converting it into a 16-bit one's complement sum. This last step the authors refer to as "folding" the accumulated carry in the higher order 16-bits into the lower order 16-bits. Their method also has the advantage that the initial 32-bit sum can be calculated in either big or little endian format, thus utilizing the natural machine language instructions of the system.

Braden's C algorithm is presented in Listing 4 - checksum.f, along with the directly ported Forth code.

ARP

The introduction briefly alluded to the ARP that bridges between the IP addresses of the network layer and the MAC addresses of the LINK layer. It works like this: when IP is ready to send an IP datagram to the MAC controller it first asks ARP for the destination MAC address corresponding to the destination IP address. ARP operates by maintaining a cache of IP addresses and their corresponding MAC addresses. In case the ARP cache does not contain a valid entry for the IP address in question, it will issue an ARP request onto the network.

An ARP request is a broadcast transmission that will be accepted and processed by any MAC controller on the local network. Should there be another device on the local network that holds the requested IP address, then that device will make an ARP reply to inform the requester of its MAC address.

For IP addresses that reference destinations beyond the local network (the "subnet"), the ARP cache substitutes the MAC address of the local router that will forward the IP datagram from the subnet to Internet. Also note that according to specification, entries in the ARP cache become invalid after 20 minutes.

Listing 5 - ARP.f, illustrates the ARP functionality of minimal IP in abbreviated form. Note that as well as providing IP with necessary MAC addresses, APR also needs to respond to incoming ARP requests, which may occur at any time. Since minimal IP handles the sending and receiving of Ethernet frames in separate processes, exclusive access to ARP is mediated by the acquisition and release of a semaphore.

Communication between protocols

The prior section on design decisions explained how a follow-the-data model is used for the implementation of the protocol stack. This is illustrated in Listing 6 - stack.f. The word `Ethernet.in` is called by the super-loop of the incoming frame process (see Listing 3 - API.f) when an Ethernet frame has been received by the MAC controller and is ready for processing.

The first step is to check that its destination MAC address either matches the MAC address of our device, or is the broadcast MAC address. The next step is to separate frame according to the datagram type: ARP and IP frames are passed to these protocols while other frames are silently dropped.

IP first checks the frame to make sure that it meets the requisite assumptions concerning checksum, expected specification and destination IP address. This implementation of IP only supports UDP datagrams and so other datagram types are dropped. UDP makes its own checksum confirmation before handling the datagram in whichever way the application requires.

Development and testing

Figure 4 shows the development and testing apparatus. Two Digilent Nexys 4 boards were directly connected with an Ethernet crossover cable. A home-made Ethernet loopback connector was also employed and is illustrated. Communication between the two board was established in steps whilst interactively developing the minimal IP Forth code. This approach proved to be very straightforward and efficient. At the final stage, Ethernet connection with a PC was tested with Wireshark.

As well as the previously mentioned TCP/IP Illustrated by Stevens [1], the papers by Baden [7] and Plummer [8], Tanebaum's Computer Networks [2] and Waren's Hacker's Delight [3] were essential resources in the development of Ethernet capability on the N.I.G.E. Machine.

[H]

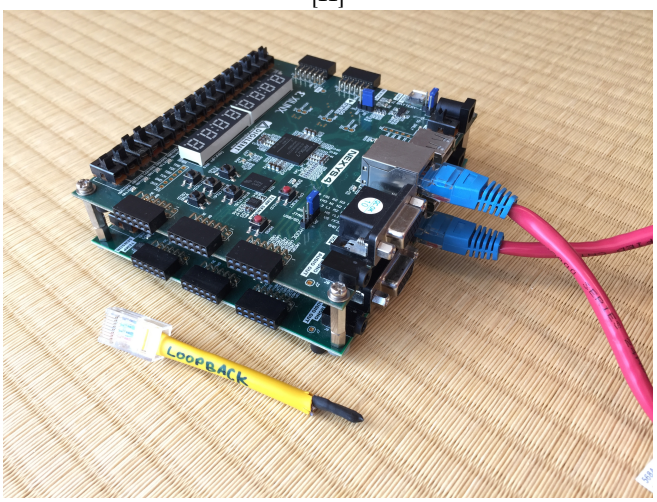


Figure 4: Digilent Nexys 4 dual development arrangement

Conclusions

Reconsidering the design decisions in the light of the finished minimal IP package, I believe the "follow-the-data" model, rather than attempting to implement a client/server model in an embedded Forth environment, was optimal. The choice of having only minimal buffers was more of an expedient to simplify the initial design. It would likely not be complicated or difficult to arrange greater buffering of incoming frames, although some further decisions would be required as to whether those buffers should be at the IP level, the UDP level, or the application level. Given that some frames are destined to be dropped anyway (unsupported protocol, etc.) buffering after initial processing by the IP might be most appropriate. Considering this point further, it would be great efficiency to push the filtering of incoming Ethernet frames based on destination MAC address to the hardware level, since in that case non-matching frames could be discarded without being buffered. (As presently configured, the protocol stack is effectively in the "promiscuous mode" of a network interface controller.)

I believe that DHCP would be a worthwhile extension, thus allowing the convenience of automatic IP address configuration when networking with any modern router. If necessary, a "socket-like" API could be added to enable multitasking applications similtaneously to take advantage of Ethernet connectivity. Lastly, whilst the accurate implementing TCP has the reputation of being formidably difficult, I am curious as to whether a simplified approach could be taken. The idea would be to exclude difficult "corner cases" in a fail-safe manner, accepting lower reliability as a consequence. This would be a substantial project for when time allows!

I hope this article has been of interest and I welcome any correspondence or suggestions. All of the N.I.G.E. Machine source code is open source [5].

andrew81244@outlook.com

Referenzen

[1] "TCP/IP Illustrated", Kevin R. Fall and W. Richard Stevens, 2nd edition revised, 2011

[2] "Computer Networks", Andrew S. Tananbaum and David J. Weatherall, 5th edition, 2010

[3] "Hacker's Delight", Henry S. Waren, 2nd revised edition, 2012

[4] Github, https://github.com/Anding/Minimal_IP

[5] Github, https://github.com/Anding/N.I.G.E.-Machine

[6] Wiznet, http://www.wiznet.co.kr

[7] Computing the Internet checksum, Braden et al., 1988, https://tools.ietf.org/html/rfc1071

[8] TCP Checksum Function Design, Plummer, 1978, appendix to https://tools.ietf.org/html/rfc1071

[9] Wireshark, https://www.wireshark.org

Das Perceptron

M. Kalus

Als Student beschäftigte mich die Frage, wie es möglich ist, dass so ein Haufen Draht mit mir 'redet' - und habe die Frage mit Hilfe von Forth geklärt, und mit Hilfe all derer, die ich in der Forth-Gesellschaft getroffen habe. Jüngst stolperte ich über neuronale Netze und 'deep learning'. Neugierig geworden, musste geklärt werden, wie das geht, so im Prinzip jedenfalls. Und auch dabei war mir Forth wieder nützlich. Gforth in der GNU-bash auf Linux Mint leistete gute Dienste dabei.

Das Perceptron geht zurück auf Frank Rosenblatt, amerikanischer Psychologe und Informatiker. Er war 32 Jahre alt, als er 1957 das Konzept entwickelte, doch erst 1960 konnte es an der Cornell University als Mark–1–Maschine umgesetzt werden. Was das für ein mühseliger Akt gewesen sein muss, lässt sich ahnen, schaut man sich so ein Ding mal an — Abb.5 — handverdrahtet! Da hat man es auf seinem Laptop heutzutage doch sehr viel bequemer.

Geschichte

Um das mal historisch einzuordnen: Der Transistor wurde zwar schon 1925 von Julius Edgar Lilienfeld beschrieben und zum Patent angemeldet, doch erst 1947 kamen Bardeen und Brattain bei AT&T's Bell Labs in Murray Hill, New Jersey, USA, dahinter wie es richtig produktiv geht. Und ab 1954 gab es das erste Transistorradio, das *Regency TR–1*, ein Mittelwellen-Superhet mit vier (!) Transistoren. Bald darauf folgten die Japaner, Sony, mit dem *shirt-pocket radio*, und aus dem asiatischen Raum drängten schon billige 2–Transistor–Reflexempfänger nach. Die Miniaturisierung nahm ihren Lauf. Integrierte Schaltkreise meldete Robert Noyce 1959 zum Patent an, noch wenige Transistoren, aber nun schon auf einem Substrat. Fairchild Semiconductor kam zu der Zeit gerade mit dem Diffusionsverfahren für die Herstellung des Bipolartransistors heraus, und es hat noch bis 1970/71 gedauert, bis die ersten Mikroprozessoren von drei Firmen fast zeitgleich vorgestellt worden sind: Der *Intel 4004*, der Texas Instruments (TI) *TMS 1000* und der Garrett AiResearch *Central Air Data Computer* (CADC).[1] In der Biologie hatte sich um die Jahrhunderwende herum die Zelltheorie durchgesetzt, und auch die Neurobiologen mussten erkennen, dass selbst Nerven und Gehirne zellulär aufgebaut seien. Um 1925 herum beschrieb Camillo Golgi das Nervengewebe als ein *Reticulum*, in dem die Zellkörper, die Neuronen, durch ihre Fortsätze kontinuierlich untereinander verbunden sind — das neuronale Netz. Psychologen waren fasziniert davon, und sind es noch heute. Und über das Hirn, seine Neuronen und über ein *Artificial Neuronal Network* (ANN) wird seither wild spekuliert. Und die ANNs werden nun massiv simuliert, um zu erforschen, wie sie denn wohl denken und lernen könnten.

Die Überlegungen Rosenblatts, das Perceptron in einem Rechner zu simulieren, lag also sozusagen in der Luft zu seiner Zeit, und IBM zog mit. So wie heute Google mit ihren gigantischen Rechenzentren und Nvidia mit ihren Multiprozessor–Grafikkarten mitziehen, wenn es um die Simulation der ANNs geht. Da sind nun Cluster aus Rechnern mit mehrkernigen CPUs im Gigahertz–Takt am Werk, wovon jeder gleich mehrere Grafikkarten beschickt, die wiederum GPUs mit 3.840 Recheneinheiten bei 1,6 GHz haben.[2] Alles nur um deep learning zu simulieren. Da wird also inzwischen tatsächlich sehr sehr viel gerechnet und der Ausdruck *Rechner* ist da wirklich mal angebracht.

Simulation eines Perceptrons

Zurück zu den Anfängen. Anders als eine digitale Schaltung, reagierte das Perceptron auf analoge Eingaben und prozessierte diese auch analog.

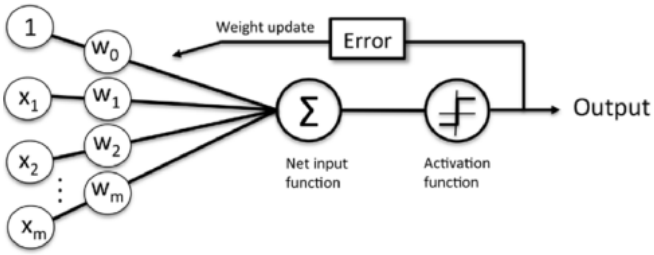


Abbildung 1: Schema des Perceptrons

Die hier vorgeführte Simulation eines Perceptrons in gForth macht das mit Gleitkommazahlen nach. Realisiert sind zwei Eingänge `X1` und `X2`, sowie ein Bias `X0`, die gewichtete[3] Summenbildung und eine Aktivierungsfunktion, die daraus die Ausgabe macht. Die ganze Propagation durchs Perceptron lautet also:

```
: propagate ( in1 in2 -- )
  in sum activate to out0 ;
```

Der Bias `x0` wirkt dabei wie ein weiterer Eingang mit dem konstanten Wert +1, der auch gewichtet und mitsummiert wird. `IN` nimmt darin die Eingabewerte an, `SUM` bildet die gewichtete Summe dieser realen Zahlen, und `ACTIVATE` reagiert darauf mit einem Schwellenwert als Ausgabe.

Natürlich müssen die Gewichte `W0 ... W1` vorher schon passend initialisiert worden sein, damit das Perceptron

[1] Quelle: Wikipedia

[2] NVIDIA TITAN Xp; 2018

[3] Rosenblatt hat die Multiplikatoren der Eingangswerte *weights*, Gewichte, genannt, und den Vorgang, das durchzurechenen, `propagate`.

eine sinnvolle Reaktion zeigen kann. Wenn die Aktivierungsfunktion nur eine einfache Sprungfunktion ist, die bei `OUT<=0` eine `fNULL`[4] ausgibt, und sonst eine `fEINS`, erhält man eine AND-Funktion bereits mit solchen Gewichten wie:

```
W0 = -0.5
W1 = +0.5
W2 = +0.5
```

Rechnen wir mal nach:

```
SUM = X0*W0 + X1*W1 + X2*W2
```

Für die Eingabe `X0=X2=1.0` erhält man damit:

```
(1.0*-0.5)+(1.0*0.5)+(1.0*0.5)=1-0.5=0.5
```

also einen Wert größer als null und damit eine `fEINS` in `OUT`. Wird jedoch ein X-Wert null, oder gar beide, bleibt die Summe null, oder wird sogar kleiner als null.

Durch geschickte Verteilung der Gewichte kann das simple einlagige Perceptron somit bereits einfache logische Muster an seinen Eingängen erkennen. AND oder OR oder NOR sind damit möglich - siehe Listing: AND-Perceptron manuell definieren.

Lernen

Interessant wird das Ganze aber erst dadurch, dass die Gewichte im Perceptron nicht vorbestimmt sein müssen, sondern erlernt werden können. Dabei wird umgekehrt vorgegangen, also vom Ausgang des Perceptrons zu den Eingängen zurück gerechnet.[5] Und da man nicht wissen kann, wie die Gewichte auf den Pfaden tatsächlich sein müssen, wird erstmal irgend eine Annahme gemacht. Danach wird in großen Netzen viel Wahrscheinlichkeitsrechnung[6] betrieben. Im einlagigen Perceptron genügt eine einfache Fehlerrechnung. Man ermittelt nach der Propagation schlicht, wie weit die erzielte Ausgabe vom gewünschten Wert abweicht. Und errechnet damit eine Korrektur für jedes der Eingangsgewichte, indem die Abweichung noch mit einem Korrekturfaktor, der Lernrate, versehen wird. Das wird so lange wiederholt, bis die Muster gelernt worden sind.

Die Formel zur Korrektur der Gewichte lautet:

```
Wi(neu) = Wi(alt) + dWi
dWi = (soll - out) * alfa * Xi
```

`Wi` sind die einzelnen Gewichte, `dWi` die jeweilige Differenz, die dann noch multipliziert mit der Lernrate `alfa` und dem aktuellen jeweiligen Inputwert `Xi` den Korrekturwert ergibt.

Im AND-Beispiel würde also immer nur derjenige Eingang neu gewichtet, dessen `Xi` gerade `fEINS` ist, der Bias aber immer. So wird mal dieses und mal jenes Gewicht etwas verändert. Solange die Muster eindeutig unterscheidbar sind in richtig oder falsch, konvergiert dieses Verfahren immer zu einer Lösung, lernt also die Aufgabe durch Versuch und Irrtum. Und weil dabei ja die gewünschten Ergebnisse von vornherein bekannt sind, also das Ziel klar ist, wie bei einer Dressur, nennt man das *supervised learning*.

Für die Initialisierung hat es sich bewährt, mit kleinen Gewichten `Wi` an den Eingängen zu beginnen. Daher habe ich eine Random-Funktion `INITWR` für die Initialisierung gemacht. Man kann aber auch andere Initialisierungen verwenden.

Nun üben wir AND

Mit dem Perceptron wird solange geübt, bis alle vier möglichen Eingangsmuster für ein AND, das *batch* aus `P11`, `P01`, `P10` und `P00`, richtig erkannt werden. Je Episode wird nacheinander jedes Muster präsentiert, immer zusammen mit seinem Sollwert, dann propagiert und backpropagiert. Diese Lern-Episoden wiederholt man solange, bis das Ergebnis stimmt. Das ist der Fall, sobald sich die Gewichte nicht mehr ändern - *early stopping*. Ich habe allerdings der Einfachheit halber keinen Stop eingebaut, denn so eine einfache Lernaufgabe, wie die logischen Verknüpfungen zweier binärer Eingänge, ist bereits nach wenigen Schritten und damit sowieso im Handumdrehen erledigt. Mit dem Kommando:

```
AND! 10 EPOCHEN
```

kann man schön nachverfolgen, wie die Gewichte verändert wurden.

Ergebnisse

Zum Schluss als Illustration des Ganzen noch einige typische Lernkurven. Die sind aus den Logs von je 10 Episoden `AND`, `OR` und `NAND` entstanden, welche dann zur grafischen Darstellung in eine Tabellenkalkulation[7] übernommen worden sind. Man sieht sehr schön, wie die Gewichte sich zunächst noch veränderten und schließlich unverändert blieben. Da hatte das Perceptron dann schon gelernt, was es sollte.

[4] Das `f` zeigt an, dass es sich um eine *floating point number* handelt.

[5] Dafür prägte Rosenblatt den Begriff *backpropagation*.

[6] *Stochastic gradient descent* (SGD).

[7] LibreOffice Calc

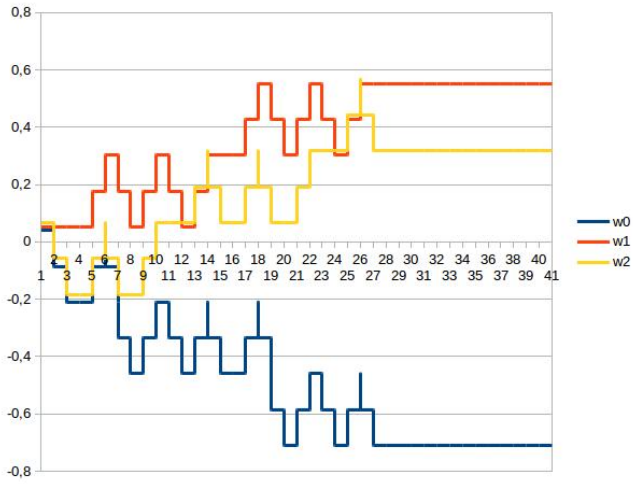


Abbildung 2: AND–Lernkurve

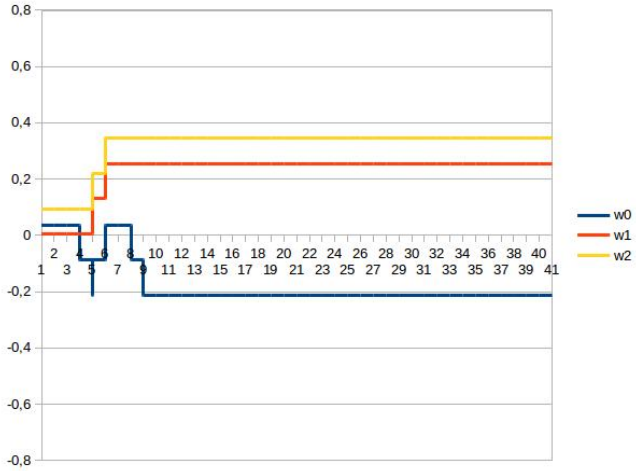


Abbildung 3: OR–Lernkurve

Listing

```
1   \ Perceptron nach Frank Rosenblatt
2   \ M.Kalus 12/2017
3
4   vocabulary perceptron    perceptron definitions
5   decimal
6
7
8   \ In den Werkzeugkasten kommt vorweg:
9   \ ---------------------------------------
10
11  : .. bye ;  \ easy typing exit from gforth
12
13  \ floating point numbers
14  6 set-precision
15  : .fs  f.s .s ; \ check both stacks
16
17  0.0e fconstant fnull
18  1.0e fconstant feins
19
```

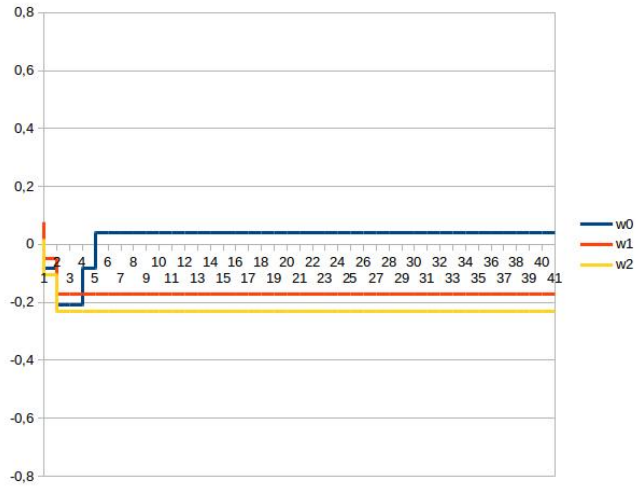


Abbildung 4: NAND–Lernkurve

Und macht man mehrere Durchläufe derselben Lektion, sieht man auch, dass sich immer wieder etwas andere Initialisierungs-Gewichte ergeben. Womit dann jedesmal ein etwas anders gewichtetes Perceptron entsteht.[8] Da haben wir also schon im Ansatz das, was heute *Neurodiversität* genannt wird - kein Gehirn gleicht einem anderen, selbst wenn es nach dem gleichen Bauplan geschaffen worden ist.

Anmerkung

Wie loggt man in gforth? Indem die Konsolenausgabe in eine Datei umgelenkt wird. Eine Kommandozeile der Art:

```
gforth perceptron.4th > trainingsdaten.txt
```

sollte es tun. Aktiviert man noch das `BYE` am Ende des Perceptron-Codes, ist das alles sofort erledigt, ohne im gforth zu verbleiben.

Links

https://en.wikipedia.org/wiki/Perceptron

```
20  \ Eingabemuster für beide Eingänge des Perceptrons
21  : p11 ( -- x1 x2 ) feins feins ;
22  : p01 ( -- x1 x2 ) fnull feins ;
23  : p10 ( -- x1 x2 ) feins fnull ;
24  : p00 ( -- x1 x2 ) fnull fnull ;
25
26  \ floating point random number generator
27  Variable seed    time&date + + + + + seed !
28  $10450405 Constant generator
29  : rnd  ( -- n )
30    seed @ generator um* drop 1+ dup seed ! ;
31  : random  ( n -- 0..n-1 )  rnd um* nip ;
32  : frandom ( n -- r )       random s>d d>f ;
33
34
35  \ Simulation des Perceptrons (Propagation)
36  \ ---------------------------------------
37
38  \ * Zustände definieren
39
40  \ 2 Eingänge
```

[8] Das hab ich hier nicht mehr vorgeführt, probiert es mal selbst.

```
fnull fvalue x1
fnull fvalue x2

\ 1 Bias
\ (ist ein Eingang mit konstantem Wert)
feins fconstant x0

\ Eingangsgewichte (weights)
fnull fvalue w0
fnull fvalue w1
fnull fvalue w2

\ Lernrate
0.25e fvalue alfa

\ Ausgegebner Wert
fnull fvalue out0   ( output )

\ * Zustände anzeigen

: .xi     x1 f.  x2 f. ;
: .wi     w0 f.  w1 f.  w2 f. ;
: .out0   out0 f.  ;
: .alfa   alfa f.  ;

\ * Funktionen definieren

\ Gewichte random initialisieren
: initwr ( -- )
  1000 frandom 1.0e-4 f*  to w0
  1000 frandom 1.0e-4 f*  to w1
  1000 frandom 1.0e-4 f*  to w2  ;

\ Eingangswerte einlesen
: in ( x1 x2 -- )    to x2  to x1 ;

\ Eingangswerte gewichtet summieren
: sum ( -- sum )
  x0 w0 f*
  x1 w1 f*
  x2 w2 f*
  f+ f+ ;

\ Perceptron aktivieren
  defer activate
: sprung ( sum -- out )
  f0<= if fnull else feins then ;
' sprung is activate

\ Propagation durchs Perceptron
: propagate ( in1 in2 -- )
  in sum activate to out0 ;




\ AND-Perceptron manuell definieren
\ ---------------------------------

\ * Gewichte für ein AND einstellen

: initmw ( -- )
  -0.5e to w0
  +0.5e to w1
  +0.5e to w2  ;

\ Perceptron testen
: .xo    ( -- )  cr .xi .out0  ;
: .mwand ( -- )
  initmw
  p11 propagate .xo
  p01 propagate .xo
  p10 propagate .xo
  p00 propagate .xo ;



\ Perceptron trainieren (Backpropagation)
\ ---------------------------------------

\ * Trainings-Zustände definieren

defer initw
' initwr is initw

\ Durchlaufnummer
0 value num

\ Bewertungs-Kategorien
' feins alias gut
' fnull alias nix

fnull fvalue soll0  \ Sollzustand
fnull fvalue err    \ Fehlerwert

\ * Zustände anzeigen

: .soll0    soll0 f. ;
: .err      err f. ;
: .num      num . ;

: .log     cr .num .xi .soll0 .out0 .err .wi ;
: .namen   cr ." num x1 x2 soll out err w0 w1 w2 " ;

: header   initw .namen .log ;

\ * Funktionen definieren

\ Formel zur Korrektur der Gewichte
\   wi(neu) = wi(alt) + dwi
\   dwi = (soll - out) * alfa * xi

\ Fehler berechnen
: dwi  ( xi -- err )
  soll0 out0 f- alfa f* f*
  fdup to err ; \ aufheben für .err

: backpropagate  ( -- )
  x0 dwi  w0 f+  to w0
  x1 dwi  w1 f+  to w1
  x2 dwi  w2 f+  to w2 ;

\ * Trainingsablauf

: ueben ( in1 in2 soll -- )
  to soll0   propagate
  backpropagate  .log ;

\ AND lernen
: and-batch ( -- )
  p11 gut ueben
  p01 nix ueben
  p10 nix ueben
  p00 nix ueben ;

: and-lernen ( n -- )
  ." AND" header
  0 do   i to num
  and-batch loop ;

\ OR lernen
: or-batch ( -- )
  p11 gut ueben
  p01 gut ueben
  p10 gut ueben
  p00 nix ueben ;

: or-lernen ( n -- )
  ." OR" header
  0 do   i to num
  or-batch loop ;

\ NAND lernen
```

```
193  : nand-batch ( -- )
194    p11 nix ueben
195    p01 nix ueben
196    p10 nix ueben
197    p00 gut ueben ;
198
199  : nand-lernen ( n -- )
200    ." NAND" header
201    0 do   i to num
202    nand-batch loop ;
203
204  \ * Trainingsepochen
205
206  defer epochen ( n -- )
207
208  : and!     [']  and-lernen is epochen ;
209  : or!      [']   or-lernen is epochen ;
210  : nand!    [’] nand-lernen is epochen ;
211
212  cr .( training )
213
214  cr  and!   10 epochen
215  cr  or!    10 epochen
216  cr  nand!  10 epochen
217
218  \ bye
219
220  cr cr words
221
222  \ logen mit der Kommandozeile in der bash :
223  \ gforth perceptron.txt > trainingsdaten.txt
224  cr cr .( finis)
225  cr cr
226
```

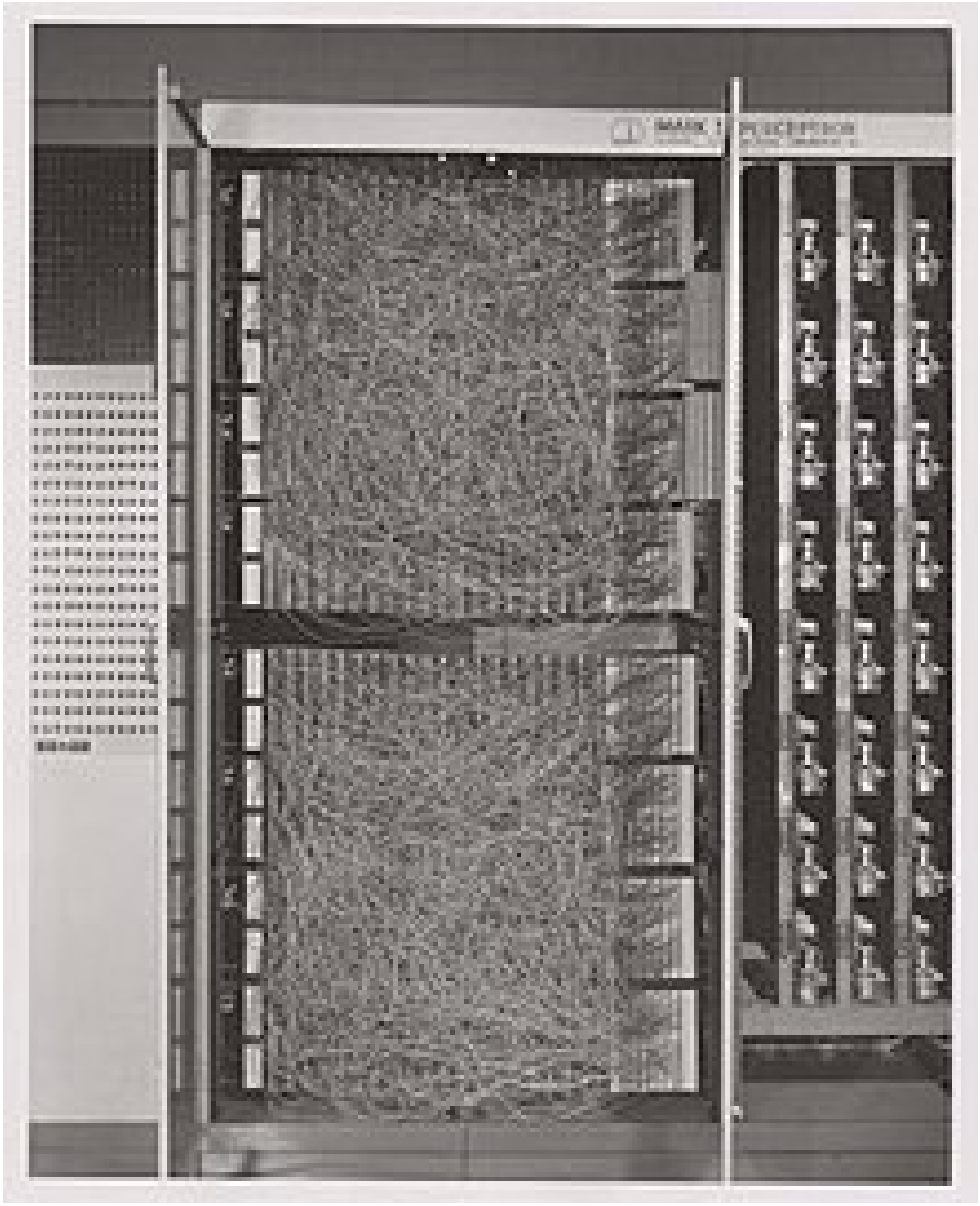


Abbildung 5: Der Mark-1

Morse 6 — eine Variante

Erich Wälde, Klaus Schleisiek

Neulich auf dem 34C3 kommt Klaus vorbei, und wir faseln hin und her über Anfänger und Forth und wie toll man bei Forth doch dem Compiler zur Laufzeit neue Kunststücke beibringen kann. „Ach“, sagt Klaus, „hattest Du nicht sowas Erweitertes mit dem Morse–Dings? Kann ich das geschwind für den Vortrag haben?“ Klar konnte er das haben. Und jetzt kommt er daher mit einer vereinfachten Version von Morse 5 [2]. Dann wollen wir das mal vorstellen.

Wir erinnern uns: Stephen Pelc hatte uns auf den Weg mitgegeben, *Notation matters!* [1]. Deswegen hatte ich mir einen Weg überlegt, wie man Zeichenfolgen dieser Bauart

```
. . . _  |   v
```

automatisch auswerten könnte, um daraus die Länge des Morsecodes (4) und seinen Wert (0b1000) automatisch zu generieren [2]. Damals habe ich das so gelöst, dass ich zwei Variablen definierte, und drei Funktionen, die das Muster auswerteten:

```
variable morse.tmp.length
variable morse.tmp.code
: _
  morse.tmp.code @
  \ place 1 at position length
  1 morse.tmp.length @ lshift  or
  morse.tmp.code !
  length++ ;
: .
  length++
;
: |
  \ produce packed value
  morse.tmp.length @ >< morse.tmp.code @ or
  \ reset tmp vars
  /m
  \ eat next char in table definition
  char
  \ use as offset and store
  mtable + !i
;
```

Die notwendige Buchhaltung und die separate Wortliste habe ich jetzt mal kurz unterschlagen. Mit anderen Worten: `.` und `_` werden in einer neuen Wortliste (`<morse>`) definiert. Beide erhöhen die Länge des Musters, und `_` platziert eine 1 an der richtigen Position in `morse.tmp.code`. `|` packt die beiden Werte zusammen, konsumiert das nächste Zeichen als Index in eine Tabelle und schreibt den gepackten Wert dort hinein. Alles bestens? Schon, aber es geht natürlich auch anders.

Magische Werte

Im gepackten Morsecode steht eine 1 für einen Strich, und eine 0 für einen Punkt. Wir könnten also auch irgendwie die Werte

```
0 0 0 1 4
```

auf den Stapel zaubern. Das wäre die Information für das Zeichen `v`. Mitzählen braucht aber die besagte Variable von oben, ist also nicht ganz Forthsch, denn da sagt man ja: nimm den Stapel. Also anderer Vorschlag, wir sammeln die Punkte und Striche und legen darunter (also zuerst) einen Wert ab (`-1`), der das Ende der Liste markiert.

```
-1 0 0 0 1
```

würde funktionieren. Wir verarbeiten also die Werte (in einer Schleife) bis wir auf eine `-1` treffen, und brechen dann die Schleife ab. Hmmm, aber die Schleife (`begin ... while ... repeat`) lässt sich besser mit `false`, also `0`, abbrechen anstatt mit `true` bzw. `-1`. Andererseits haben wir die Null schon in Benutzung. Aber egal, dann nehmen wir eben `1` und `2`, statt `0` und `1`. Kann uns ja keiner verbieten, mal eben mutwillig eine `1` zu addieren. Also

```
0 1 1 1 2
```

Die Information ist immer noch da: von oben, im Stapel gesehen, bekommen wir ein Mal Strich (`2`) und drei Mal Punkt (`1`) und dann das Ende des Musters mit der Null. Das Muster sieht rückwärts aus. Wenn wir die zugehörigen Bits in einen Wert hineinrotieren (`lshift`) und beim Lesen in umgekehrter Richtung wieder raus, dann kommen zuerst die 3 Punkte und dann der Strich — alles wird gut.

Auf den Stapel!

Mit diesen Ersetzungen im Plan wird die Angelegenheit lösbar. Punkt und Strich mutieren zu simplen Konstanten, welche bekanntlich zur Laufzeit ihren Wert auf den Stapel legen.

```
1 constant .
2 constant _
```

Die Funktion `|` wird jetzt vielleicht etwas aufwändiger, weil wir die Anfangswerte für `count` und `morsebits` (jeweils `0`) noch zusätzlich oben auf den Stapel schieben.

```
\ produce packed value, store in morsetable
: |  ( 0 n1 .. nr -- 0 )
                    \ example:  . - | a
                    \ -- 0 1 2
  0  ( count )      \ -- 0 1 2 cnt
  0  ( morsebits )  \ -- 0 1 2 cnt morsebits
  begin
    rot             \ -- 0 1 cnt morsebits 2
    ?dup while      \
      1-            \ -- 0 1 cnt morsebits 1
      swap 2* or    \ -- 0 1 cnt morsebits'
      swap 1+       \ -- 0 1 morsebits' cnt+1
      swap          \ -- 0 1 cnt' morsebits'
  repeat            \ -- count morsebits
```

```
                  \ count  in bits 10..8
                  \ morsebits in    7..0
  swap #8 lshift or \ -- packed-code
  mtable          \ -- code addr[0]
  char            \ -- code addr[0] idx
  \ limit index, $7F and
  [ #ascii_codes 1- ] literal and
   + !i           \ --
  \ place end marker for next code
  0
;
```

Liegt alles auf dem Stapel, befördert `rot` den Wert des zuletzt gesehenen Symbols (Punkt oder Strich) nach oben. Davon wird die oben mutwillig addierte Konstante 1 wieder abgezogen (`1-`). Das aktuell aufgesammelte Bitmuster wird um eine Position nach links verschoben (`2*`) und mit dem Wert des letzten Symbols kombiniert (`or`). Zuletzt wird die Länge um 1 erhöht, die Reihenfolge wieder hergestellt und danach der Vorgang wiederholt. Bis eben `rot` eine Null ans Tageslicht zerrt. Dann bricht die Schleife ab.

Nach der Schleife werden `count` und `morsebits` in einen Wert zusammengepackt. `char` konsumiert das nächste Zeichen aus der Eingabe und legt seinen ASCII–Wert auf den Stapel. Danach wird dieser Wert begrenzt und als Index in die Tabelle `mtable` benutzt. Dort wird der gepackte Wert abgelegt. Für das nächste Morsezeichen wird abschließend die Ende–der–Schleife–Null auf den Stapel gelegt.

Die Zeichen–Tabelle erzeugen

Die Zeichentabelle soll nicht verschwinden, wenn die Stromversorgung für den Kontroller unterbrochen wurde. Also generieren wir die Tabelle im Flash–Speicher und belegen sie mit dem Wert `$ffff` vor. Das entspricht leerem Flash–Speicher.

```
: fl.allot ( length -- )
  dp + to dp
;
: fl.erase ( addr n -- )
  0 ?do $ffff over i + !i loop drop
;

\ create an "erased" block of flash memory
create mtable #ascii_codes fl.allot
       mtable #ascii_codes fl.erase
```

Danach legen wir eine neue Wortliste `<morse>` an, in die die Funktionen kompiliert werden, welche die Morsecode–Tabelle übersetzen. Nach den oben schon erwähnten Worten '.', '-' und '|' werden neue Worte wieder in die zuvor aktive Wortliste `forth` kompiliert.

```
vocabulary <morse> \ create wordlist
also               \ add new entry on search
<morse>            \ order list
definitions        \ new definitions go there

1 constant .
2 constant _
: |  ( 0 n1 .. nr -- 0 ) ... ;

previous definitions \ new definitions to
                     \ previous wordlist
```

Um die Morsecode–Tabelle zu *verdauen*, markieren wir den Anfang mit einer Funktion, welche die neue Wortliste aktiviert und die Buchhaltung initialisiert. Am Ende der Tabelle prüfen wir, ob auf dem Stapel eine Null liegt und deaktivieren die Wortliste.

```
: morsetable: ( -- 0 )
  also <morse>      \ activate <morse>
  0                 \ first code-end
;
: ;morsetable ( 0 -- )
  0 - abort" malformed morse table"
  previous          \ deactivate <morse>
;
```

Diese beiden Worte schalten *die neuen Tricks*, die wir dem Kompiler beigebracht haben, ein und aus. Wir haben den Kompiler zur Laufzeit verändert — ein wichtiges Merkmal von Forth. Die Tabelle ist identisch zu den schon vorgestellten Versionen:

```
morsetable:
    . _         | a    _ .         | n
    _ . . .     | b    _ _ _       | o
    _ . _ .     | c    . _ _ .     | p
    _ . .       | d    _ _ . _     | q
    .           | e    . _ .       | r
    . . _ .     | f    . . .       | s
...
;morsetable
```

Die restlichen Funktionen sind identisch zu denen, die in der alten Version schon vorgestellt wurden. Es wird `morseemit` definiert, welches ein Zeichen (als Zahl) vom Stapel nimmt und als Index in die Tabelle `mtable` benutzt. Der dort vorgefundene Wert wird ausgepackt und als Folge von Strich– und Punkt–Tönen ausgegeben. Eine Veränderung habe ich doch vorgenommen: das Zeichen wird zuerst gemorst und danach auf der seriellen Verbindung ausgegeben. So wird das auch bei Lernprogrammen gemacht. Es ist auch logisch: zuerst hört man das Morsezeichen und am Ende weiß man, was es bedeutet.

`vy 73 de dl7tux`

Referenzen

1. Pelc, VD 2012-03 S. 23ff
2. Wälde, VD 2012-03, S. 25ff

Listings

dangershield.fs

```
\ blocks/dangershield.fs

PORTB #2 portpin: sw1       \ switches
PORTB #3 portpin: sw2
PORTB #4 portpin: sw3

PORTD #5 portpin: led1      \ leds
PORTD #6 portpin: led2
PORTD #3 portpin: bz        \ buzzer

PORTC #2 portpin: sl1       \ sliders
PORTC #1 portpin: sl2
PORTC #0 portpin: sl3

PORTC #3 portpin: photocell
PORTC #4 portpin: thermometer
PORTC #5 portpin: knocksensor

PORTD #4 portpin: sr_in     \ shift register
PORTD #7 portpin: sr_oe     \ output enable
PORTB #0 portpin: sr_cl
```

base.fs

```
\ 2011-10-26  EW, 2011-11-02  CS
\ 2017-12-16  EW

\ 2 ms T_period =^= 500 Hz
: buzz ( cycles -- )
  0 ?do bz low 1ms bz high 1ms loop
;
: gap  ( cycles -- )
  0 ?do bz high 1ms bz high 1ms loop
;
: blink ( cycles -- )
  led1 high
  0 ?do  1ms 1ms 1ms loop
  led1 low
;

Edefer transmit
: piepser   ['] buzz  is transmit ;
: blinker   ['] blink is transmit ;

\ 1 point gap between elements of one symbol
\ one point gap is after every symbol, so
\ 1+2 points gap between two symbols
\ 1+4 points gap between words
#50 Evalue point
: kurz  point      transmit point gap ;
: lang  point #3 * transmit point gap ;
: Zend  point #2 * gap ; \ end of symbol
: Wend  point #4 * gap ; \ end of word

: init
  led1 pin_output led1 low
  led2 pin_output led2 low
  bz   pin_output

  piepser
;
```

morse6.fs

```
\ 2018-01-20  ew  morse6.fs
\
\ transmit morse codes
\
\ different approach by Klaus Schleisiek
\ adapted for arduino duemilanove
\     plus danger shield

marker --morse--

$80 constant #ascii_codes

: fl.allot ( length -- )
  dp + to dp
;
: fl.erase ( addr n -- )
  0 ?do $ffff over i + !i loop drop
;

\ create an "erased" block of flash memory
create mtable #ascii_codes fl.allot
       mtable #ascii_codes fl.erase

vocabulary <morse> \ create wordlist
also            \ add new entry on search
<morse>         \ order list
definitions     \ new definitions go there

1 constant .
2 constant _

\ produce packed value, store in morsetable
: |  ( 0 n1 .. nr -- 0 )
                    \ example:  . - | a
                    \ -- 0 1 2
  0  ( count )      \ -- 0 1 2 cnt
  0  ( morsebits )  \ -- 0 1 2 cnt morsebits
  begin
    rot             \ -- 0 1 cnt morsebits 2
    ?dup while      \
      1-            \ -- 0 1 cnt morsebits 1
      swap 2* or    \ -- 0 1 cnt morsebits'
      swap 1+       \ -- 0 1 morsebits' cnt+1
      swap          \ -- 0 1 cnt' morsebits'
  repeat            \ -- count morsebits
                    \ count  in bits 10..8
                    \ morsebits in    7..0
  swap #8 lshift or \ -- packed-code
  mtable            \ -- code addr[0]
  char              \ -- code addr[0] idx
  \ limit index, $7F and
  [ #ascii_codes 1- ] literal and
   + !i             \ --
  \ place end marker for next code
  0
;
previous definitions \ new definitions to
                    \ old wordlist

: morsetable: ( -- 0 )
  also <morse>      \ activate <morse>
  0                 \ first code-end
;
```

```
: ;morsetable ( 0 -- )
  0 - abort" malformed morse table"
  previous          \ deactivate <morse>
;

morsetable:
  . _         | a    _ .         | n
  _ . . .     | b    _ _ _       | o
  _ . _ .     | c    . _ _ .     | p
  _ . .       | d    _ _ . _     | q
  .           | e    . _ .       | r
  . . _ .     | f    . . .       | s
  _ _ .       | g    _           | t
  . . . .     | h    . . _       | u
  . .         | i    . . . _     | v
  . _ _ _     | j    . _ _       | w
  _ . _       | k    _ . . _     | x
  . _ . .     | l    _ . _ _     | y
  _ _         | m    _ _ . .     | z

  . _ _ _ _   | 1
  . . _ _ _   | 2
  . . . _ _   | 3
  . . . . _   | 4
  . . . . .   | 5
  _ . . . .   | 6
  _ _ . . .   | 7
  _ _ _ . .   | 8
  _ _ _ _ .   | 9
  _ _ _ _ _   | 0

  . _ . _ . _ | .   \ Punkt
  _ _ . . _ _ | ,   \ Komma
  _ _ _ . . . | :   \ Doppelpunkt
  _ . . . . _ | -   \ Bindestrich
  . _ _ _ _ . | ’   \ Apostroph
  . . _ _ . . | ?   \ Fragezeichen
  _ . _ _ . _ | (   \ Klammer
  _ . _ _ . _ | )   \ Klammer
;morsetable

: unpack  dup $00ff and swap >< $00ff and ;
: domorse ( code length -- )
  0 ?do  dup  1 and  if
      lang
    else
      kurz
    then
    2/
  loop
  drop
  Zend
;

variable o-emit
’ emit defer@ o-emit !
: morseemit
  $00ff and
  dup >r
  dup bl = if
    Wend
    drop
    bl o-emit @ execute
    exit
  then
  \ tr [A-Z] [a-z]
  dup $40 $5a within if
    $20 +
  then
  mtable + @i
  dup $ffff <> if
    unpack domorse
  else
    drop
  then
  r> o-emit @ execute
;
: morse
  [’] emit defer@ o-emit !
  [’] morseemit is emit
;
: endmorse
  o-emit @ is emit
;

: msg
  s" ab ab ab" itype ;
: call
  s" cq dl cq dl de dl7tux dl7tux k" itype ;
: test
  init piepser lang &500 ms
  morse msg endmorse cr
  morse s" 1 2 3 schnell herbei" itype
  endmorse cr
  morse s" a: b. c," itype endmorse cr
  morse call endmorse cr
;
```

Color–QR–Codes scannen

Bernd Paysan

net2o verwendet QR–Codes in einer farbigen Variante. Der Scanvorgang wird dadurch noch einfacher als bei normalen QR–Codes. Dieser Artikel erklärt, wie man in Forth (insbesondere auf Android) solche Codes scannt.

Einleitung

QR–Codes haben sich in den letzten Jahren immer weiter ausgebreitet. Gerade zum Austausch kurzer nicht–menschenlesbarer Informationen sind sie gut geeignet, weil Maschinen sie besser scannen können als Buchstabensuppe. In net2o wollte ich QR–Codes verwenden, um Keys zu tauschen und zu authentisieren, und mich dabei nicht auf große und träge Libraries verlassen. Es sollte auch möglichst schnell scannen und das Programm sollte möglichst einfach sein.

Die Grundidee von QR–Codes ist schnell beschrieben: Man hat ein quadratisches Feld mit weißen und schwarzen Feldern, die jeweils für ein Bit stehen. An drei Ecken werden besondere Muster aufgebaut, damit man die Orientierung erkennen kann. Diese Muster muss der Scanner erst finden, damit er den QR–Code richtig hinrotieren kann.

Farbe ins Spiel!

Da die Hauptanwendung bei net2o das Scannen vom Bildschirm ist, und auch sonst der farbige Ausdruck keine großen Schwierigkeiten mehr macht, lag nahe, diesen Elementen zur Orientierung einfach einen eigenen Farb–Kanal zu geben (Blau), und die anderen beiden Farbkanäle (Rot und Grün) dazu zu nutzen, zwei statt einem Bit pro Element zu übertragen. Die Suche nach den Ecken findet dann im Blau–Kanal statt, und für die Markierung, welches Eck an welcher Stelle ist, sind mit den beiden anderen Kanälen genau die 4 möglichen Zustände da, die man für eine eindeutige Zuordnung (inklusive aller Drehungen und Spiegelungen) braucht.

Der Input aus der Kamera kommt auf Android und mit SOIL unter Linux als OpenGL–Textur herein, was sehr bequem ist, kann man mit OpenGL doch komplexe grafische Transformationen mit ganz einfachen Befehlen erwirken: Man malt die Textur in einen Renderbuffer, und extrahiert aus diesem Render–Buffer dann die Pixel, die man braucht. Nachteil: Selbst wenn man das Programm unter Linux als einfaches Script zum Testen mit Bildern aus der Kamera versorgt, läuft das Script nur, wenn es einen OpenGL–Context bekommt. Und dafür ist halt ein lokaler X–Server (oder Wayland) nötig.

Schwierigkeiten

Die erste Version des Scanners entstand auf meinem Galaxy Note 2, und da hatte der QR–Code noch einen schwarzen Hintergrund. Auf diesem Gerät funktionierte sie leidlich, auch wenn das Scharfstellen etwas langsam ging, und der Belichtungsabgleich der Kamera zum Überstrahlen neigte — und der Farbabgleich das fehlende Blau dann in den überstrahlten Teilen dazuerfand, was das Scannen der Ecken erschwerte.

Mit meinem Note 8 wurde das dann alles noch viel schlechter — weil die Kamera viel besser ist. Das Scharfstellen ging hier nicht langsam, sondern rasend schnell, und die scharfgestellte Preview zeigte jeden einzelnen Sub–Pixel des Displays als mindestens einen Pixel an (je nachdem, wie nah man drangeht). Beim notwendigen Verkleinern sorgten Aliasing–Effekte dann dafür, dass es wilde dunkle Moiré–Muster gab, die die Erkennung schwer durcheinander brachten. Außerdem ist die Kamera so fix und lichtempfindlich, dass sie die mit mehreren hundert Hz flackernde Dimmung des Displays als unterschiedlich helle Streifen darstellt (Folge des sequentiellen Auslesens des CMOS–Sensors).

Um den Weißabgleich und das Überstrahlen in Griff zu bekommen, habe ich das Bild invertiert. Der Blau–Kanal für die Markierungen an den Ecken (die zum einfacheren Erkennen mit etwas Abstand außerhalb des Quadrats gewandert sind) ist also durch *fehlendes Blau* gekennzeichnet. Das hat gut geklappt, der Sensor hat keine Probleme, das Bild ohne Überstrahlen aufzunehmen.

Als Nächstes musste ich das Roh–Bild aus der Kamera 1:1 in eine Textur schreiben, und diese Textur dann mit einem Mipmap versehen, damit das Verkleinern ohne Moiré funktionierte. Bei der Gelegenheit habe ich noch etwas ausgemistet, und Altlasten im `gl-helper.fs` von MINOS2 beseitigt.

Erklärung des Quellcodes

Der QR–Code ist 18×18 Pixel groß (16×16 Daten, und am Rand Platz für Fehlerkorrektur), zusammen mit den Markern an der Ecke 24×24 Pixel. Ich schneide das Kamera–Bild auf ein Quadrat zurecht, und zeichne das (verkleinert) in 128×128 Pixel. `draw-scan` zeichnet einfach ein Rechteck, und `scan-xy` liefert die nötigen Eckpunkte, damit aus dem ursprünglich rechteckigen Vorschaubild der Kamera nur das Quadrat in der Mitte übrig bleibt.

Der so gezeichnete Renderbuffer wird dann mit `glReadPixels` von der GPU in den Hauptspeicher verschoben. Dort sucht `adapt-rgb` erst mal nach den hellsten und dunkelsten Werten in den RGB–Kanälen, und adaptiert die verschiedenen Schwellwerte: Blau auf 25% zwischen Min– und Maximum, Grün auf 40% und Rot auf 50%. Der niedrige Wert bei Blau ist wichtig, da wir

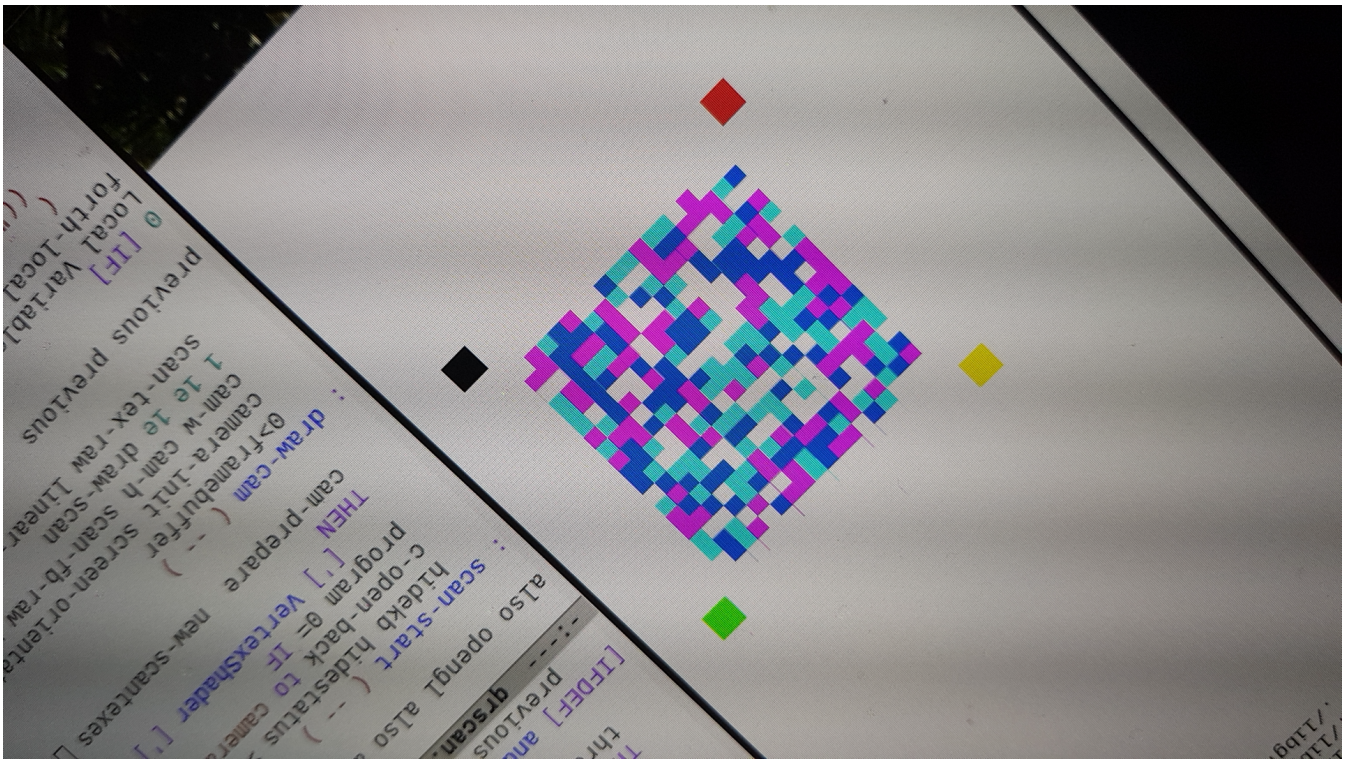


Abbildung 1: Ein gedrehter farbiger QR–Code (Aufnahme: Note 8)

wirklich nur die Marker finden wollen, und dort dann auch die anderen Farben korrekt und prägnant sein müssen.

Genau nach diesen Markern sucht `search-corners` als nächstes. Der ganze Bereich wird systematisch durchkämmt, und überall, wo das Blau unterhalb des gesetzten Schwellwerts ist, der Abstand zum Mittelpunkt des Scans berechnet — der Punkt mit dem geringsten Abstand wird abhängig von den anderen beiden Farben in einem Array gespeichert.

Nun guckt `?legit`, ob wir vier solche Punkte gefunden haben. Nur dann können wir die weiteren Berechnungen auch durchführen. Die vier Punkte $p_{0..3}$ bilden ein Z.

Dazu müssen wir eine passende Transformations–Matrix finden, die uns das Bild so schert, dreht und vergrößert oder verkleinert, dass es schön in ein Rechteckraster passt. Als Erstes berechne ich dazu den Schnittpunkt der beiden Geraden, die durch die jeweils diagonal gegenüberstehenden Eckpunkte bestimmt wird. Wer sich noch dunkel an den Mathe–Unterricht erinnern kann: Das war das mit der CRAMERschen Regel und den vielen Determinanten. Vom Rechenaufwand ist die bei zwei Variablen gerade noch tragbar. Ich schreibe mal die vernünftig faktorisierte Gleichung hin:

$$
\begin{aligned}
dx_{01} &= x_0 - x_1 \\
dx_{23} &= x_2 - x_3 \\
dy_{01} &= y_0 - y_1 \\
dy_{23} &= y_2 - y_3 \\
det_1 &= dx_{01} * dy_{23} - dy_{01} * dx_{23} \\
dxy_{01} &= x_0 y_1 - x_1 y_0 \\
dxy_{23} &= x_2 y_3 - x_3 y_2 \\
x &= \frac{dxy_{01} dx_{23} - dxy_{23} dx_{01}}{det_1} \\
y &= \frac{dxy_{01} dy_{23} - dxy_{23} dy_{01}}{det_1}
\end{aligned}
$$

Hier könnte man noch gucken, ob sich die Linien wirklich schneiden, und könnte abbrechen, wenn nicht. x und y geben uns die Werte für die Translation in der Transformationsmatrix.

Die Werte für Skalierung und Rotation bekommen wir, indem wir die waagerechten und senkrechten Linien (also die, die sich eben nicht schneiden) jeweils addieren, und geeignet skaliert in die Matrix oben links eintragen, das macht `scale+rotate`. Die so ermittelte Matrix stellt das Ergebnis der Transformation dar, die wir jetzt gerade betrachten: Schickt man die gewünschten, normalisierten Eckkoordinaten durch diese Matrix, so erhält man das tatsächlich beobachtete, transformierte Quadrat.

Eigentlich wollten wir genau das Gegenteil, wir wollten ja dieses Quadrat wieder normiert haben. Also: Invertieren wir die Matrix! Das ist jetzt auch wieder klassische Numerik; der Algorithmus hat drei Komponenten: Skalieren einer Zeile, Skalieren und addieren einer zweiten Zeile, oder Vertauschen zweier Zeilen. Vertauschen muss man aufgrund der besonderen Form dieser Matrix nur ggf. die ersten beiden Zeilen, das ist also ein Shortcut; der Code ist nicht allgemein–gebräuchlich. Eigentlich muss man die Matrix komplett vorsortieren.

Diese so invertierte Matrix setzen wir dann als Abbildungs–Matrix für OpenGL, zeichnen das Kamera–Bild damit erneut, und haben es zumindest an die richtige

Position gedreht und verschoben. Perspektivische Korrektur will ich auch noch einbauen. Grundüberlegung wird auch hier wieder sein, dass das Bild etwas gekippt ist, und die Verhältnisse der Seitenlängen (vom Schnittpunkt aus gerechnet) angeben, wie stark.

Jedenfalls können wir nun zum zweiten Mal den gezeichneten Framebuffer ins RAM holen, und dieses Mal einfach die Zeilen auslesen. Ausgewertet wird jeweils jeder 4. Punkt, also Punkte, die in etwa in der Mitte der gescannten Pixel des QR–Codes liegen. Auch hier wieder in 128×128 Pixel, wobei nur 96×96 wirklich gebraucht werden. Zur Korrektur eventueller Fehler werden noch die Zeilen und Spalten am Rand gelesen; der Tag/Typ des QR–Codes wird in den vier Ecken codiert (also ein Byte).

Anders als normale QR–Codes habe ich mich nach anfänglichen Überlegungen dagegen entschieden, eine Fehlerkorrektur einzubauen, und stattdessen eine Absicherung über einen Hash zu nehmen. In den Hash fließt neben dem Code selbst noch ein Initial–Vektor, der auch als Challenge für ein Challenge–Response–Verfahren benutzt wird.

Damit muss der Scan bei jedem Punkt klappen, oder man müsste zumindest auf Verdacht mal brute Force einzelne Punkte invertieren, und gucken, ob dann ein korrekter Hash herauskommt. Eine Alternative wäre, etwa einen Reed–Solomon–Code zu verwenden, mit dem man dann einige Fehler korrigieren kann. Tatsächlich treten bei dieser Art Anwendung (Abscannen vom Bildschirm) aber keine groben Fehler auf, die nicht wenigstens im nächsten Bild wieder in Ordnung sind. Und Dreck auf der Smartphone–Linse kann der Nutzer auch wegwischen.

Kamera? Bilder? Und Benchmarks?

Auf Android nutze ich die Rückseitenkamera; der Scanvorgang funktioniert auch als gegenseitiger Abgleich, wenn einer scannt, und der andere nur sein Gerät in die Kamera hält — die andere Richtung des Abgleichs erfolgt dann über's Netz(2o), die Challenge am Rand des Scans ist Beweis genug, dass die Gegenseite das Bild gesehen hat. Unter Linux habe ich derzeit noch kein Kamera–Interface implementiert, und die üblichen Laptops haben ja ohnehin nur eine Front–Kamera, die nur sehr bedingt für solche Scans geeignet ist. Volume–Up dient als Auslöser, um die aktuelle Kamera–Vorschau sowie die beiden Verkleinerungen als PNG abzuspeichern. Die API, die ich mit `minos2/android-recorder.fs` (gehört zu Gforth) nutze, ist ziemlich simpel. Man erzeugt eine Kamera, schaltet den Preview ein, und bekommt die Previews in eine Textur zusammen mit einem Event geliefert, auf das man warten kann. Das macht alles `camp-prepare`, und mit `cam-end` schaltet man die Kamera auch wieder ab. `camera-init` wartet auf die Textur und setzt den OpenGL–Context so, dass man das Preview–Bild zeichnen kann.

Damit man das Scannen dennoch auch unter Linux testen kann, wird dort über libSOIL Bilder (PNGs oder JPGs) eingelesen, die dann durch den Algorithmus gejagd werden. Das hat zudem noch den schönen Vorteil, dass man bequem und reproduzierbar debuggen kann, denn die Bilder kommen ja nicht live von einer Kamera, sondern sind aufgezeichnet.

Wer will, kann das gleich als Script (Quelltext `qrscan`) anlegen, und direkt starten. Das Script funktioniert nur, wenn man auch einen GL–Kontext (EGL+GLESv2 reichen) aufmachen kann, also remote einloggen geht nicht so gut.

Damit kann man auch ein paar Benchmarks machen. Auf einer Intel HD–GPU im Laptop dauert ein Scan (ohne Laden und Mipmappen des Bildes) 1–2ms (mit OpenGL statt OpenGL ES etwa 1,5 mal länger), auf dem Note 8 11–13ms, und auf einem Odroid C2 mit schwacher GPU ca. 30ms. Externe GPUs brauchen aufgrund des Verschiebens von GPU–RAM zu CPU–RAM länger, eine etwas ältere Radeon–Karte hier braucht auch etwa 10ms pro Scan. Ähnlich lang dauert es, wenn ich den NVidia–Optimus–Chip auf dem Laptop bemühe, der ebenfalls sein eigenes GPU–RAM hat (das ist dann aber wieder OpenGL statt OpenGL ES, was ebenfalls langsamer ist).

Trotzdem: Die Erkennungszeiten reichen völlig aus, um ohne nervige Verzögerung im Nullkommanix zu scannen. Der Algorithmus funktioniert auch schon, wenn die Kamera noch nicht richtig scharf gestellt hat (eigentlich sogar besser, denn die Gegenmaßnahme gegen das Moire–Problem wäre dann nicht nötig), also: Scan an, und fertig. Das einzige, was noch fehlt, ist die Korrektur der perspektivischen Verzerrung, dann muss man die Kamera nicht mal mehr einigermaßen plan halten.

Referenzen

[1] Bernd Paysan, *net2o Quellcode & Wiki,* `https://net2o.de/`

Listings

`qrscan.fs`

```
1   \ scan color QR codes on Android
2
3   \ Copyright (C) 2016-2018   Bernd Paysan
4
5   \ This program is free software: you can
6   \ redistribute it and/or modify it under the terms
7   \ of the GNU Affero General Public License as
8   \ published by the Free Software Foundation,
9   \ either version 3 of the License, or (at your
10  \ option) any later version.
11
12  \ This program is distributed in the hope that it
13  \ will be useful, but WITHOUT ANY WARRANTY;
14  \ without even the implied warranty of
15  \ MERCHANTABILITY or FITNESS FOR A PARTICULAR
16  \ PURPOSE.  See the GNU Affero General Public
17  \ License for more details.
18
19  \ You should have received a copy of the GNU
20  \ Affero General Public License along with this
21  \ program.  If not, see
22  \ <http://www.gnu.org/licenses/>.
23
```

```
require minos2/gl-helper.fs
[IFDEF] android
    require minos2/android-recorder.fs
[ELSE]
    [IFUNDEF] cam-w
        $100 value cam-w
        $100 value cam-h
    [THEN]
[THEN]

\ replace some tools available under net2o

[IFUNDEF] qr(
    debug: qr(
    +db qr( \ turn it on )
[THEN]

[IFUNDEF] xtype
    : hex[ ]] [: [[ ; immediate
    : ]hex ]] ;] $10 base-execute [[ ; immediate
    : xtype ( addr u -- )  hex[
        bounds ?DO  I c@ 0 <# # # #> type  LOOP  ]hex ;
[THEN]

[IFUNDEF] taghash?
    : taghash? ( addrkey u1 addrecc u2 tag -- flag )
        drop 2drop 2drop true ;
[THEN]

\ scan matrix manipulation

Create scan-matrix
1e sf, 0e sf, 0e sf, 0e sf,
0e sf, 1e sf, 0e sf, 0e sf,
0e sf, 0e sf, 1e sf, 0e sf,
0e sf, 0e sf, 0e sf, 1e sf,

scan-matrix 11 sfloats + Constant 3d-enabler

32 sfloats buffer: scan-inverse
32 sfloats buffer: inverse-default
inverse-default 32 sfloats bounds
[?DO] 1e fdup [I] sf! [I] 4 sfloats + sf! 9 sfloats [+LOOP]

84e FValue x-scansize
84e FValue y-scansize

0e FValue y-offset
0e FValue x-offset

\ matrix inversion

' dfloats alias 8*
: .mat { mat -- }
    4 0 DO  cr
        8 0 DO
            mat J 8* I + sfloats + sf@ f.
        LOOP
    LOOP ;
: init-scan' ( -- )
    inverse-default scan-inverse [ 32 sfloats ]L move ;
: sfax+y8 ( ra addr1 addr2 -- )
    [ 8 sfloats ]L bounds ?DO
        dup sf@ fover I sf@ f* f+ dup sf! sfloat+
    [ 1 sfloats ]L +LOOP  drop fdrop ;
: sfax8 ( ra addr -- )
    [ 8 sfloats ]L bounds ?DO
        fdup I sf@ f* I sf!
    [ 1 sfloats ]L +LOOP  fdrop ;
: tij8 ( addr1 addr2 -- )
    [ 8 sfloats ]L bounds ?DO
        dup sf@ I sf@ dup sf! I sf! sfloat+
    [ 1 sfloats ]L +LOOP  drop ;

: matrix-invert4 { mat -- }
    \ shortcut to invert typical matrix
    mat sf@ fabs mat [ 8 sfloats ]L + sf@ fabs f< IF
        mat dup [ 8 sfloats ]L + tij8
        \ exchange two lines
    THEN
    4 0 DO
        4 0 DO
            mat J [ 9 sfloats ]L * + sf@ 1/f
            I J <> IF
                mat I 8* sfloats +
                mat J 8* sfloats +
                over J sfloats + sf@ f*
                fnegate sfax+y8
            ELSE
                mat J 8* sfloats + sfax8
            THEN
            ( mat .mat cr ) \ debugging output
        LOOP
    LOOP ;

scan-inverse  0 sfloats + Constant x-scale
scan-inverse  1 sfloats + Constant y-rots
scan-inverse  8 sfloats + Constant x-rots
scan-inverse  9 sfloats + Constant y-scale
scan-inverse 24 sfloats + Constant x-spos
scan-inverse 25 sfloats + Constant y-spos

: >scan-matrix ( -- )
    scan-inverse matrix-invert4
    scan-matrix [ scan-inverse 4 sfloats + ]L
    [ 32 sfloats ]L bounds ?DO
        I over [ 4 sfloats ]L move
        [ 4 sfloats ]L +
    [ 8 sfloats ]L +LOOP  drop
    -1e 3d-enabler sf! ;

\ scan constants

$40 Value scan-w
scan-w 2/ dup * 1- 2/ 2/ 1+ Constant buf-len
scan-w 2* Value scan-right
scan-w 2* dup 3 rshift + negate Value scan-left

also opengl

: draw-scan ( direction xscale yscale -- )
    \G draw a scan rotated/tilted by scan matrix
    fover fnegate fover fnegate 0e
    { f: sx f: sy f: -sx f: -sy f: z }
    v0 i0 >v
     -sx  sy z >xyz n> rot>st   $FFFFFFFF rgba>c v+
      sx  sy z >xyz n> rot>st   $FFFFFFFF rgba>c v+
      sx -sy z >xyz n> rot>st   $FFFFFFFF rgba>c v+
     -sx -sy z >xyz n> rot>st   $FFFFFFFF rgba>c v+
    v> drop 0 i, 1 i, 2 i, 0 i, 2 i, 3 i,
    GL_TRIANGLES draw-elements ;

Variable scan-buf-raw
Variable scan-buf0
Variable scan-buf1
Variable red-buf
Variable green-buf
Variable blue-buf

$28 Value blue-level#
$70 Value green-level#
$70 Value red-level#

' sfloats alias rgbas
\ one rgba is coincidently the size of an sfloat
' sfloat+ alias rgba+

[IFDEF] distdebug
    3 cells buffer: dist0
    dist0 cell+ Constant dist0-max
    dist0-max cell+ Constant dist0-min
    3 cells buffer: dist1
```

```
    dist1 cell+ Constant dist1-max
    dist1-max cell+ Constant dist1-min

    : ?? ( value level -- flag )
        - dup dup 0< IF  negate dist0
        ELSE  dist1  THEN
        2dup +! cell+
        2dup @ umax over ! cell+
        tuck @ umin swap !
        0< ;
[ELSE]
    ' u< alias ??
[THEN]

: extract-strip ( addr u step -- strip )
    rgbas { step }
    0 -rot bounds U+DO
        2* I 1+ c@ green-level# ?? -
        2* I    c@ red-level#   ?? -
    step +LOOP ;

$51 buffer: guessbuf
guessbuf $40 + Constant guessecc
guessecc $10 + Constant guesstag

scan-w 3 rshift Constant scan-step

: >strip ( index --- addr )
    2* 2* scan-w + scan-w 2* * scan-w + rgbas
    scan-buf1 $@ rot safe/string drop ;
: >strip32 ( addr -- addr' u step )
    scan-right - $100 4 ;
: >guess ( -- addr u )
    guessbuf $40 2dup bounds -8 -rot U+DO
        dup >strip >strip32
        extract-strip I be-l!
        1+
    4 +LOOP  drop ;
: ecc-hor@ ( off -- l )
    >strip >strip32 extract-strip ;
: ecc-ver@ ( offset -- ul )
    #-8 >strip + scan-w dup 3 lshift *
    scan-w 3 lshift extract-strip ;
: tag@ ( -- tag )
    #-9 >strip scan-left + $130
    [ #17 4 * ]L extract-strip 4 lshift
    #08 >strip scan-left + $130
    [ #17 4 * ]L extract-strip or ;

: >guessecc ( -- )
    #-9        ecc-hor@ guessecc      be-l!
    #08        ecc-hor@ guessecc  4 + be-l!
    scan-left  ecc-ver@ guessecc  8 + be-l!
    scan-right ecc-ver@ guessecc $C + be-l! ;

: ecc-ok? ( addrkey u1 addrecc u2 -- flag )
    2dup + c@ taghash? ;

$8000 Constant init-xy

: get-minmax ( addr u -- min max )
    $FF $00 2swap
    bounds ?DO
        I c@ tuck umax >r umin r>
    4 +LOOP ;
: get-minmax-rgb ( addr u -- minr maxr
                           ming maxg
                           minb maxb )
    swap 3 bounds DO
        I over get-minmax rot
    LOOP  drop ;

#10 Cells buffer: p0
p0 2 cells + Constant p1
p1 2 cells + Constant p2
p2 2 cells + Constant p3
p3 2 cells + Constant px

: min²! ( x y addr -- ) >r
    over dup * over dup * +
    r@ 2@ dup * swap dup * + u< IF
        r> 2!  EXIT
    THEN  2drop rdrop ;

: search-corners ( -- )
    init-xy p0 !  p0 p0 cell+ 7 cells cmove
    \ fill all with the same contents
    scan-buf0 $@ drop
    scan-w dup negate DO
        scan-w dup negate DO
            dup 2 + c@ blue-level#  u< IF
                dup 1+  c@ green-level# u< 2*
                over    c@ red-level#   u< -
                3 and 2 xor
                2* cells p0 + I J rot min²!
            THEN
            rgba+
        LOOP
    LOOP  drop ;

: ?legit ( -- flag )
    p0 2@ init-xy dup d<>
    p1 2@ init-xy dup d<> and
    p2 2@ init-xy dup d<> and
    p3 2@ init-xy dup d<> and ;

: compute-xpoint ( -- rx ry )
    p0 2@ s>f s>f { f: y0 f: x0 }
    p3 2@ s>f s>f { f: y1 f: x1 }
    p1 2@ s>f s>f { f: y2 f: x2 }
    p2 2@ s>f s>f { f: y3 f: x3 }
    x0 y1 f* y0 x1 f* f- { f: dxy01 }
    x2 y3 f* y2 x3 f* f- { f: dxy23 }
    x0 x1 f- x2 x3 f- { f: dx01 f: dx23 }
    y0 y1 f- y2 y3 f- { f: dy01 f: dy23 }
    dx01 dy23 f* dy01 dx23 f* f- 1/f { f: /det1 }
    dxy01 dx23 f* dxy23 dx01 f* f- /det1 f* \ x
    dxy01 dy23 f* dxy23 dy01 f* f- /det1 f* \ y
    fover f>s fdup f>s px 2! ;

: p+ ( x1 y1 x2 y2 -- x1+x2 y1+y2 )
    rot + >r + r> ;
: p2/ ( x1 y1 -- x2 y2 )
    2/ swap 2/ swap ;
: p- ( x1 y1 x2 y2 -- x1-x2 y1-y2 )
    rot swap - >r - r> ;

: scan-grab ( w h addr -- )
    >r  0 0 2swap
    2dup * rgbas r@ $!len
    GL_RGBA GL_UNSIGNED_BYTE r> $@ drop glReadPixels ;
: scan-grab-buf ( addr -- )
    scan-w 2* dup rot scan-grab ;
: scan-grab-cam ( addr -- )
    cam-w cam-h rot scan-grab ;

tex: scan-tex-raw
tex: scan-tex

0 Value scan-fb-raw
0 Value scan-fb

: scan-grab-raw ( -- )
    cam-w cam-h scan-fb-raw >framebuffer
    scan-buf-raw scan-grab-cam ;
: scan-grab0 ( -- )
    scan-buf0 scan-grab-buf ;
: scan-grab1 ( -- )
    scan-buf1 scan-grab-buf ;

also soil
```

```
0 Value scan#

: save-png0 ( -- )
    [: ." scanimg0-" scan# 0 .r ." .png" ;] $tmp
    SOIL_SAVE_TYPE_PNG 128 dup 4
    scan-buf0 $@ drop SOIL_save_image ;
: save-png1 ( -- )
    [: ." scanimg1-" scan# 0 .r ." .png" ;] $tmp
    SOIL_SAVE_TYPE_PNG 128 dup 4
    scan-buf1 $@ drop SOIL_save_image ;
: save-png-raw ( -- )
    [: ." scanimgraw-" scan# 0 .r ." .png" ;] $tmp
    SOIL_SAVE_TYPE_PNG cam-w cam-h 4
    scan-buf-raw $@ drop SOIL_save_image ;
: save-pngs ( -- )
    scan-grab-raw save-png-raw
    save-png0 save-png1
    1 +to scan#
    0>framebuffer ;

previous

: .xpoint ( x y -- )
    p0 2@ swap . . space
    p1 2@ swap . . space
    p2 2@ swap . . space
    p3 2@ swap . . space
    fswap f. f. cr ;

: new-scantex-raw ( -- )
    scan-tex-raw 0>clear
    cam-w cam-h GL_RGBA new-textbuffer to scan-fb-raw ;
: new-scantex ( -- )
    scan-tex 0>clear
    scan-w 2* dup GL_RGBA new-textbuffer to scan-fb ;
: new-scantexes ( -- )
    scan-fb 0= IF
        new-scantex-raw new-scantex 0>framebuffer
    THEN ;
: scale+rotate ( -- )
    p1 2@ p0 2@ p- p3 2@ p2 2@ p- p+ p2/
    s>f y-scansize f/ y-rots sf!
    s>f x-scansize f/ x-scale sf!
    p0 2@ p2 2@ p- p1 2@ p3 2@ p- p+ p2/
    s>f y-scansize f/ y-scale sf!
    s>f x-scansize f/ x-rots sf! ;
: set-scan' ( -- )
    compute-xpoint ( .. x y )
    scale+rotate
    y-offset f+ scan-w fm/ y-spos sf!
    x-offset f+ scan-w fm/ x-spos sf! ;

: scan-xy ( -- sx sy )
    1e cam-h cam-w over umin swap fm*/
    1e cam-w cam-h over umin      fm*/ ;

: scan-legit ( -- ) \ resize a legit QR code
    clear init-scan' set-scan' >scan-matrix
    scan-matrix MVPMatrix set-matrix
    scan-matrix MVMatrix  set-matrix
    scan-tex-raw linear-mipmap 0 scan-xy
    draw-scan scan-grab1 ;

: scan-legit? ( -- addr u flag )
    [IFDEF] distdebug
        dist0 off dist0-max off dist0-min on
        dist1 off dist1-max off dist1-min on
    [THEN]
    scan-legit >guess
    >guessecc tag@ guesstag c!
    2dup guessecc $10 ecc-ok? ;

: tex-frame ( -- )
    program init-program set-uniforms
    unit-matrix MVPMatrix set-matrix
    unit-matrix MVMatrix set-matrix ;

: draw-scaled ( -- )
    tex-frame scan-w 2* dup scan-fb >framebuffer
    scan-tex-raw linear-mipmap 0 scan-xy draw-scan
    scan-grab0 ;

previous

[IFDEF] android
    require android/qrscan-android.fs
    also android
[ELSE]
    [IFDEF] linux
        require linux/qrscan-linux.fs
    [THEN]
[THEN]

[IFUNDEF] scan-result
    : scan-result ( addr u tag -- )
        >r
        bounds ?DO  ." qr : " I $10 xtype cr $10 +LOOP
        r> ." tag: " dup hex. cr
        ." ecc: " guessecc $10 xtype cr
        [IFDEF] distdebug
            ." dist/min/max: "
            dist0 @ s>f [ 18 18 * ]L fm/ f>s .
            dist0-min ? dist0-max ? space
            dist1 @ s>f [ 18 18 * ]L fm/ f>s .
            dist1-min ? dist1-max ? cr
        [THEN] ;
[THEN]

: adapt-rgb ( -- )
    scan-buf0 $@ get-minmax-rgb
    over - 2/ 2/   + to blue-level#
    \ blue level is 1/4 of total
    over - 2 5 */  + to green-level#
    \ green at 40% of total
    over - 2/      + to red-level# ;
    \ red at 50% of total

: scan-once ( -- )
    draw-cam
    !time draw-scaled adapt-rgb
    search-corners
    ?legit IF  scan-legit? IF
            guessecc $10 + c@ scan-result
            qr( ." took: " .time cr )
            qr( save-png1 1 +to scan# )
        ELSE  2drop  THEN
    THEN
    ekey? IF  ekey k-volup =  IF
            save-pngs
        THEN  THEN ;
: scan-loop ( -- )
    1 level# +!@ >r  BEGIN
        scan-once >looper
    level# @ r@ <= UNTIL
    rdrop ;

: scan-qr ( -- )
    scan-start  [’] scan-loop catch  level# off
    cam-end 0>framebuffer
    [IFDEF] showstatus showstatus [THEN]
    [IFDEF] terminal-program
        terminal-program terminal-init
    [THEN]
    dup IF
        ." Scan failed" cr
    ELSE
        ." Scan completed" cr
    THEN
    throw ;

previous
```

qrscan-linux.fs

```
\ scan color QR codes on Android

\ Copyright (C) 2018   Bernd Paysan

\ This program is free software: you can
\ redistribute it and/or modify it under the terms
\ of the GNU Affero General Public License as
\ published by the Free Software Foundation,
\ either version 3 of the License, or (at your
\ option) any later version.

\ This program is distributed in the hope that it
\ will be useful, but WITHOUT ANY WARRANTY;
\ without even the implied warranty of
\ MERCHANTABILITY or FITNESS FOR A PARTICULAR
\ PURPOSE.  See the GNU Affero General Public
\ License for more details.

\ You should have received a copy of the GNU
\ Affero General Public License along with this
\ program.  If not, see
\ <http://www.gnu.org/licenses/>.

require minos2/soil-texture.fs

: scan-start ( -- )
    dpy 0= IF window-init THEN
    new-scantexes ;
: draw-cam ( -- )
    scan-tex-raw next-arg
    qr( ." image: " 2dup type cr )
    load-texture to cam-h to cam-w
    argc @ 1 <= IF
        level# off
    THEN ;
: cam-end ( -- ) ;

```

qrscan-android.fs

```
\ scan color QR codes on Android

\ Copyright (C) 2016   Bernd Paysan

\ This program is free software: you can
\ redistribute it and/or modify it under the terms
\ of the GNU Affero General Public License as
\ published by the Free Software Foundation,
\ either version 3 of the License, or (at your
\ option) any later version.

\ This program is distributed in the hope that it
\ will be useful, but WITHOUT ANY WARRANTY;
\ without even the implied warranty of
\ MERCHANTABILITY or FITNESS FOR A PARTICULAR
\ PURPOSE.  See the GNU Affero General Public
\ License for more details.

\ You should have received a copy of the GNU
\ Affero General Public License along with this
\ program.  If not, see
\ <http://www.gnu.org/licenses/>.

also opengl also android

: scan-start ( -- )
    hidekb hidestatus >changed  screen+keep
    c-open-back to camera
    program 0= IF
        ['] VertexShader ['] FragmentShader
        create-program to program
    THEN
    cam-prepare  new-scantexes ;

: draw-cam ( -- )
    0>framebuffer
    camera-init
    screen-orientation 1e 1e draw-scan sync
    cam-w cam-h scan-fb-raw >framebuffer
    1 1e 1e draw-scan
    scan-tex-raw linear-mipmap mipmap ;

previous previous
```

qrscan

```
#! /usr/bin/gforth-fast
\ -*- mode: Forth -*-

warnings off \ in production mode,
\ we don't want any warnings

fpath+ /usr/share/gforth/0.7.9_20180208/net2o-0.5.6-20180208

require qrscan.fs
scan-qr bye
```

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`

Mainz Rolf Lauer möchte im Raum Frankfurt, Mainz, Bad Kreuznach eine lokale Gruppe einrichten.
Mail an `rolf@llar.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`, voller Key:
`kQusJzA;7*?t=uy@X}1GWr!+0qqp_Cn176t4(dQ*`

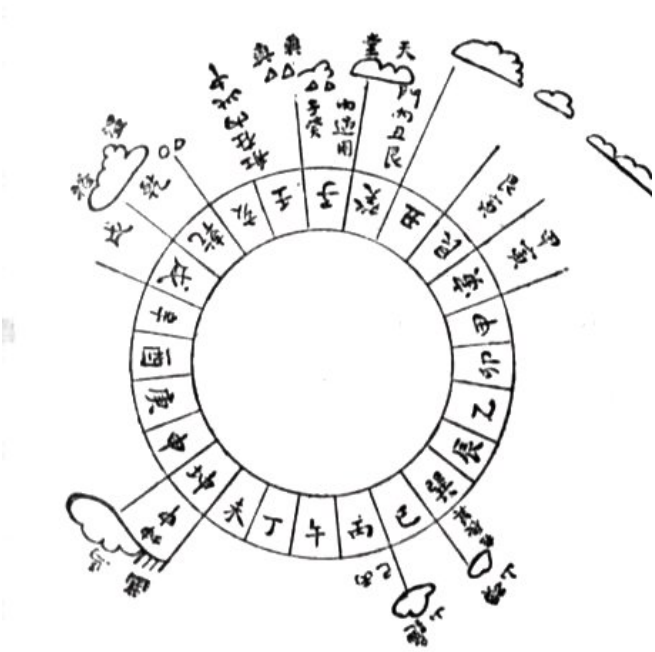
05.–08.04.2018 Forth–Tagung in Essen–Horst
`https://tagung.forth-ev.de`

14.–16.09.2018 Maker Faire Hannover
`http://www.makerfairehannover.com`

September 2018? EuroForth
`http://www.euroforth.org`

27.–30.12.2018 35C3 Leipzig?
`http://events.ccc.de/congress`

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 2018 vom 5. bis 8. April im Linux Hotel in Essen–Horst

Die Tagung findet in der Villa Vogelsang statt, Antonienallee 1, 45279 Essen-Horst. Das Gebäude liegt in einem Park mit Aussicht auf die Ruhr. Idylle pur!

Anreise

Essen ist mit der Bahn, Essen–Horst mit der S–Bahn erreichbar; per Auto über die A40 (Essen–Kray oder Essen–Frillendorf). Flughäfen gibt es mindestens drei zur Auswahl: Düsseldorf, Dortmund, Köln. Die liegen aber nicht „um die Ecke“. Und ja, man kann auch zu Fuß oder mit dem Fahrrad kommen — ist erwiesenermaßen machbar!

Anmeldung

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

Programm

Donnerstag

ab 14:00	Frühankommer Workshops

Freitag

vormittags	Frühankommer Ausflug
14:00	Begin der Tagung, Vorträge und Workshops

Samstag

vormittags	Ausflug
nachmittags	Vorträge und Workshops

Sonntag

09:00	Mitgliederversammlung
12:00	Ende der Tagung