Volume V/Nr.3 - September 1989

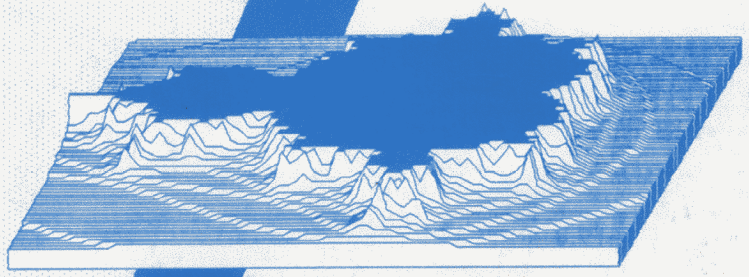
VIERTE DIMENSION

Themen

- Warum Postfix?
- Mandelbrot
- Der fleißige Biber
- Assembler im Vergleich
- CASE-Situation, Teil 2
- Read-only Stringfelder

FORTH MAGAZIN

7,50 DM



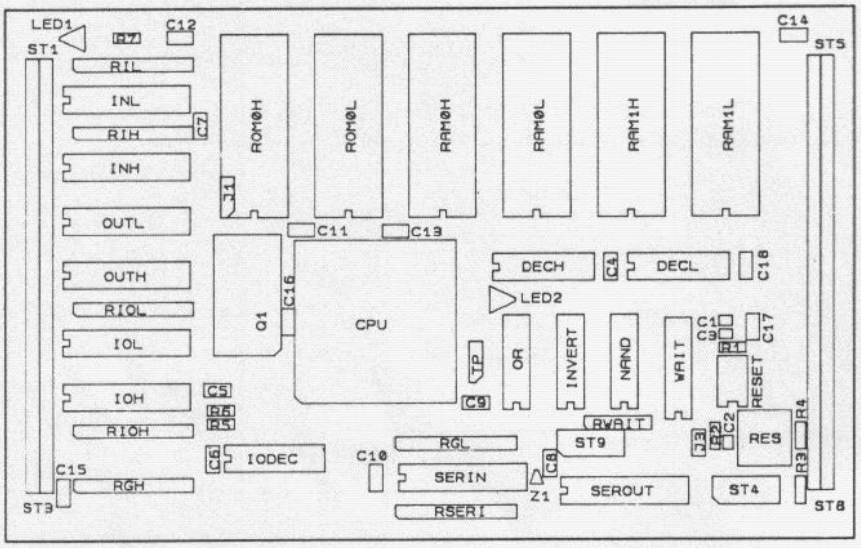
Real Time Express Real Time Express Real Time Express

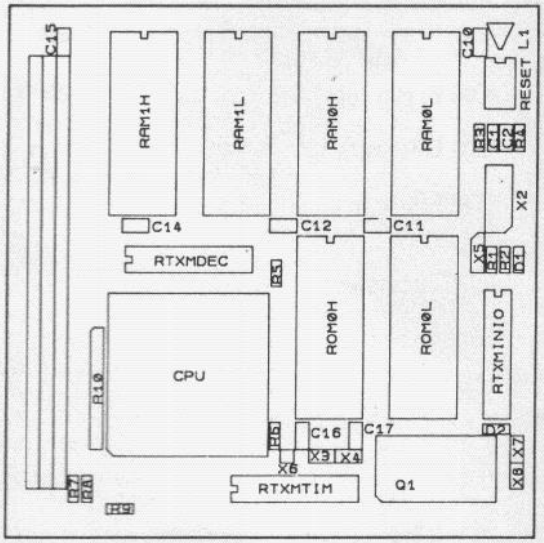
10.000.000 FORTH-Operationen pro Sekunde!

Der REALTIME-EXPRESS läuft weiter !

Die "Emulatorkarte": sämtliche Signale des RTX2000 auf einer 96-poligen VG-Leiste. Serielle Softwareschnittstelle, Systemtakt 6,7 MHz, 64 kByte RAM bestückt, erweiterbar on Board auf 128 kByte, FG-FORTH, kompaktes System 100mm x 100mm durch PALs

Die mc-RISC-Karte: vorgestellt in mc 6/89, 16 Bit Input-Port, 16 Bit-Output-Port, 16 Bit I/O-Bus, serielle Softwareschnittstelle, 8 MHz Systemtakt, 64 kByte RAM bestückt, on Board auf 128 kByte erweiterbar, FG-FORTH, Format 100mm x 160mm.





SONDERPREIS für Mitglieder der FORTH-Gesellschaft:

DM 1.777,--

(incl. Versand, incl. MWSt)

Hochschulrabatt auf Anfrage!

FORTH

REAL TIME EXPRES und RTX ist TradeMark der HARRIS CORPORATION, Palm Bay, Florida

RTX2000

Karten zum Einsteigen erhalten Sie ab sofort bei:
BRÜHL ELEKTRONIK ENTWICKLUNGS-GESELLSCHAFT mbH, Hegelstraße 10, 8500 Nürnberg 10, Tel. 0911/359088
RTX2000, 64 KB PROM, 64 KB RAM, FG-FORTH-Compiler: DM 2.000,–

EDITORIAL

Trotz Umzugsstreß haben wir uns bemüht, auch die vorliegende Ausgabe der 'Vierten Dimension' aktuell und informativ zu gestalten.

Leider lag uns bis Redaktionsschluß nur ein einziger(!) Leserbrief vor. Wahrscheinlich liegt das am vielzitierten Sommerloch. Wir hoffen aber, daß sich in der nächsten Nummer wieder interessierte Leser zu Wort melden. Schließlich lebt eine Zeitschrift nur durch ihre Leser.

In dieser Ausgabe finden Sie einen weiteren interessanten Artikel zur UPN-Diskussion, ein nettes Mandelbrotprogramm, den zweiten Teil des CASE-Artikels und viele weitere interessante Beiträge.

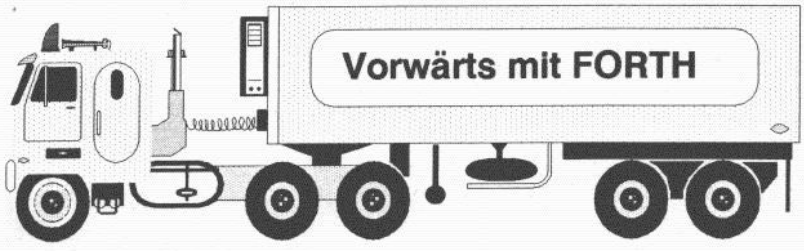
Viel Spaß bei der Lektüre wünscht Ihnen die Redaktion der 'Vierten Dimension'

Rainer Aumiller

Denise Luda

P.S.: Wie oben schon angedeutet, sind wir umgezogen. Die neue Adresse der Redaktion lautet:

D. LUDA Software
Gustav-Heinemann-Ring 42
8000 München 83
Tel.: 089/670 83 55



IMPRESSUM

Titel:
FORTH MAGAZIN 'Vierte Dimension'
Zeitschrift der Mitglieder der FORTH-Gesellschaft e.V. © 1989

Herausgeber:
FORTH-Gesellschaft e.V.

Redaktion:
D. LUDA Software, Gustav-Heinemann-Ring 42, 8000 München 83, Tel. 089/670 83 55

Kontaktadresse:
Entweder direkt die Redaktion anrufen bzw. anschreiben, das FORTH-Büro in München, Postfach 1110, 8044 Unterschleißheim, Tel.: 089/3173784 kontaktieren oder die FORTH-Mailbox München (s.u.) 'Konferenz Vierte Dimension' benutzen.

Quelltextservice:
Der Quelltext von Beiträgen, die mit diesem Symbol gekennzeichnet sind, ist auf der Leserservice-Diskette zur jeweiligen Ausgabe oder in der FORTH-Mailbox in München Tel. 089/7259625 8N1 zu finden.

Quelltext Service

Autoren dieser Ausgabe:
Jörg Staben, Friederich Prinz, Wolf Wejgaard, Jörg Plewe, Christoph Krinninger, Frank Stüss, Stefan Kempf, Michael Sundermann, Konrad Scheller.

Erscheinungsweise:
Vierteljährlich

Redaktionsschluß:
Die zweite Woche im mittleren Quartalsmonat

Auflage:
ca. 1000 Stück

Druck:
Buch- und Offsetdruckerei Bickel Söhne, Frankfurter Ring 243, 8000 München 40

Bezugspreis:
Einzelheft DM 7,50, Abonnement 4 Hefte DM 40,- inklusive Versand.

Vierte Dimension Inhalt

LMI Metacompiler für den Harris RTX-2000

Produktankündigung

Laboratory Microsystems Inc. kündigt eine neue Version des LMI FORTH Metacompilers (Cross-compiler) mit dem Harris RTX-2000 Microcontroller als Target an. Der LMI Metacompiler macht aus jedem IBM PC, PC/AT oder PS/2 kompatiblen ein preiswertes Entwicklungssystem für diese leistungsstarke neue CPU.

Der Harris RTX-2000

Der Harris RTX-2000 ist ein einzigartiger 16-Bit Microcontroller der einen 100 ns Maschinenzyklus bietet und die Ausführung von Anweisungen in nur einem Zyklus ermöglicht. Sowohl Subroutinenaufrufe bzw. -returns wie auch 16-Bit Multiplikationen können in einem Taktzyklus ausgeführt werden. Die Anweisungen des RTX-2000 basieren auf den wichtigsten Elementen der Programmiersprache FORTH, so daß ein Assembler nicht benötigt wird. Desweiteren hat der Chip eine der Programmiersprache FORTH speziell angepaßte dual-starke Architektur.

Wichtige Merkmale des RTX-2000 Microcontrollers sind die on-chip Interruptsteuerung, drei on-chip 16-Bit Counter/Timer und der geringe Leistungsverbrauch (durchschnittlich 5mA/MHz).

Der RTX-2000 ist eine Abwandlung des Novix NC-4000 von Charles Moore, allerdings wurden vorhandene Fehler des Novix Chip herausgefiltert, sowohl die Byte- wie auch die Wortadressierung wird unterstützt und der adressierbare Speicherplatz für Code und Daten liegt bei 1 MB.

Der RTX 2000 ist speziell konstruiert für anspruchsvolle Echtzeitverarbeitungen wie bspw. für Digital- und Imageprozesse, für Roboter, Animation und Simulation. Der ASIC-Bus des RTX-2000 ermöglicht eine Erweiterung der Architektur mit off-chip Beschleunigungslogik und anwendungsspezifischen I/O-Vorrichtungen.

Der RTX-2000 wurde unter Verwendung des Harris Advanced Standard Cell and Compiler Library in CMOS gestaltet und hergestellt. Als Teil der kompatiblen Cell-Library der Harris Familie, kann der RTX-2000 in kundenspezifische ASIC's integriert werden.

Der LMI Metacompiler

Der LMI FORTH-Metacompiler ist ein professionelles Werkzeug für die Anwendungsentwicklung. Er compiliert den FORTH-Quellcode in eine Stand-alone ROM- oder RAM- basierende Applikation. Diese neue Applikation kann jede beliebige Form annehmen, einschließlich die eines neuen interaktiven FORTH Interpreters bzw. Compilers.

Der LMI Metacompiler zeichnet sich durch folgende Merkmale aus:

- tabellengesteuerte Multi-Pass-Compilation
- hervorragendes Fehlerhandling
- unterstützt Local Labels und bedingte Compilationsanweisungen
- ermöglicht die Definition und Invocation neuer Definitionswörter und übernimmt diese direkt in den Target Code
- wahlweise generieren von »Headerless«-Code, um Speicherplatz im Zielsystem zu reservieren
- wahlweise Kompilation von Zwischenergebnissen; d.h. die Applikation kann durch Kompilation der Erweiterungen aus einzelnen »Lagen« aufgebaut werden
- kompatibel mit dem FORTH-83 Standard
- ausführliches 250seitiges Handbuch in englisch
- lizenzfreie Anwendungspakete

Der LMI Metacompiler hat eine ganze Reihe von Merkmalen, die das Programmieren erleichtern. Die herausragendste Eigenschaft ist die Möglichkeit, einen kompletten, interaktiven FORTH Compiler/Interpreter, der auf dem RTX-2000 läuft, zu generieren. Der PC kann Disk Server und Terminal für den RTX-2000 sein, ein neuer Code wird direkt auf dem RTX-2000 entwickelt und interaktiv getestet. Der Metacompiler hat einen tabellengesteuerten Optimizer, der die Opcode-Sequenzen umgehend analysiert, um den schnellstmöglichen und kompaktesten Maschinencode zu generieren. Für den Programmierer läuft dies alles im Hintergrund ab.

Hard- und Software-Anforderungen

Der LMI Metacompiler benötigt einen IBM PC, PC/AT, PS/2 oder kompatiblen PC mit wenigstens 320 KB RAM sowie das Betriebssystem MS-DOS oder PC-DOS 2.0 oder höher. Der Metacompiler mit den zugehörigen Tools und den Quelldateien für das RTX-2000 FORTH Targetsystem beanspruchen ungefähr 400 KB; eine Festplatte wird daher dringend empfohlen.

Die LMI Metacompiler Software ist voll kompatibel mit dem Harris RTX-2000 Entwicklungssystem (die Blue Box), benötigt allerdings keine Harris Software Tools.

Verfügbarkeit des Metacompilers

Der LMI FORTH Metacompiler für den RTX-2000 ist ab sofort im Handel. Der Metacompiler ist entweder auf zwei 5,25" double-sided double-density Disketten oder auf einer 3,5" Microdiskette im IBM PC-DOS Format oder MS-DOS Format erhältlich.

Für die folgenden CPU Targets sind ebenfalls unterschiedliche Versionen des LMI Metacompilers erhältlich: 8080/85, Z-80, HD641180, 80x86/87, 8096/97, 8051/31, 80535, 680x0, 6303, Z-8, 1802, 6502, 68HC11, V25 und TMS 34010.

FORTH SYSTEME
Postfach 1103
Kühnheimerstr. 21
7814 Breisach am Rhein
Tel.: (07667) 551
Fax: 07667555

Wiesel-2000 Entwicklungsboard

Produktankündigung

FORTH-SYSTEME stellt einen modular aufgebauten stand-alone Computer im Europakartenformat vor, der auf dem RTX-2000 Microcontroller von Harris basiert. Wiesel-2000 kann als Entwicklungssystem für den RTX-2000 benutzt werden, stellt aber auch eine gute Lösung für den Einsatz in kleinen oder Prototypstückzahlen dar. Durch die Hochsprachenprogrammierbarkeit **ohne** Leistungsverlust, lassen sich Entwicklungszeiten drastisch verringern.

Der Harris RTX-2000

Der RTX-2000 ist ein RISC-Microcontroller dessen primitiv-Funktionen direkt der Hochsprache FORTH entsprechen. Der RTX-2000 stellt drei 16-Bit-Timer, einen 16x16 Bit Multiplizierer und einen integrierten Interrupt-Controller zur Verfügung. Sowohl Subroutinenaufrufe bzw. -returns, wie auch 16-Bit Multiplikationen können in einem Taktzyklus ausgeführt werden. Als Hochgeschwindigkeitsschnittstelle besitzt der RTX-2000 den 16-Bit breiten ASIC Bus. Mit ihm können bis zu 8/10/12 MWorte/s übertragen werden. Der Chip hat eine der Programmiersprache FORTH speziell angepaßte dualstarke Architektur, d.h. Daten- und Returnstack sind auf dem Chip integriert. Die Befehlsbearbeitung erfolgt in einem Zyklus, wodurch ein 12 MHz System im Durchschnitt 12 Millionen Befehle in der Sekunde bearbeiten kann. Durch parallel ablaufende Operationen kann dieser Durchsatz im Parkbetrieb noch erhöht werden. Die geringe Interruptresponszeit von nur 4 Taktzyklen bringt speziell für die Echtzeitverarbeitung erhebliche Vorteile.

Das Wiesel-2000

Die geballte Rechnerleistung, die auf dieser Europakarte vereint ist, läßt neue Lösungsmöglichkeiten in den Bereichen Messen, Steuern und Überwachen zu.

Das Wiesel-2000 gibt es als 8, 10 und 12 MHz Version. Das Mainboard enthält bei der *Minimal-Konfiguration :*

- 32 Kbyte Highspeed SRAM (4x16Kx4 0 Wait)
- 64 Kbyte EPROM
- RS-232-Schnittstelle (UART 16C450)
- Transputer-Link von INMOS (IMSC012)
- 16-Bit I/O-Port, 8-Bitweise konfigurierbar

Die **Profi Extended Konfiguration** hat anstelle der 32 Kbyte 128 Kbyte SRAM (4x64Kx4 0 Wait).

Als Erweiterungen für die CPU Konfiguration werden zwei RAM-Erweiterungsboards angeboten, die in Stapeltechnik auf das Mainboard gesteckt werden können. Jede dieser Speicherkarten enthält ihre eigene Speicherdekodierung.

Die HS (high speed) RAM-Erweiterung enthält bis zu 512 Kbyte high speed SRAM (16x64Kx4 0 Wait), kann aber auch 16Kx4 0 Wait Bausteine aufnehmen. Mischkonfigurationen sind ebenfalls möglich.

Auf der HD (high density) RAM-Erweiterungskarte ist Platz für 1 MByte RAM. Es können acht 32Kx8 bzw. 128Kx8 Bausteine von ihr aufgenommen werden. Auch hier sind Mischkonfigurationen möglich.

Das Wiesel-2000 zeichnet sich weiterhin durch folgende Merkmale aus:

- flexibles und änderungsfreundliches Timing durch reprogrammierbare GAL's, die in der Lage sind, die Systemclock mit einer Auflösung von 25/21 ns um bis zu 75/63 ns abhängig von der Adresse zu verlängern und/oder bis zu 3 Waitzyklen zu produzieren.
- Wire-Wrap-Feld für beliebige Verbindungen der externen Interrupts mit RTX-2000.
- Anwendungssoftware kann interaktiv über die serielle Schnittstelle oder den Link entwickelt, sofort getestet und in neue EPROM's gebrannt werden.
- Unterstützung der Fehlersuche durch einen integrierten Decompiler und Single-Step Tracer.
- Optimierender Compiler standardmäßig im EPROM enthalten
- ausführliches deutschsprachiges Handbuch

Hard- und Software-Anforderungen

Als Host-Rechner kann jeder Rechner verwendet werden, der über eine serielle Schnittstelle verfügt; bei IBM PC oder kompatiblen Rechnern kann die mitgelieferte Treibersoftware verwendet werden.

Die EPROM's des Wiesel-2000 enthalten ein auf den RTX-2000 abgestimmtes LMI FORTH, das dem FORTH-83 Standard entspricht. Die EPROM-Software wurde mit dem optimierenden LMI-Metacompiler auf dem PC entwickelt, dieser Metacompiler steht auch für Ihre eigene Projektentwicklung oder für die Modifikation der Basissoftware zur Verfügung.

Die Verfügbarkeit des Wiesel-2000

Das Wiesel-2000 ist ab sofort im Handel. Das HS-Board ist standardmäßig mit vier 16Kx4 Bausteinen, das HD-Board mit zwei 32Kx8 Bausteinen ausgerüstet.

FORTH SYSTEME
Postfach 1103
Kühnheimerstr. 21
7814 Breisach am Rhein
Tel.: (07667) 551
Fax: 07667555

LESERBRIEF

Optimierung

Sehr geehrte Damen und Herren, ich suche ein in FORTH geschriebenes Programm für die Optimierung. Ich denke dabei an lineare Programmierung basierend auf der Simpler Methode

H.C. Overbeek
Stephensonstraat 14
NL 7553 TB Hengelo - Holland

Treffen der lokalen Gruppe Westberlin (i.A.) der FORTH-Gesellschaft für 1989/90:

Grüße aus der Mauerstadt an alle Wessis:

Wir möchten Euch auch ganz herzlich einladen, mal bei uns vorbeizuschauen. Wenn Ihr mal nach Berlin kommt, würden wir am liebsten gleich noch einen Vortrag von Euch hören. Zu folgenden Themen suchen wir am dringendsten Referenten: Target-Compilation (Bernd, Du hast es versprochen!), Leibniz/Hypertext (na, Andreas?), Benutzungs-Oberfläche, RTX 2000: Erfahrungen im kommerziellen Einsatz, gute Dokumentation (was ist das?, automatische Erstellung). Im Austausch haben wir Spezialisten für 8051, 68HC11, FORTH-Musik, OOP, und eine volksFORTH-Herkules-Grafik. Claus (030/216 89 38), Helge (859 17 54)

Do, 28.09.89
Asyst von Keithley - Hall (Keithley, Köln), Döring (Keithley, Berlin)

Asyst ist ein kommerzielles Programmpaket zur Erfassung, Aufbereitung und grafischen Präsentation von Meßdaten. Es ist komplett in FORTH geschrieben und kann um anwendungsspezifische Programme erweitert werden. Vertreter der Firma Keithley Instruments tragen vor.

Do, 26.10.89
FORTH macht Musik - Marcus Verwiebe (TU Berlin)

Im Rahmen des Projektes CAMP (Computer Aided Music Processing) wurden in den vergangenen zwei Jahren experimentelle Arbeiten auf einem 32-Bit-FORTH-System (Atari ST) durchgeführt. Ausgangspunkt der Entwicklung war das Computer-Musiksystem FORMULA (FORth MUsic LAnguage). Es bietet Multitasking mit zeitkorrektem Scheduling von Ereignissen und Möglichkeiten zur interaktiven Steuerung und Programmierung von elektronischen Klangerzeugern kompatibel zum MIDI Standard (Musical Instrument Digital Interface).

Do, 30.11.89
FORTH lernen - Gespräch mit Hans-J. Thiess

Die Probleme von FORTH-Anfängern unterscheiden sich von jenen bei anderen Programmiersprachen. Um uns über diese Probleme auszutauschen, halten wir die Form des Vortrags für ungeeignet. Stattdessen fordern wir alle Anfänger und Fortgeschrittenen auf, Lehrbücher und Material mitzubringen, das sie für geeignet (oder ungeeignet) halten und über ihre Erfahrungen beim Lernen der Programmiersprache FORTH zu berichten. Hans-J. Thiess, Autor mehrerer Lehrbücher zu verschiedenen Programmiersprachen und Leiter von Kursen, wird anwesend sein. Das Gespräch könnte auch der Ausgangspunkt eines FORTH-Kurses sein.

Do, 21.12.89
-- Weihnachtskränzchen --

Anläßlich der Jahreszeit diesmal nicht am letzten Donnerstag im Monat und ohne Thema. Zeit: 19.30 Uhr. Ort: Helge Horch, Dickhardtstr. 28, 1/41, 859 17 54 Bitte Plätzchen, Weihnachtsstollen etc. selbst mitbringen.

Do, 25.01.90
FORTH-System im Selbstbau - Frank Wilde

Ein Grund für die Vielfalt der FORTH-Dialekte ist sicherlich die vergleichsweise einfache Implementation eines FORTH-Interpreter/Compilers, der so auf die eigenen Bedürfnisse abgestellt werden kann. Über Konzepte, Lust und Frust bei der Erstellung eines eigenen FORTH-Systems berichtet Frank Wilde, Autor eines 32-Bit-jumpSubroutine-token-threaded FORTH-Systems für Atari ST.

Do, 22.02.90
FORTH-Standard - Marcus Verwiebe, N.N.

Die Standardisierung von FORTH wird oft herbeigesehnt und ebenso oft verteufelt. Freunde wie Feinde der Standardisierung führen hierbei mit guten Gründen das Argument erhöhter Leistungsfähigkeit ins Felde. Nach dem FIG-FORTH-(Quasi)-Standard der FORTH Interest Group (USA) der 70er Jahre, der mittlerweile allgemein als überholt angesehen wird, folgte in kurzem Abstand der Europäische 79er- und der US-Amerikanische 83er-Standard der sich bisher als der Tragfähigste erwiesen hat. Heute richtet sich die Aufmerksamkeit auf die Arbeit des ANSI-FORTH-Standard-Komitees, über die es in Europa allerdings bislang wenig Informationen gibt. Wir wollen über Entwicklung und aktuellen Stand berichten.

Do, 29.03.90
OOF - Object oriented FORTH - Helge Horch, N.N

FORTH eignet sich weit besser als herkömmliche Sprachen zur Integration neuartiger Konzepte - läßt sich in dieser Hinsicht noch am ehesten mit Lisp vergleichen. Die Philosophie des Objektorientierten Programmierens, wie sie mit Simula-66 begann und in Smalltalk-80 einen - wars denn '66 ? - neuen Höhepunkt erreichte, hat verschiedene Autoren zum Erstellen objektorientierter Erweiterungen angeregt. (z.B. Objective-C, C++). In der FORTH-Community ist hier insbesondere der Ansatz von Dick Pountain zu nennen. Helge Horch hat Pountains Aufsatz auf das volksFORTH83 angepaßt und uns bereits im Juni/88 darüber berichtet. Auf neue Ergebnisse sind wir gespannt.

Do, xx.04.90
FORTH als Produktivkraft - Gerd Blanke (Fa. Microtaurus, Berlin)

Warum benutzen professionelle Software-Entwickler FORTH? Über die Stärken - und auch die Schwächen - der Sprache und ihrer verschiedenen Programmierumgebungen bei der Erstellung von Anwendungen nicht nur der Meß- und Steuertechnik wird Gerd Blanke anhand langjähriger Erfahrungen seiner Firma vortragen.

Zeit, sofern nicht anders angegeben: 19.30 Uhr an jedem letzten Donnerstag im Monat.

Ort, sofern nicht anders angegeben: TU Berlin MA 621. Straße des 17.Juni Mathematikgebäude 6.Stock U-Bhf Ernst-Reuther-Platz

Neuheit

SILICON COMPOSERS stellt weitweit den ersten 32-Bit FORTH-Mikroprozessor "SC32 Stack-Chip" vor.

SC32 Stack-Chip Hardware-Features:

- 32-Bit CMOS Mikroprozessor, 34.000 Transistoren.
- 32-Bit Adress- und Datenbus, kein Multiplexbetrieb.
- Ausführung einer Instruktion innerhalb eines Taktzykluses (10 MIPS).
- Speicherzugriff innerhalb zwei Taktzyklen.
- durchgehender 16 Gigabyte Datenspeicher.
- durchgehender 2 Gigabyte Codespeicher.
- 8 oder 10 Mhz Systemtakt.
- 2 zirkulare Stack-Cache-Speicher (16 Elemente).
- Stacktiefe allein durch den vorhandenen Speicher begrenzt.

SILICON Composers, Inc.
414 California Avenue
Palo Alto, CA 94306 USA

Kurznachricht

Die FORTH-Tagung '90 und die ordentliche Mitgliederversammlung der FORTH-Gesellschaft e.V. findet im April 1990 in Frankfurt statt.

Buchbesprechung

Manfred Mader: "Einführung in FORTH-83"

von Christoph Krinninger

Im Heim-Verlag, der insbesondere Bücher für Atari ST veröffentlicht, ist vor kurzem ein umfassendes Werk über das volksFORTH erschienen. Hinter dem unscheinbaren Titel verbirgt sich nicht nur ein Buch für Anfänger, sondern das bisher detaillierteste und vollständigste Handbuch für das volksFORTH, das auch noch dem Fortgeschrittenen etliche Implementationsdetails des volksFORTH veranschaulicht. Auf etwa 600 Seiten ist das Innere von FORTH so präzise beschrieben, daß man eine Neuimplementation spielend meistern kann. Da die volksFORTH Versionen für die unterschiedlichen Betriebssysteme und Rechner weitgehend identisch aufgebaut sind, bietet dieses Buch nicht nur Atari ST Besitzern Wissenswertes.

Auf den ersten 200 Seiten wird dem Anfänger eine Einführung in FORTH an vielen kleinen Beispielen und Illustrationen gegeben. Die Beschreibung des äußeren Interpreters, die Demonstration des Stack-Mechanismus, die Verwendung von mathematischen und Stringoperatoren und vieles andere mehr versetzen den Anfänger in die Lage, selbstständig in FORTH zu programmieren.

Der weit größere Teil des Buches beschäftigt sich mit den internen Details des volksFORTH. Angefangen vom Speicheraufbau, über den Wortaufbau bis zu komplizierten Themen, wie dem Heap und headerlosen Worten, wird auf weiteren 200 Seiten jedes noch so heiße Eisen bis auf Byte-Ebene erklärt. Besonders möchte ich das Kapitel über den Interpreter und Compiler erwähnen, in dem anhand des Sourcecodes des volksFORTH Wort für Wort die Funktion und Wirkungsweise dieser beiden zentralen Institutionen durchgekaut werden. Wer das Buch bis zu diesem Teil verstanden hat, ist durchaus in der Lage, selbst FORTH-Systeme zu implementieren. Es schließen sich Kapitel über das File-Interface, den Multitasker, den Decompiler und Tracer sowie den übrigen Files auf den volksFORTH-Disketten an. Natürlich wird jeder größere Abschnitt mit einem Aufgaben- und Beispielekapitel abgeschlossen, an dem man das Gelernte in der Anwendung sehen kann.

Im letzten Teil folgen die Abschnitte FORTH-Assembler, Line-A Grafik und nicht zuletzt die GEM-Programmierung. Diese Kapitel fallen etwas magerer aus, sind aber so gehaltvoll, daß sie als umfangreiches Nachschlagewerk für den Fortgeschrittenen dienen können. Hier sei für Anfänger dieser Atari ST spezifischen Themen auf ergänzende Begleitlektüre verwiesen. Ebenfalls etwas knapp fällt das Stichwortverzeichnis aus, es beschränkt sich auf die FORTH-Worte; hier ist für eine Neuauflage noch viel Platz für Ideen.

Fazit: Dieses Buch füllt endlich die Lücke zwischen den Werken von Brodie und Zech.Es ist sowohl für Anfänger, Fortgeschrittene und solche, die es werden wollen, geeignet. Für Implementierer bietet es endlich neben dem Sourcecode des volksFORTH auch eine Beschreibung der manchmal nicht so trivialen Innereien. Betrachtet man den Preis des Buches, dem auch eine Diskette mit Beispielen und einigen nützlichen Hilfsprogrammen beiliegt, so bekommt man mit dem public-domain volksFORTH ein Werkzeug in die Hand, an dem sich selbst kommerzielle FORTH-Systeme sowohl in Ausstattung, Dokumentation als auch in der Leistung messen müssen. Es sollte bei keinem ernsthaften volksFORTH Programmierer im Regal fehlen. Der Titel "volksFORTH Profibuch" wäre durchaus angemessen.

"Einführung in FORTH-83",
Manfred Mader, Heim Verlag,
Darmstadt 1989, Bestell-Nr. B-419
ISBN 3-923250-69-X, Preis DM 54.-
(incl. Diskette)

Warum Postfix?

Wolf Wejgaard,
Neuhöflirain 10,
CH-6045 Meggen

Wer sich mit FORTH beschäftigt, muß in Kauf nehmen, daß er »anders ist, als die anderen«. Außerhalb der FORTH-Gemeinschaft befindet er sich vorwiegend in Verteidigungsposition und muß erklären, warum er sich so komischen Schreibweisen freiwillig unterwirft, wie der umgekehrten polnischen Notation (UPN).

Ich habe lange gebraucht, um eine gute Antwort zu finden. Das Problem ist in Wirklichkeit, daß die »Anderen« nicht die UPN verwenden!

Soweit zur Polemik, aber jetzt sachlich:

Ausgangspunkt ist die Frage, wie man am einfachsten mit einem Computer umgeht, d.h. wie man direkt und klar, mit einem Minimum an (überflüssigem) Formalismus dem Computer sagt, was er tun soll.

Auf der untersten Ebene stehen die Maschinenbefehle zur Verfügung. Diese sind einfach und klar, doch wird es bald mühsam, genügend Maschinenbefehle aneinanderzureihen, bis der Computer etwas vernünftiges macht, und daher wurde die *Subroutine* erfunden. Jeder Assembler-Programmierer hat sehr bald eine Sammlung nützlicher (Sub-) Routinen zur Hand. Und eine geniale Menge solcher Routinen bilden bekanntlich die Grundlage der FORTH-Maschine. Aus bestehenden Routinen können dann wieder neue gebildet werden.

Dieses Zusammenfassen von Grundbefehlen und Strukturen zu Routinen und weiteren darüber liegenden Schichten von Routinen, ist ein Merkmal jeder vernünftigen Programmiersprache. In FORTH reden wir von (und in) »Worten«.

Der *Unterschied* der Programmiersprachen beginnt mit der Gretchenfrage, die da lautet: Wie hast Du's mit den *Argumenten*? Eine Routine ist selten nützlich, wenn man ihr nicht Daten zum Bearbeiten übergeben kann, und meist auch neue Daten geliefert bekommt. In der Technik der Übergabe von »Argumenten« unterscheiden sich die Sprachen. Man kann es beliebig kompliziert und sehr einfach machen. Der einfachste Mechanismus ist ein *Stack*.

Ein Datenstack hat verschiedene Vorteile:

- lokale und globale Daten sind automatisch getrennt (lokale Daten werden auf dem Stack übergeben, sind also automatisch nur für die beteiligten Worte sichtbar; globale Daten sind mit einem Namen definiert).
- er erlaubt Rekursion (eine Routine kann sich selbst aufrufen).
- der Stackmechanismus ist sehr einfach zu implementieren (viele Prozessoren haben den Mechanismus eingebaut).
- Routinen können interaktiv getestet werden: Daten auf den Stack legen, Routine rufen und Ergebnis vom Stack lesen (darum ist der FORTH-Interpreter so einfach zu realisieren).

Eine direkte Folge der Übergabe von Argumenten auf einem Stack ist die UPN, auch *Postfix* genannt. Das heißt nicht, daß wir es dabei belassen müßten; mehrere FORTH-Programmierer haben Infix-Pakete publiziert, die also eine »normale« algebraische Notation zulassen. Wir haben die *Wahl*. Warum bleiben wir bei Postfix?

Ich möchte hier die Argumentation von Jörg Plewe aus VD 5/2 (Juni 89) Seite 37 aufgreifen und ergänzen.

```
2 + 3 ist klar, aber
was ist 2 + 3 * 4 ?
```

Wir brauchen offensichtlich eine *Zusatzinformation*, nämlich eine *Präzedenzregel*, die festlegt, welche Operatoren Vorrang haben. Alternativ können wir die Zusatzinformation auch als *Klammernotation* einführen, und bekanntlich werden beide Arten verwendet. Da wir solche Ausdrücke wie (2 + 3) * 4 dem Computer präsentieren, muß auch er, bzw. sein Compiler, diese Zusatzinformation kennen. Der Compiler ist also eine Spur komplexer als notwendig, denn offensichtlich könnten wir auf die Zusatzregeln verzichten, indem wir 2 3 + 4 * schreiben.

Die Frage, ob Postfix oder Infix, läßt sich damit auf die Entscheidung zurückführen, wie kompliziert wir unser Leben gestalten wollen. Postfix entspricht dem Wunsch nach einer *einfachen Technik*, und ist mit ein Grund dafür, daß der FORTH-Compiler so wunderschön einfach ist.

Das zentrale Wort heißt »einfach«. Und das hat nichts mit »simpel« zu tun. »Einfach« ist die positive Eigenschaft, mit einem Minimum an Mitteln sein Ziel zu erreichen. Es ist das »small is beautiful«, es ist mit Eleganz verbunden - und nicht leicht zu erreichen. Wer sich zu FORTH hingezogen fühlt, hat eine Ader für diese Denkart.

Aber es bleibt der Zweifel: alle anderen arbeiten mit Infix, da muß doch was dran sein! Halten wir endlich einmal fest, daß Infix nur im Bereich der *Mathematik* eine Rolle spielt! Außerhalb der Mathematik, im »richtigen Leben«, sind wir im allgemeinen postfix-orientiert: Wer einen Brief schreiben will, nimmt Papier und Schreibstift zur Hand (Operanden) und beginnt dann mit dem Schreiben (Operator). Wer einen Nagel einschlagen will, nimmt erst Brett und Nagel (Operanden) zur Hand, bevor er den Hammer (Operator) schwingt. Oder rührt jemand seinen Tee, bevor der Zucker drin ist? Auch der zentrale Prozessor unseres Computers braucht zuerst die Operanden in den jeweiligen Registern, bevor er seine Operation ausführen kann.

Und: eine der ersten Operationen eines normalen Compilers ist das Umwandeln des Programmes in eine Stacknotation, also in Postfix. Denn die meisten Programme werden effektiv über Stacks verarbeitet! Oder nicht?

Stichworte

» UPN,
» Infix, Postfix,
» Stacknotation

Zum Schluß möchte ich die Verständlichkeit als Zeugen bemühen. Ein gut zerlegtes Postfix-FORTH-Programm liest sich wie ein klarer Bericht. Beispiele dazu liefert die FORTH-Literatur zur Genüge. Ich finde auch den Gegenschluß nützlich: wenn sich mein Programm nicht gut liest, muß ich es nochmal überarbeiten.

Der Hang zur Komplexität, ist ein Übel, das ich auch bei mir immer wider entdecke. Da tut es mir jeweils gut, die Ideen von Charles Moore wiederzulesen, wie z.B. im Artikel in VD 3/3 (Oktober 87) Seite 11, den ich heute zufällig wieder entdeckt habe. Auch Leo Brodie möchte ich empfehlen, in diesem Zusammenhang sein Buch "In FORTH denken". (Warum wird Brodie so selten erwähnt?)

Wolf Wejgaard
Dr.sc.nat.dipl.Phys.ETH
Heuhöflirain 10
CH-6045 Meggen

Mandelbrot für das volksFORTH

Quelltext Service

Stichworte

- Mandelbrot,
- volksFORTH,
- komplexe Zahlen

von Christoph Krinninger

Eine der populärsten Computer-Grafiken ist das sogenannte Apfelmännchen, das mit einem Algorithmus des französischen Mathematikers Benoit Mandelbrot berechnet wird. Jeder Punkt der Grafik wird nach der iterativen Formel

```
Z = Z * Z  +  C
```

berechnet. Die komplexe Zahl Z wird mit sich selbst multipliziert und zu einer weiteren komplexen Zahl C addiert. Diese Formel wird mehrmals wiederholt und getestet, ob die Größe der Zahl Z den Wert 2 überschreitet. Die Iterationstiefe wird dann als »Höhe« oder häufig als Falschfarbe dieses Punktes verwendet.

Der Startwert für den Real- und Imaginärteil der Zahl Z sind die x- und y-Koordinaten des zweidimensionalen Grafikausschnittes. Diese Startkoordinaten heißen im abgedruckten FORTH-Programm CX und CY, jeder Wert ist um den Faktor 8192 vergrößert, -1 lautet also -8192, 0.02 besitzt den Wert 164 usw. Um verschiedene Ausschnitte der Grafik betrachten zu können, muß man also die Startwerte für CX und CY (bzw. CYBASE) verändern. Nach jedem Durchlauf entlang der Y-Achse wird CY auf den Wert in CYBASE zurückgesetzt. Die Schrittgröße wird in CXSTEP und CYSTEP festgelegt, um also große Teile des Apfelmännchens betrachten zu können, muß diese groß gewählt werden, um einen kleinen Ausschnitt vergrößert betrachten zu können, ist die Schrittweite entsprechend kleiner zu wählen. Das besondere an diesem Programm ist die Tatsache, daß vollständig auf ein Floating-Point Paket verzichtet werden kann. Die beiden Zahlen auf dem Stack zu Beginn der inneren Schleife sind der Imaginär- und Realteil der Zahl Z. Diese Zahl wird quadriert zu C addiert, eine quadrierte komplexe Zahl ist aber immer negativ. Der Realteil von Z wird folgendermaßen berechnet:

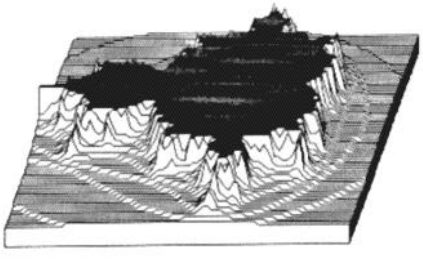


Bild 1

```
Zreal = Zreal * Zreal -
Zimag * Zimag + CX
```

Für den Imaginärteil gilt:

```
Zimag = 2 * Zreal *
Zimag + CY
```

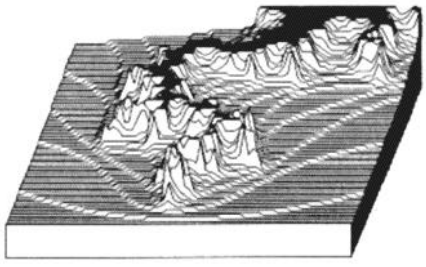


Bild 2

Die Größe einer komplexen Zahl entspricht der Hypotenuse eines Dreieckes mit den Seiten Zreal und Zimag.

```
Mag * Mag = Zreal *
Zreal + Zimag * Zimag
```

Ist das Ergebnis größer als 2, wird Z weiter ansteigen und daher die Schleife abgebrochen. Es ist aber einfacher, das Quadrat der Größe von Z zu testen, also bei dem Wert 4 abzubrechen. Wenn man alle Werte um den Faktor 1000 vergrößert, muß man also bei 4000 abbrechen, beim Faktor 8192 ist der Testwert 32768. Stattdessen kann man bei 16-Bit Zahlen mit Vorzeichen auch prüfen, ob die Zahl negativ geworden ist. Das Programm wurde wie immer in volksFORTH für den ATARI ST geschrieben.

Die wesentlichen Grafikbefehle stammen aus dem VDI-Paket, näheres ist der Fachliteratur zu entnehmen. Wie in der letzten Ausgabe ist folgendes zu beachten: Das Wort COORDINATES legt je nach Auflösung weit über 100 Parameter auf den Stack. Die normale Stacktiefe ist je-

doch teilweise nicht auf so extensive Benutzung ausgelegt. Beim volksFORTH kann die Größe des Stacks mit dem Wort RELOCATE (in RELOCATE.SCR) festgelegt werden. Man gibt am besten folgendes ein:

```
INCLUDE RELOCATE.SCR
<return>
R0 @ 500 RELOCATE
<return>
```

Sollte nicht genügend Speicherplatz vorhanden sein, so kann man mit dem Wort BUFFERS, ebenfalls in RELOCATE.SCR, die Zahl der Disk-Buffer verkleinern. Ebenso muß das Punktearray für das VDI vergrößert werden. Ändern Sie in GEM\BASICS.SCR, SCR #2, Zeile 2 den Ausdruck

```
CREATE PTSIN &60 ALLOT
```

in

```
CREATE PTSIN &256 ALLOT
```

Anschließend muß das FORTH-System neu zusammengestellt werden. Der Sourcecode zu diesem Artikel kann über den Diskettenservice oder die FORTH-Mailbox FBM bezogen werden.

Bibliographie

[1] FORTH News, Martin Tracy, Dr. Dobb's Journal, February 1989, S. 136 ff

Screen # 0

```
\ Mandelbrot                                              03sep89 ck

Die wohl berühmteste Computer-Grafik ist das sogenannte
Apfelmännchen nach einem Algorithmus des französischen
Mathematikers Benoit Mandelbrot.
```

Screen # 1

```
\ Loadscreen                                              03sep89 ck

Onlyforth gem also      \needs pline     2 loadfrom vdi.scr
Onlyforth gem also      \needs overwrite 8 loadfrom vdi.scr

Onlyforth gem also

decimal

\needs it       : it ;
forget it       : it ;


2 14 thru
```

Screen # 2

```
\ Startwerte                                              03sep89 ck

Variable cx     Variable cy     -10000 Constant cy.base

cy.base cy !    -13000 cx !     15 Constant maxloop

250 Constant cx.step            250 Constant cy.step

Variable x      Variable y      Variable z

80 Constant size        Create square   size size * allot

20 Constant xoffset     20 Constant yoffset
```

Screen # 3

```
\ Punkte setzen oder löschen                              03sep89 ck

: pixel-on      ( x y -- )
        1 sl_color
        2dup 2 pline ;

: pixel-off     ( x y -- )
        0 sl_color
        2dup 2 pline ;

: set.pixel     ( n n I -- )
        X @ xoffset +
        Y @ yoffset +
        pixel-off
        square Y @ size * X @ + + c!
        2drop ;
```

Screen # 15

```
\ Bibliografie                                            03sep89 ck

The FORTH Column
Martin Tracy
Dr. Dobb's Journal, February 1989
S. 136 ff

Geeignete Testwerte für Bildausschnitte:

CY.BASE         CX        CX.STEP  CY.STEP  MAX.LOOP
-10000         -13000       250      250      15
  2200         -10100        10       10      22
  2200          -9700        10       10      22
  2000          -9400        10       10      22
  2500          -8800        10       10      22
  2000          -8000        10       10      22
```

Screen # 16

```
\                                                         03sep89 ck

Aus dem VDI-Packet des volksFORTH werden die Ausgabe- und
Attributfunktionen benötigt.



Kleiner Trick, um wiederholt compilieren zu können.
```

Screen # 17

```
\                                                         03sep89 ck

CX CYBASE CY    Startwerte für Ausschnitt aus dem Apfelmännchen
MAXLOOP         Maximale Iterationstiefe


CX.STEP CY.STEP Schrittweite

X Y X           Hilfsvariablen für Koordinaten

SIZE            Seitengröße des Bildquadrates
SQUARE          Feld für Grafik
XOFFSET YOFFSET Offset des kleinen Windows auf dem Bildschirm
```

Screen # 18

```
\                                                         03sep89 ck

PIXEL-ON        Pixel setzen


PIXEL-OFF       Pixel löschen


SET.PIXEL       Pixel löschen und "Höhe" im Bildarray speichern
```

Mandelbrot für das volksFORTH

Screen # 4

```
\ Eigentliche Hauptschleife                              03sep89 ck

: inner.loop    ( zreal zimag -- zreal.new zimag.new )
        maxloop 0 DO
                2dup dup abs 16384 > IF I set.pixel LEAVE THEN
                dup 8192 */
                swap dup abs 16384 > IF I set.pixel LEAVE THEN
                dup 8192 */
                2dup +              0< IF I set.pixel LEAVE THEN
                swap -
                cx @ +
                -rot
                4096 */
                cy @ +
        LOOP ;
```

Screen # 5

```
\ Zeichenroutine                                         03sep89 ck

: draw.mandelbrot ( -- )
        size 0 DO       ( x-axis on screen )
                I x !
                size 0 DO       ( y-axis on screen )
                        0 0
                        J xoffset + I yoffset + pixel-on
                        I y !
                        inner.loop
                        2drop
                        cy.step cy +!
                LOOP
                cy.base cy !  cx.step cx +!
                stop? IF LEAVE THEN
        LOOP ;
```

Screen # 6

```
\ Diverses für 3-D Grafik                                03sep89 ck

: init.square   ( -- )
        square size size * maxloop fill ;

5 Constant x-scale      3 Constant y-scale

Variable arrayoffset
Variable x-shift        Variable y-shift

&150 Constant y-screenoffset   &200 Constant x-screenoffset
x-screenoffset size x-scale * +       Constant x-offsetbase
y-screenoffset &10 +                Constant y-offsetbase

Variable lastpoint      Variable lastpoint'
```

Screen # 7

```
\ Offsetberechnung für 3-D Grafik                        03sep89 ck

-1 Constant x-screenshift
 2 Constant y-screenshift

: x-shift+      ( x -- x )      x-shift @ + ;

: y-shift+      ( y -- y )      y-shift @ + ;

: x-offset      ( -- x )
        x-screenoffset x-shift+ ;

: y-offset      ( -- y )
        y-screenoffset y-shift+ ;
```

Screen # 8

```
\ Diverse Parameter für die Ausgabe                      02sep89 ck

: x-offbase     ( -- x )
        x-offsetbase x-shift+ x-scale - ;

: y- ffbase     ( -- y )
        y-offsetbase y-shift+ ;

: basement      ( x1 y1 .. xn yn n -- x1 y1 .. xm ym m )
        >r
        x-offbase   y-offset    x-offbase   y-offbase
        x-offset    y-offbase   x-offset    y-offset
        r> 5 + ;
```

Screen # 19

```
\                                                        03sep89 ck

INNER.LOOP      Eigentliche Hauptschleife
                Z = Z * Z + C
                   Wenn Überlauf bei /*, dann sofort Abbruch
                Quadrat des Imaginärteils, skaliert mit 8192
                   Wenn Überlauf bei /*, dann sofort Abbruch
                Quadrat des Realteils, skaliert mit 8192
                Quadrat der Größe von Z größer 4 ?
                Realteil von Z-Quadrat
                CX Realteil von C

                2*Zreal*Zimag, skaliert mit 8192 (2/8192=1/4096)
                CY Imaginärteil von C
```

Screen # 20

```
\                                                        03sep89 ck

DRAW.MANDELBROT         Kleines Apfelmännchen in
                        Kontrollfenster auf dem Bildschirm
                        zeichnen


                        Pixel in Kontrollfenster setzten

                        Eigentliche Hauptroutine
```

Screen # 21

```
\                                                        03sep89 ck

INIT.SQUARE     Array für Grafik löschen


X-SCALE         Vergrößerungsfaktoren für 3-D Grafik
Y-SCALE

ARRAYOFFSET     Hilfsvariable
X-SHIFT Y-SHIFT Versatz der einzelnen Segmente

                Koordinaten für Sockel in 3-D Grafik

LASTPOINT       Hilfsvariablen, um Vektoren zu komprimieren
```

Screen # 22

```
\                                                        03sep89 ck

X-SCREENSHIFT   Versatz der Segmente in 3-D Grafik
Y-SCREENSHIFT
```

Screen # 23

```
\                                                        03sep89 ck


BASEMENT        Koordinaten des Sockels
```

Screen # 9

```
\ Komprimierung der Vektoren                                    03sep89 ck

: compact.coords        ( x1 y1 n x2 y2 -- x2 y1 n )
        drop swap 2swap -rot drop ;

: new.coordinates       ( x1 y1 n x2 y2 -- x1 y1 x2 y1 n+1 )
        lastpoint @ lastpoint' !
        dup lastpoint !  rot 1+ ;

: y-coordinate  ( n -- y )
        y-scale * y-offset swap - ;

: y-max.coord   ( -- ymax )
        maxloop y-coordinate ;
```

Screen # 10

```
\ Koordinatenermittlung                                         02sep89 ck

: coordinates   ( -- x1 y1 .. xn yn n )
        x-offset   y-offset     ( first point )
        lastpoint on lastpoint' on
        0
        size 0 DO
                I x-scale * x-offset +  ( x )
                arrayoffset @  I + c@
                y-coordinate ( y )
                dup lastpoint @ =  over lastpoint' @ = and
                IF   compact.coords
                ELSE new.coordinates
                THEN
        LOOP basement ;
```

Screen # 11

```
\ Linie oder Punkt zeichnen                                     03sep89 ck

Variable loop.counter

: draw.line     ( x1 y1 x2 y2 -- )
        1- swap 1+ swap
        2swap 1- swap 1+ swap 2swap
        2 pline
        -2 loop.counter +! ;

: draw.point    ( x y -- )
        1- swap 1+ swap
        2dup 2 pline
        -1 loop.counter +! ;
```

Screen # 12

```
\ Maximalhöhe verstärkt zeichnen                                03sep89 ck

: outline.surface ( -- )
        coordinates loop.counter !
        BEGIN
                dup y-max.coord =
                IF
                        2over nip  y-max.coord =
                        IF draw.line ELSE draw.point THEN
                ELSE
                        2drop -1 loop.counter +!
                THEN
        loop.counter @ 0=
        UNTIL
;
```

Screen # 13

```
\ Umriß zeichnen                                                02sep89 ck

: draw.surface  ( -- )
        x-shift off  y-shift off
        overwrite       true sf_perimeter       1 sl_color
        1 sl_width      0 sf_interior   0 sf_style
        0 0 sl_ends
        square
        size 0 DO
                dup arrayoffset !
                coordinates fillarea
                outline.surface
                x-screenshift x-shift +!
                y-screenshift y-shift +!
                size +
        LOOP drop ;
```

Screen # 24

```
\                                                               03sep89 ck

COMPACT.COORDS  Koordinaten zusammenfassen

NEW.COORDINATES Neue Koordinaten abschließen

Y-COORDINATE    Höhe in Bildschirmkkordinate umrechnen

Y-MAX.COORD     Maximale mögliche Y-Koordinate
```

Screen # 25

```
                                                                03sep89 ck

COORDINATES     Koordinaten eines 3-D Segmentes

                X-Koordinate ermitteln
                Höhe aus Grafikarray lesen
                Y-Koordinate ermitteln
                ggf. Koordinaten zusammenfassen
```

Screen # 26

```
\                                                               03sep89 ck

LOOP.COUNTER    Hilfsvariable für Pseudo DO..LOOP

DRAW.LINE       Verstärkte Linie zeichnen

DRAW.POINT      Verstärkten Punkt zeichnen
```

Screen # 27

```
\                                                               03sep89 ck

OUTLINE.SURFACE Maximale Werte (=Apfelmännchen) verstärkt
                zeichnen

                Maximaler Y-Wert ?

                Verstärkt zeichnen
```

Screen # 28

```
\                                                               03sep89 ck

DRAW.SURFACE    3-D Segmente zeichnen

                Diverse VDI-Parameter


                Alle Segmente von hinten nach vorne
                zeichnen
```

Screen # 14

```
\ Mandelbrot-Hauptroutine                          03sep89 ck

: mandel        ( -- )
        page
        overwrite       true sf_perimeter       1 sl_color
        1 sl_width      0 sf_interior   0 sf_style
        xoffset 1- yoffset 1-
        xoffset size + yoffset size +   bar
        init.square
        draw.mandelbrot
        draw.surface
        10 0 at
```

Screen # 29

```
                                                   03sep89 ck

Diverse VDI-Parameter

Kleines Window für Kontrolle zeichnen

Mandelbrot berechnen
Mandelbrot zeichnen
```

Behandlung einer CASE - Situation, Teil 2

von Jörg Staben

Positionelles CASE

Einen ganz anderen Lösungsansatz bietet ein positioneller CASE-Konstrukt, bei dem die Fallunterscheidung durch den Fall-Index tabellarisch vorgenommen wird.

Dieser Ansatz setzt die Fähigkeit der Programmiersprache voraus, Prozeduren in Tabellen ablegen und eine in einer Tabelle gefundene Prozedur ausführen zu können.

Bei den bisherigen Lösungen wurden immer eine Reihe von Vergleichen zwischen einem Fall-Index und einer Liste von Fall-Konstanten vorgenommen; nun wird der Fall-Index selbst benutzt, die gewünschte Prozedur auszuwählen. Die Verwendung des Fall-Index als Selektor bringt auch Vorteile in der Laufzeit, weil die Vergleiche entfallen.

Wenn FORTH-Worte in Tabellen abgelegt werden sollen, stellt sich das Problem, daß ein FORTH-Wort bei seinem Aufruf normalerweise die eincompilierten Worte ausführt. Bei einer Tabelle ist das nicht erwünscht; dort ist sinnvollerweise gefordert, daß die Startadresse der Tabelle übergeben wird, damit der Fall-Index als Offset in diese Tabelle genutzt werden kann.

Dies läßt sich in volksFORTH entweder auf die traditionelle Weise mit] und [oder dem volksFORTH-spezifischen *Create:* lösen:

```
Create  Glas
        ]  nehmen  links schieben
                   rechts schieben
           trinken  [

Create: Glas
         nehmen
         links schieben
         rechts schieben
         trinken ;
```

In der Tabelle *Glas* wird auch deutlich, welche Funktion das Dummy-Wort *schieben* außer einer besseren Lesbarkeit noch hat: Es löst die Schwierigkeit, daß 6 möglichen Wurfergebnissen nur 4 mögliche Aktionen gegenüberstehen.

Zugleich fassen wir unsere Fehlerbehandlung im Fall eines unglaubwürdigen Fall-Index in einem Wort *schimpfen* zusammen:

```
: schimpfen
        invers ." Betrug! " normal ;
```

Damit steht der Entwicklung einer Zugriffsprozedur für unsere Tabelle nichts mehr im Wege.

Die Art und Weise des Zugriffs im Wort *bewegen* entspricht der auf eine Zahl in einem eindimensionalen Feld, einem Vektor. So übernehmen wir diese Art des Zugriffs, lesen aber nicht die gefundene Adresse aus, um den Zahlenwert zu erhalten, sondern führen das Wort an der gefundenen Adresse mit *PERFORM* aus:

```
: bewegen  ( adr n -- cfa )
    2* +  perform ;

: richtig  ( n  --  0<= n <= 3 )
    swap
    1 max  6 min
              \ ein bißchen Sicherheit
    3 case? IF  2  1- exit  ENDIF
    5 case? IF  4  1- exit  ENDIF
    1-  ;
              \ ein bißchen Justage
```

Dieses Wort *richtig* läßt zwar Werte kleiner als 1 und größer als 6 zu, justiert sie aber auf den Bereich zwischen 1 und 6 . Auch hier müßte eine Möglichkeit geschaffen werden, ein Wurfergebnis außerhalb der 6 Möglichkeiten als Betrugsversuch zurückzuweisen !

Die letzte Frage, wie man nun die Tabelle und die Zugriffsprozedur verbindet, wird von einem noch relativ unbekannten Wort des volksFORTH gelöst:

```
\ :Does>  für  Create <name>
\              :Does> <action> ;
\              ks 25 aug 88

| : (does>    here >r [compile] Does> ;

  : :Does>
   last @ 0= Abort" without reference"
   (does>  current @  context !
   hide 0 ] ;
```

Stichworte

- CASE,
- volksFORTH,
- Würfelspiel

Dieses Wort *:DOES>* weist dem letzten über *Create* definierten Wort einen Laufzeit-Teil zu. Dieses Wort wurde von Herrn Klaus Schleisiek-Kern programmiert und auch hier gilt der Hinweis, nach dem Kompilieren das mit | als headerless deklarierte Wort durch *clear* zu löschen.

Damit läßt sich die inzwischen 8. Version der Auswertung so schreiben:

```
Create: Auswertung.8
          nehmen
           links schieben
           rechts schieben
          trinken ;
       :Does>
          richtig bewegen ;
```

Setzt man *:DOES>* nicht ein, so hat man die Tabelle und die Zugriffsprozeduren als unabhängige Worte und das vollständige Programm ein anderes Erscheinungsbild als bisher gewohnt, weil *richtig* und *bewegen* nach schönster UPN-Philosophie Operatoren auf *Glas* sind:

```
: CRAPS
     cr  Anfrage cr
           input#
           Glas richtig bewegen
     cr  Glückwunsch  ;
```

Entschließt man sich dagegen, Tabelle und Zugriffsprozedur in einem Wort zu definieren, so finden wir wieder das aus der CASE-Diskussion gewohnte Erscheinungsbild:

```
: CRAPS2
     cr  Anfrage cr
           input#
           Auswertung
     cr  Glückwunsch  ;
```

Nachdem wir oben Tabelle und Zugriffsprozedur auf diese Tabelle »von Hand« entworfen haben, bietet sich bei häufigerem Einsatz solcher Tabellen die Verwendung von **positional CASE defining words** an. Auch hier wiederum zuerst die traditionelle Variante, danach die volksFORTH-gemäße Lösung:

```
: Case:    ( -- )
   Create:  Does> ( pfa -- )
            swap 2* +  perform ;

\ alternative Definition für CASE:
  : Case:
      :  Does>    ( pfa -- )
         swap 2* +  perform ;
```

Eine sehr elegante Möglichkeit, die oben angesprochene unbefriedigende Fehlerbehandlung im Falle eines unglaubwürdigen Fall-Indexes zu handhaben, bietet das Wort *Associative:* .

Dieses Wort *Associative:* durchsucht eine Tabelle nach einer Übereinstimmung zwischen einem Zahlenwert auf dem Stack und den Zahlenwerten in der Tabelle und liefert den Index der gefundenen Zahl (match) zurück. Im Falle eines Mißerfolgs (mismatch) wird der größtmögliche Index + 1 (out of range = maxIndex + 1) übergeben:

```
: Associative:            ( n -- )
   Constant   Does>       ( n - index )
   dup @ -rot
   dup @ 0
   DO  2+  2dup  @ =
       IF
           2drop drop
           I  0 0  LEAVE
       THEN
   LOOP 2drop ;


6 Associative:  Auswerten

                  1 ,
                2 , 3 ,
                4 , 5 ,
                  6 ,


Case:  Handeln     \ besteht aus :

                nehmen
              links  links
             rechts  rechts
                trinken
          schimpfen ;
```

In unserer 9. Auswertung wird statt der Primitivabsicherung über *MIN* und *MAX* eine **out of range** Fehlerbehandlung namens *schimpfen* an der Tabellenposition *maxIndex + 1* durchgeführt. Die ganzen internen Variationen haben bisher kaum Einfluß auf das Erscheinungsbild unseres Programms gehabt:

```
: CRAPS   ( -- )

     cr  Anfrage cr
         input#  Auswerten  Handeln
     cr  Glückwunsch   ;
```

Einsatzmöglichkeiten

Dieser letzte Teil dieser Ausführungen über die Möglichkeiten, eine CASE-Situation zu handhaben, greift Anregungen aus der Literatur [5] und [6] auf und möchte an Hand der geringfügigen Unterschiede die Bandbreite dieser Möglichkeiten darstellen.

Dabei helfen uns zwei kleine Worte - *CLS* löscht, im volksFORTH oft vermißt, den gesamten Bildschirm und *CELLS* macht die Berechnung des Tabellenzugriffs deutlicher:

```
: cells  2* ;
: cls    full page ;
```

Das Inhaltliche und die tabellarische Struktur bleiben unverändert, lediglich die Behandlung einer **out of range** Situation wird diesmal mit *min* und *max* und zweimaligem Eintragen der Fehler-Routine *schimpfen* verwirklicht.

```
Create: Handlung
     schimpfen   nehmen links links
                 rechts rechts trinken
     schimpfen  ;

\ Die Ausführung einer Liste
\ nach Floegel 7/86
: auswählen   ( adr n -- *cfa )
     2 arguments
     swap  0 max     \ out of range MIN
           7 min     \ out of range MAX
     cells +  ;

: auswerten   ( n -- ) 1 arguments
     Handlung auswählen  perform ;

: .all  ( -- )
     8 0  DO
           cr I dup . auswerten
           2 spaces
        LOOP ;
```

Auswählen übergibt bei gegebenem Vektor und gegebenem Index einen Zeiger auf die **code field address** des entsprechenden Wortes. *Auswerten* führt das so ausgewählte Wort aus und *.all* diente nur zur Kontrolle. Ein Wort, das angelegte Datenstrukturen auf dem Bildschirm darstellt, sollte in der Entwicklungsphase eines Programmes immer dabei sein.

Eine weitere Möglichkeit, Werte in einen Vektor einzutragen, hat Herr Floegel in seinem Buch [4] dargestellt:

```
Create Tabelle  8 cells allot
       :Does> ( i -- adr )
               swap cells + ;s

 ' schimpfen  0 Tabelle !
 ' nehmen     1 Tabelle !
 ' links dup  2 Tabelle !
              3 Tabelle !
 ' rechts dup 4 Tabelle !
              5 Tabelle !
 ' trinken    6 Tabelle !
 ' schimpfen  7 Tabelle !

: auswerten  ( i -- )
     0 max  7 min  Tabelle perform ;

: .action    ( i -- )
     Tabelle @  >name
     bright .name normal ;
: .Tabelle  ( -- )
     cr  8 0 DO  cr I .action  LOOP ;
```

Auch hier besteht mit *.action* und *.Tabelle* wieder die Möglichkeit, sich den Vektor darstellen zu lassen.

Nachdem hoffentlich deutlich geworden ist, daß *TABELLE* und *VEKTOR* synonym sind, soll das Beispiel des Würfelspiels bis über die Grenze strapaziert werden.

Hierbei handelt es sich um eine geringfügige Modifikation von [5]. Hier werden keine neuen Strukturen der

Behandlung einer CASE-Situation

Programmsteuerung gezeigt, sondern die Verknüpfung eines Vektors von Worten und einer Menü-Option steht im Vordergrund:

```
Create function
        ] noop noop noop noop
          noop noop noop noop [
       :Does> ( i -- adr )
          swap  0 max 7 min
          cells + ;
```

function ist ein **execution vector**, der mit NOOP vorbesetzt ist. Zur Laufzeit liefert er die Adresse des indizierten Elementes zurück.

```
: .action  ( i adr -- )
     @    >name  bright .name normal ;
```

.action gibt den Namen eines Wortes aus, dessen CFA in eine Adresse eingetragen wurde.

```
: option   ( i -- )
  r>
  dup 2+ >r              \ i *w.adr
  @                      \ i w.adr
  stash swap function !
                         \ i  w.adr i adr
  function .action ;     \ i adr
```

option holt die Adresse des auf *option* folgenden Wortes. Das Wort soll nicht ausgeführt werden, sondern das danach. Nur der Pointer auf das Wort soll ausgewertet werden. Nach dem übergebenen Index wird der Pointer in *function* eingetragen. Der Name des so eingetragenen Wortes wird angezeigt !

```
\ Menü jrg 06feb89
: Menü
    0 option schimpfen
      1 option nehmen
      2 option links
      3 option links
      4 option rechts
      5 option rechts
      6 option trinken
    7 option schimpfen  ;
```

Wenn das Wort *Menü* aufgerufen wird, werden nicht nur die Optionen in die Tabelle eingetragen, sondern auch namentlich auf dem Bildschirm dargestellt. Diese Technik bietet sich für eine Menüzeile an fester Bildschirmposition an, ähnlich der Statuszeile des volksFORTH.

```
: fkey   ( -- )
   key &58 + abs  function perform ;
```

FKEY liefert beim Druck einer Funktionstaste einen Wert von -59 bis -68 zurück. Dieser wird für 10 Funktionstasten in den Bereich von -1 bis -10 skaliert und der Absolutwert gebildet.

```
\ Auswahl    jrg 06feb89
Create Titel
," Bitte Ihre Würfelzahl: "
Create Taste
," Die entsprechende Funktionstaste: "

: string2/  c@ 2/ - ;
: center  c/row 2/ ;

: Auswahl
     cls cr cr
     row    center Titel string2/
            at  Titel count type
     row 2+ 11
            at  Menü
     cr cr
     10 0 DO
          row 1+ center Taste
          string2/ at
          Taste count type
          fkey  LOOP ;
```

Hier kann man zwar die Augenzahl seines Wurfes über die entsprechende Funktionstaste auswerten lassen, aber diese Programmierung der Menü-Darstellung ist äußerst verbesserungsbedürftig. Deshalb bleibt nur noch, Ihnen zu diesem Thema die Worte *Input:* und *Output:* zusammen mit den Dienstworten *IN:* und *OUT:* zur Analyse zu empfehlen und auf weitere Anregungen zu gleichen oder ähnlichen Programmsituationen zu hoffen:

```
\ in/output structure      KS 31 OCT 86
| : Out:
    Create dup c, 2+
    Does> c@ output @ + perform ;

  : Output:  Create: Does> output ! ;
0   Out: emit    Out: cr
    Out: type    Out: del
    Out: page    Out: at
    Out: at?     drop

| : In:
    Create dup c, 2+
    Does> c@ input @ + perform ;
  : Input:   Create:  Does> input ! ;
0   In: key       In: key?
    In: decode    In: expect  drop
```


[1] Ultimate CASE-Statement, Wil Baden, VD2/87 S.40 ff.

[2] Just in CASE, Dr. Charles Eaker, FORTH DIM II/3

[3] FORTH 83, R. Zech, S.98ff/S.318f.

[4] FORTH Handbuch, E. Floegel, S.109

[5] Menus in FORTH, W. Wejgaard, Elektroniker 9/88, S.109 ff.


```
global locksave
: _error
  1 fgcolor textmode cr
  abort ;
create fileinfo 260 allot
: FDIR
  " Directory of Volume DF0:" count type cr
  2 fgcolor textmode
  bold textmode
  0" df0:" !d1
  -2 "d2
  dos@ 14 dup
  0 = if _error then
  locksave !
  locksave @ !d1
  fileinfo !d2
  dos@ 17
  dup 0 = if _error then
  fileinfo 8 +
  30 type
  italic textmode
  1 fgcolor textmode
  cr cr ." ---> Contents:" cr
  plain textmode
  3 fgcolor textmode
  begin
   locksave @ !d1
   fileinfo !d2
   dos@ 18
   dup 0 = if _error then
   fileinfo 8 + 30 type cr
  again ;
```

Beispielprogramm für MULTI-FORTH (siehe nächste Seite)

MULTI-FORTH, der Einstieg

Der Bericht eines, von langwierigen Programmierabläufen wie Compilieren und Linken geplagten FORTH-Neulings.

Stefan Kempf, Öschelbronnerstraße 2/3,
7130 Mühlacker 2

Nach einer langen und nervenaufreibenden Odyssee durch die verschiedensten Programmiersprachen, vorbei an BASIC, PASCAL, C, MODULA 2 und ASSEMBLER, landete ich schließlich bei FORTH. Ich entschloß mich, den Einstieg in FORTH mit MULTI-FORTH zu wagen.

Nach dem Auspacken präsentierte es sich schlicht (2 Disketten), jedoch mit einem auffallend großzügig gestalteten Handbuch. Als ich die obligatorische Einleitung hinter mich gebracht hatte, ging es auch schon mit den Überraschungen los. Die beigelegten Demoprogramme ließen mein Herz als Amiga-Fan höher schlagen. MULTI-FORTH deckt alle Bereiche des Amiga, seien es DOS, Windowprogrammierung, Hardwareeinsatz oder Librarybenutzung ab. Das bedeutet allerdings, daß der Befehlssatz deutlich erweitert werden mußte. Neben diesen Bereicherungen, fiel mir das Wort *LOCALS|*, das zur Stackmanipulation dient und vom FORTH-83 Standard abweicht, auf.

Zur Verdeutlichung hier ein kleines Beispiel:

```
: QUADRAT (a\b\c -- a^2+b^2+c^2)
  dup *
  swap dup * +
  swap dup * + ;
```

wird bei der Verwendung von *LOCALS|* zu [1]:

```
: QUADRAT (a\b\c -- a^2+b^2+c^2)
  locals| c b a |
  a a *
  b b * +
  c c * + ;
```

Nachdem das Handbuch auf die Besonderheiten der FORTH-Arithmetik und der Stackbehandlung eingegangen war, folgen nun Vergleiche und Schleifen. Dieser Bereich der Befehle unterscheidet sich jedoch nicht vom FORTH-83 Standard.

Alle Anweisungen von *CASE* bis *BEGIN-WHILE-REPEAT* sind vorhanden.

Doch nun zu den weitaus interessanteren, amigaspezifischen Erweiterungen. MULTI-FORTH bietet die Möglichkeit Datenstrukturen anzulegen und Library-Calls durchzuführen. Dies bildet die Voraussetzung für die Windowprogrammierung.

Hier ein Beispiel für die Verwendung von Strukturen:

```
struct NewWindow mywindow
 mywindow InitWindow
    0 mywindow +nwTopEdge w!
    0 mywindow +nwLeftEdge w!
  320 mywindow +nwWidth w!
  200 mywindow +nwHeigth w!
  ACTIVATE WINDOWDEPTH | mywindow
+nwFlags !
  MOUSEBUTTONS mywindow +nwIDCMPFlags !
structend
```

Bevor ich nun näher auf die oben erwähnten Library-Calls eingehe, sollte ich zuerst Grundsätzliches zu den Amiga-Libraries sagen.

Das Betriebssystem des Amiga ist in Devices, die zur Hardwarerestaurierung dienen und in Bibliotheken, den sogenannten Libraries, aufgeteilt. Beim Amiga 500 gibt es 15 Libraries; beim Amiga 2000 kommt noch die Janus-Library dazu, die das Dual-Ported RAM verwaltet. Die in den Libraries gesammelten Routinen werden über negative Offsets angesprungen. Die Libraries müssen jedoch zuerst geöffnet werden, um mit ihnen arbeiten zu können. Ich glaube es ist von Vorteil, hier ein kurzes ASSEMBLER-Programm, das Libraries verwendet, anzuführen [2].

```
OpenLib = -408
ExecBase = 4

Execute = -222

move.l execbase,a6
lea libnam9,a1
jsr OpenLib(a6)
move.l d0, libhandle
beq error  : ggf zu Fehler-routine
verzweigen!
move.l libhandle,a6
move.l #command,d1
clr.l d2
move.l conhandle,d3
jsr Execute (a6)
rts

libname: dc.b "dos.library",0
libhandle: dc.l 0
command: dc.b "dir",0
```

Da MULTI-FORTH die zeitaufwendige Prozedur des Öffnens und Schließens der Libraries übernimmt, verkürzt sich das Programm deutlich.

```
0"dir" !d1 0 !d2 conhandle @ !d3 dos
37
```

Hieraus läßt sich erkennen, wie einfach die Benutzung der Libraries durch MULTI-FORTH gestaltet ist. Darüberhinaus unterstützt MULTI-FORTH die Floating-Point-Arithmetik und das Multitasking des Amiga. Zusammenfassend läßt sich sagen, daß MULTI-FORTH, obwohl es leicht vom FORTH-83 Standard abweicht, eine sehr gute und leicht zu handhabende Programmierumgebung ist. MULTI-FORTH läßt für den Amiga-Anwender kaum Wünsche offen und ist so eine sehr gute Bereicherung auf dem Markt der Programmiersprachen. Hier noch einmal kurz die Features von MULTI-FORTH:

- Möglichkeit zu amigaspezifischen Programmen
- nahezu FORTH-83 Standard
- Interpreter und Compiler
- Möglichkeit zur Erstellung von Snapshots und allein lauffähigen Programmen

Quellenangaben:

[1] MULTI-FORTH
Benutzerhandbuch

[2] AMIGA
Maschinensprache
DATA-BECKER

Stefan Kempf
Öschelbronnerstr. 2/3
7130 Mühlacker 2

Stichworte

» MULTI-FORTH,
» Einstieg,
» AMIGA

Assembler im Vergleich

von Michael Sundermann

Das volksFORTH 3.81.41 für den IBM PC stellt zwei Assembler zur Verfügung, den F83-Assembler von Laxen&Perry und den eigentlichen volksFORTH-Assembler. Beide Assembler benützen andere Notationen, um dasselbe auszudrücken. Dies kann manchmal zu großen Schwierigkeiten führen, insbesondere dann, wenn es sich um kleine Unterschiede handelt.

Aus diesem Grund habe ich mir die Arbeit gemacht, beide Assembler zu vergleichen, damit die wichtigsten Unterschiede deutlich hervortreten. In die Vergleichstabellen habe ich noch den Microsoft Assembler aufgenommen, der als neutraler Bezugspunkt dient. Er wird auch in den meisten Assembler-Büchern behandelt, so daß er vielen bekannt sein dürfte.

Generell kann man sagen, daß der F83-Assembler fast die gleiche Syntax wie der Microsoft Assembler besitzt, mit der Ausnahme, daß alles rückwärts notiert wird, gemäß der umgekehrten polnischen Notation. Folgendes Beispiel soll zeigen, daß dennoch kleine Unterschiede existieren: Der Microsoft Assembler verwendet

```
out 60,ax
```

dem entspräche in F83-Assembler:

```
ax # 60 out
```

Der F83-Assembler akzeptiert dies zwar, generiert aber falschen Code. Richtig programmiert man:

```
# 60 ax out
```

Register: 8086-CPU	F83-Asm	volksFORTH-Asm	Forth-Maschine F83-Asm	volksFORTH-Asm
AX (AH AL)	AX (AH AL)	A (A+ A-)		
BX (BH BL)	BX (BH BL)	BX (B+ B-)	RP	R (R+ R-)
CX (CH CL)	CX (CH CL)	C (C+ C-)		
DX (DH DL)	DX (DH DL)	D (D+ D-)		
SP	SP		SP	S
BP	BP		UP	U
SI	SI	SI	IP	I
DI	DI	DI	W	W
PC				
CS	CS	C:		
DS	DS	D:		
SS	SS	S:		
ES	ES	E:		
PSW				

Tabelle 1

Der volksFORTH-Assembler benützt dagegen eine andere Notation, die mit dem Microsoft Assembler keine Gemeinsamkeiten mehr besitzt.

Beide Assembler haben die schlechte Eigenschaft nur das Allerwenigste zu kontrollieren. In beiden Assemblern kann man zum Beispiel schreiben:

```
F83-Asm          0 # es mov
volksFORTH-Asm   0 # e: mov
```

Beide generieren einen falschen Code und melden keinen Fehler, obwohl dieser Befehl nicht erlaubt ist. Man kann keinen Direktwert in ein Segmentregister laden, dies ist nur über den Umweg über ein normales Register möglich.

Es bleibt einem nichts anderes übrig, als ständig im 8088/8086 Handbuch nachzuschlagen, ob ein Befehl überhaupt erlaubt ist und dann noch an Hand eines Dumps

```
FORTH-Wort: dump (addr anzahl -- )
```

zu vergleichen, ob die Assembler auch den richtigen Code generieren. In der Praxis spielt das keine große Rolle, weil man so oder so 80 Prozent eines Programmes mit 20 Prozent des Befehlssatzes codiert. Weiter schreibt man in alter FORTH-Tradition pro Wort sowieso nur 1 bis 3 Zeilen Code, wodurch falscher Code beim Austesten schnell gefunden wird.

In der Tabelle 1 werden die Namen der Register gegenübergestellt. Der F83-Assembler benützt die üblichen Namen. Beim volksFORTH-Assembler werden die Namen auf ein Zeichen abgekürzt. Das ist praktisch und spart Schreibarbeit. Es gibt auch eine Ausnahme; das BX-Register heißt nicht B sondern BX.

Die virtuelle FORTH-Maschine wird im volksFORTH vollständig auf die Register abgebildet. So kommt es, daß manche Register unter zwei Namen angesprochen werden können. Einmal im Sinne der FORTH-Maschine und einmal im Sinne eines nor-

Stichworte

» Microsoft-Assembler,
» F83-Assembler,
» volksFORTH

Adressierungsarten	M-Asm	F83-Asm	volksFORTH-Asm
Werte	wert equ 500 var1 dw ?	500 constant wert variable var1	500 constant wert variable var1
unmittelbar	mov ax,500 mov ax,wert	500 # ax mov wert # ax mov	500 # a mov wert # a mov
direkt	mov ax,var1	900 #) ax mov var1 #) ax mov	900 #) a mov var1 #) a mov

indirekt	mov ax,[bx]
	mov ax,[bp]
	mov ax,[si]
	mov ax,[di]
		0 [bx] ax mov
		0 [bp] ax mov
		0 [si] ax mov
		0 [di] ax mov
			bx) a mov
			u) a mov
			si) a mov
			di) a mov

indirekt + displacement	mov ax,[bx+700]
	mov ax,[bp+700]
	mov ax,[si+700]
	mov ax,[di+700]
		700 [bx] ax mov
		700 [bp] ax mov
		700 [si] ax mov
		700 [di] ax mov
			700 bx d) a mov
			700 u d) a mov
			700 si d) a mov
			700 di d) a mov

indirekt und indiziert	mov ax,[bx+si]
	mov ax,[bx+di]
	mov ax,[bp+si]
	mov ax,[bp+di]
		0 [bx+si] ax mov
		0 [bx+di] ax mov
		0 [bp+si] ax mov
		0 [bp+di] ax mov
			bx si i) a mov
			bx di i) a mov
			u si i) a mov
			u di i) a mov

indirekt und indiziert + displacement	mov ax,[bx+si+700]
	mov ax,[bx+di+700]
	mov ax,[bp+si+700]
	mov ax,[bp+di+700]
		700 [bx+si] ax mov
		700 [bx+di] ax mov
		700 [bp+si] ax mov
		700 [bp+di] ax mov
			700 bx si di) a mov
			700 bx di di) a mov
			700 u si di) a mov
			700 u di di) a mov

Tabelle 2

malen Registers. Vorher ist jedoch das Register zu retten, da ansonsten mit hoher Wahrscheinlichkeit der Absturz der FORTH-Maschine droht.

Weiter werden in der Tabelle 2 die Adressierungsarten verglichen, wobei sich der F83-Assembler das Leben einfach macht und für die indirekte Adressierung als auch für die indirekte Adressierung mit Displacement dieselbe Konstruktion benützt.

```
M-Asm:     mov ax,[bx]
F83-Asm:   0 [bx] ax mov
M-Asm:     mov ax,[bx+700]
F83-Asm:   700 [bx] ax mov
```

So liegt es dann letztlich am Programmierer, die Null nicht zu vergessen. Aber dennoch generiert der F83-Assembler bei der Adressierung ohne Displacement den schnelleren Code.

Der volksFORTH-Assembler geht hier sehr logisch vor und benützt folgende Symbole:

-) indirekt
- i) indirekt und indiziert
- d) indirekt und displacement
- di) indirekt und indiziert und displacement

Tabelle 3 zeigt alle Varianten des *JMP*-Befehls. Deutlich geht daraus hervor, daß kleine Unterschiede für große Verwirrung sorgen können. Zum Beispiel:

```
marke #) jmp
marke # jmp
```

Jetzt sollte man nur noch wissen, welcher Befehl zu welchem Assembler gehört, denn beide generieren den gleichen Code.

Weiter sieht man, daß wenn Marken im Spiel sind, es sich immer um Rückwärtssprünge handelt. Dies ist in FORTH leicht zu implementieren.

Vorwärtssprünge über große Distanzen sind ein Kapitel für sich und werden hier nicht behandelt. Sie werden normalerweise selten gebraucht. Es ist in FORTH üblich Bottom-up zu programmieren und daher sind alle Namen bekannt, die man verwenden will.

Sprünge	M-Asm	F83-Asm	volksFORTH-Asm
Sprung relativ zum Programmzähler (8-bit oder 16-Bit)	marke: ... ... jmp marke	label marke ... ... marke #) jmp	label marke ... ... marke # jmp
Sprung über Register	jmp bx	bx jmp	bx jmp
Sprung indirekt über Speicher (16-Bit)	zeiger dw ? jmp zeiger	variable zeiger zeiger s#) jmp	variable zeiger zeiger #) jmp
Sprung indirekt über Speicher (16-Bit)	jmp [bx]	0 [bx] jmp	bx) jmp
Sprung indirekt über Speicher (32-Bit)	zeig dd ? jmp zeig	create zeig 4 allot far zeig s#) jmp	create zeig 4 allot far zeig #) jmp
Sprung indirekt über Speicher (32-Bit)	jmp dword ptr [bx]	far 0 [bx] jmp	far bx) jmp
Sprung absolut (32-Bit) se = segment of = offset	marke label far ... ... (anderes Segment) jmp marke	label se label of ... ... far of se #) jmp	label se label of ... ... far of se # jmp

Tabelle 3

Trotzdem hoffe ich, daß ein anderer Lust verspürt, über dieses Thema in allen Variationen zu berichten.

Vorwärtssprünge über kleine Distanzen (-128 bis 127) sind natürlich notwendig und werden mit Hilfe von Kontrollstrukturen gelöst. Dies ist einer der großen Vorteile von FORTH. Man gibt sich nicht mit 'gotos' und 'labels' ab, sondern arbeitet mit *IF,WHILE* und anderen Strukturen.

Bei den *IN/OUT*-Befehlen (Tabelle 4) verhalten sich die Assembler genau gegensätzlich.

```
F83-Asm:          60 # ax out
volksFORTH-Asm:   a 60 #) out
```

Der Befehl, der das Einer-Komplement liefert, lautet bei den Assemblern anders.

```
F83-Asm:          bx not
volksFORTH-Asm:   bx com
```

Der volksFORTH-Assembler schert hier aus der Reihe, weil dieser das *NOT* bei den Kontrollstrukturen verwendet.

Eine Besonderheit bietet noch der volksFORTH-Assembler bei den Shift-Befehlen. Er braucht dort ein *c** statt ein *cl*.

```
F83-Asm:          bx cl shr
volksFORTH-Asm:   bx c* shr
```

Jetzt kommen wir zu den höheren Kontrollstrukturen in FORTH. In Tabelle 5 werden die entsprechenden Strukturen den *JMP*-Befehlen gegenübergestellt. Beim F83-Assembler fällt auf, daß er beide Möglichkeiten,

Verschiedenes	M-Asm	F83-Asm	volksFORTH-Asm
Segment laden	mov ax, es mov es, ax	es ax mov ax es mov	e: a mov a e: mov
Segment Präfix	mov ax, es:[bx] mov es:[bx], ax	es: 0 [bx] ax mov ax es: 0 [bx] mov oder es seg 0 [bx] ax mov ax es seg 0 [bx] mov	e: seg bx) a mov a e: seg bx) mov
Return	ret ret 6 proc far ... ret proc far ... ret 6	ret 6 + ret far ret far 6 + ret	ret 6 + ret lret 6 + lret
Repeat string	rep movsb rep movsw repz cmpsb repnz cmpsb	rep byte movs rep movs repz byte cmps repnz byte cmps	rep byte movs rep movs 0= rep byte cmps 0< > rep byte cmps
1-Komplement	not bx	bx not	bx com
2-Komplement	neg bx	bx neg	bx neg
Verschieben (analog shl, ror, ...)	shr bx,1 shr bx,c	1 # bx shr bx cl shr	1 # bx shr bx c* shr
IN/OUT	in ax,,60 in ax,,dx out 60,,ax out dx,,ax	60 # ax in dx ax in 60 # ax out dx ax out	a 60 #) in a d in a 60 #) out a d out

Tabelle 4

entweder mit Labels und den *JMP*-Befehlen oder mit *IF, WHILE*, usw. zu arbeiten, unterstützt. Das Wort 0= entspricht dem Befehl *jmp if not equal*, denn die Logik ist genau verkehrt. Der Microsoft Assembler springt, wenn eine Bedingung erfüllt ist.

```
jne then
nop
then:
```

Wenn das Zeroflag = 0 ist, wird zur Marke then: gesprungen, und wenn das Flag = 1 ist, wird der Code unmittelbar nach dem *JNE*-Befehl ausgeführt, hier der *NOP*-Befehl. Diese Eigenheit führt dazu, daß man in FORTH das Gegenteil, also 0= schreibt.

```
0= if nop then
```

Wenn das Zeroflag = 1 ist, wird die Bedingung 0= wahr und der Then-Teil wird ausgeführt, hier auch wieder der *NOP*-Befehl. Wenn das Zeroflag = 0 ist, wird wie oben zu dem Befehl hinter dem *THEN*-Teil gesprungen. Das verhält sich nur auf der Programmierebene so. Es wird in beiden Fällen der gleiche Code generiert.

Der F83-Assembler spart sich eine Menge von Worten, indem er sie mit dem *NOT*-Wort ins Gegenteil verkehrt. Dies ist zwar einfacher für den Entwickler gewesen, aber der Mensch tut sich manchmal schwer mit Verneinungen. In FORTH weiß man sich zu helfen.

Bsp: `0= not constant 0<>`

Wenn man Tabelle 5 genauer betrachtet, stellt man fest, daß der F83-Assembler nicht alle Befehle durch eine höhere Konstruktion ersetzen kann wie zum Beispiel der Befehl *JCXZ*.

Man kann diesem Mangel auf zwei Weisen abhelfen. Im volksFORTH-Assembler codiert man zum Beispiel folgendes:

```
c0= not ?[
...
]?
```

Dies bedeutet im Microsoft Assembler:

```
jcxz marke
...
marke:
```

Dazu definiert man im F83-Assembler im Assembler Vokabular folgendes

```
$e3 constant cx0<>
```

und schreibt dann analog:

```
cx0<> if
...
then
```

Die andere Möglichkeit besteht darin den Befehl *JCXZ*, der im F83 definiert ist, selbst zu verwenden.

```
label done
...
done jcxz
```

Kontroll-strukturen	(c = carry v = overflow s = sign z = zero p = par.)	M-Asm	F83-Asm	volksFORTH-Asm
jmp if above	c = 0 and z = 0	ja	u< = ja	> =
jmp if above or equal	c = 0	jae jnc	u< jae	cs
jmp if below	c = 1	jb jc	u> = jb	cs not
jmp if below or equal	c = 1 or z = 1	jbe	u> jbe	> = not
jmp if greater	s = v and z = 0	jg	< = jg	< =
jmp if greater or eq.	s = v	jge	< jge	<
jmp if less	s< > v	jl	> = jl	< not
jmp if less or equal	s< > v and z = 1	jle	> jle	< = not
jmp if equal	z = 1	je jz	0< > je	0= not
jmp if not equal	z = 0	jne jnz	0= jne	0=
jmp if sign	s = 1	js	0> = js	0< not
jmp if not sign	s = 0	jns	0< jns	0<
jmp if overflow	v = 1	jo	jo	os not
jmp if no overflow	v = 0	jno	ov jno	os
jmp if parity even	p = 1	jpe	jpe	pe not
jmp if parity odd	p = 0	jpo	jpo	pe
jmp if cx is zero	cx = 0	jcxz	jcxz	c0= not
loop	cx< > 0	loop	loop	c0=
loop if equal	cx< > 0 and z = 1	loope	loope	?c0= not
loop if not eqeal	cx< > 0 and z = 0	loopne	loopne	?c0=

Tabelle 5

Kontrollstrukturen	M-Asm	F83-Asm	volksFORTH-Asm
Verzweigung	jne marke ... marke:	0= if ... then	0= ?[...]?
Verzweigung mit else	jne else ... jmp short ende else: ... ende:	0= if ... ... else ... then	0= ?[... ...][...]?
Endlos-Schleife	marke: ... jmp short marke:	begin ... again	[[...]]
Until-Schleife	marke: ... jne marke:	begin ... 0= until	[[... 0= ?]

While-Schleife	marke:
	...
	jne ende:
	...
	jmp short marke
	ende:
		begin
		...
		0= while
		...
		repeat
			[[
			...
			0= ?[[
			...
			]]?

Schleife mit cx-Register	marke: ... loop marke	begin ... loop	[[... c0= ?]

Tabelle 6

Oder man arbeitet mit dem Wort here, das im Assembler Vokabular enthalten ist. Mit

```
here 3 + jcxz nop nop
```

wird, falls cx Null ist, die Ausführung nach dem ersten *NOP*-Befehl, also mit dem zweiten *NOP* fortgesetzt.

Um die Konstruktion mit *HERE* zu ermöglichen, muß man noch einen Fehler im File f83asm.scr korrigieren.

Im Screen# 6 fügt man folgendes ein:

```
| : within
    1- >r over > swap r> >  or not ;
| : 3MI ... within ... ;
```

statt:

```
| : 3MI ... uwithin ... ;
```

Der Platz reicht gerade für die zusätzliche Zeile mit dem Wort *WITHIN* aus.

Der Fehler liegt darin, daß man zwar eine vorzeichenlose Zahl als Argument des Wortes *JCXZ* übergibt, diese Zahl aber wieder von einer ähnlich großen Zahl abgezogen wird. So erhält man dann eine Zahl, die die relative Distanz zum Sprungziel aufweist. Da die Distanz nur ein Byte umfaßt, damit Sprünge von -128 bis 127 zuläßt, wird sie deshalb mit *WITHIN* auf Gültigkeit überprüft.

Tabelle 5 beschreibt jetzt die eigentlichen Kontrollstrukturen. Die beiden Assembler bieten die gleichen Strukturen und generieren **genau** den Code, der mit dem Microsoft Assembler umständlich beschrieben wird. Nehmen wir zum Beispiel die *IF-THEN-ELSE* Struktur:

```
F83-Asm:          0= IF 5 # ax add
                     ELSE 10 # ax add
                     THEN ...
volksFORTH-ASm:   0= ?[ 5 # a add
                     ][ 10 # a add
                     ]? ...
```

Wenn das Zero-Flag gesetzt ist, so wird 5 sonst 10 dazuaddiert. Im Microsoft Assembler lautet dies:

```
          jne else1
          add ax,5
          jmp short ende
else1:    add ax,10
ende:     ...
```

In allen 3 Fällen wird der gleiche Code generiert:

```
100: 75 05 jne nach Adresse 102+5=107
102: 05 0500 addiere 5
105: eb 03 jmp nach Adresse 107+3=10a
107: 05 0a00 addiere 10
10a: ...
```

Ich hoffe, daß jetzt keiner mehr Schwierigkeiten hat, so wie ich am Anfang, die Assembler zu verstehen und anzuwenden. Viel Spaß.

Quelle:

[1] Handbuch zum volksFORTH

[2] Quelltexte asm.scr

[3] Quelltexte f83asm.scr

Der fleißige Biber

Eine Turing-Maschine nach Tibor Rado

von Friederich Prinz, Moers

Geschichtliches

Um 1900 herum formulierte der Mathematiker David Hilbert eine Forderung, die von diesem Zeitpunkt an die Mathematiker der ganzen Welt in fieberhaftes Suchen versetzte. Hilbert wollte einen Lösungsweg wissen, der jeder korrekt formulierten mathematischen Aussage sofort entweder mit einer eindeutigen Antwort oder mit dem Beweis der Unrichtigkeit begegnen könnte. Hilbert dachte sich diesen Lösungsweg als ein Werkzeug für den Alltag des Mathematikers, als eine universelle Entscheidungshilfe, die eben jede mathematische Aussage sofort überprüfen könnte. Fortan suchten die Mathematiker der ganzen Welt nach der Lösung des Hilbert'schen Problems, nach der Lösung des Entscheidungs-Problems.

1936 bewies der englische Mathematiker ALAN TURING, daß es eine solche Lösungsvorschrift nicht geben kann. Turing bewies, daß jedes mathematische Problem ein individuelles Problem ist und einen individuellen Lösungsweg ebenso braucht, wie auch einen individuellen Beweis seiner Richtigkeit.

Um diesen Beweis zu führen, bediente Turing sich einer Maschine, die es noch gar nicht gab, der heute so genannten Turing-Maschine. Turing schlug vor, man solle sich ein Band vorstellen, das in beiden Richtungen unendlich lang ist. Dieses Band sei in unendlich viele gleich große Felder aufgeteilt, von denen jedes genau ein Zeichen eines bestimmten, genau definierten Alphabets aufnehmen kann. Dabei ist die Anzahl der Zeichen, die das Alphabet enthält zunächst von untergeordneter Bedeutung. Weiterhin schlug Turing vor, man solle davon ausgehen, daß ein Schreib-Lesekopf auf das Band aufgesetzt werden könne, der in der Lage ist, das jeweilige Zeichen des Feldes, auf dem er sich gerade befindet, zu erkennen und auf Anweisung zu verändern. Diese Anweisungen soll die Maschine aus einer Art Tabelle entnehmen, wobei die Anweisung davon abhängig sei, welches Zeichen der Lesekopf dem aktuellen Feld entnehme.

Quelltext Service

Damit hatte Turing quasi 'im Kopf' eine sogenannte »von Neumann«-Maschine entworfen und beschrieben, einen sequentiellen Computer, wie wir ihn heute millionenfach benutzen. Natürlich steigt die Komplexität dieser Maschine, und damit die Komplexität der Aufgaben, die sie ausführen kann, mit der Anzahl der Zeichen des Alphabets und der Größe der Tabelle, bzw. der darin enthaltenen Zustände, die diese Maschine auf das jeweils gelesene Zeichen verschieden reagieren lassen.

Das ist, sehr vereinfacht, der Beweis, daß es keinen universellen Algorithmus zur Lösung universeller Probleme geben kann.

Turings Gedanke war nun, daß eine bestimmte so definierte Maschine auch nur bestimmte Aufgaben lösen kann. Zum Beispiel kann eine Maschine mit einem Alphabet aus zwei Zeichen und zwei Zuständen zur Reaktion in der Tabelle wesentlich weniger verschiedene Aufgaben lösen, als eine Maschine mit 20 oder 200 oder noch mehr Zeichen im Alphabet. Mit anderen Worten: Jede so definierte Maschine kann nur Aufgaben lösen, die dieser Definition entsprechen. Das ist, sehr vereinfacht, der Beweis, daß es keinen universellen Algorithmus zur Lösung universeller Probleme geben kann. Es ist, so komplex man die Maschine auch gestaltet, stets ein Problem denkbar, das noch komplexer ist !

Eine Konsequenz aus Turings Überlegungen ist, daß es Probleme gibt, die vom Computer nicht lösbar bzw. nicht berechenbar sind. Tatsächlich sind nur alle diejenigen Probleme von einem Computer berechenbar, die sich auf einer Turing-Maschine darstellen lassen !

Ein berühmtes Problem der Informatik ist der Hilbert'schen Forderung sehr ähnlich, das Halte-Problem. Dabei wird nach einem Algorithmus gefragt, der es erlaubt, jedes beliebige Programm daraufhin zu untersuchen, ob es nach einer endlichen Zahl von Schritten ordentlich terminiert oder nicht. Auch dieses Problem ist nicht lösbar, weil nicht auf einer Turing-Maschine darstellbar ! Der Mathematiker TIBOR RADO hat dieses Halte-Problem wie folgt definiert : Man nehme eine Turing-Maschine mit dem Alphabet NULL und EINS und der Menge von 'N' Zuständen . Der Schreib-Lesekopf werde auf eine beliebige Stelle des unendlichen Bandes gesetzt und die Maschine bekomme die Anweisung wie folgt zu verfahren : Lese das Zeichen auf dem Band / arbeite entsprechend der aktuellen Zustandsnummer wie folgt / schreibe entsprechend der Zustandsnummer ein Zeichen auf das Band / stelle fest, ob der nächste Schritt nach links oder

Stichworte

- Turing-Maschine,
- von Neumann-Maschine

BiberNr.	Schritte	Einsen	Zeit ohne Anzeige	Zeit graph.Anzeige
1	67	0	00.00..00.11	00.00..00.77 Sek.
2	187	0	00.00..00.28	00.00..02.14 Sek.
3	52	0	00.00..00.05	00.00..00.60 Sek.
4	15.589	165	00.00..21.14	00.03..16.69 Sek.
5	134.467	501	00.03..12.45	00.29..13.99 Sek.

Tabelle 1

nach rechts auf dem Band erfolgen soll / stelle fest, welcher Zustand als nächstes eingenommen werden soll. Beginne von vorn.

Dieser Algorithmus setzt natürlich einen exakt definierten Start-Zustand voraus. Der Einfachheit halber nimmt man den Zustand 1. Außerdem muß eine definierte Abbruchbedingung vorhanden sein. Die nachfolgende Turing-Maschine verwendet dazu in der Richtungsangabe ein Zeichen, das nicht im Alphabet als Schreib- oder Lesezeichen definiert ist. Nun fordert Rado, daß man unter allen möglichen Maschinen mit der definierten Anzahl von 'N' Zuständen diejenige aussuchen soll, die nach einer endlichen Anzahl von Schritten die größte Anzahl an Einsen produziert hat. (Bei dem Alphabet aus NULL und EINS) Diese Funktion F(N) kann nicht für jedes beliebige 'N' berechnet werden.

1983 wurde ein Wettbewerb ausgeschrieben, der diejenige Turing-Maschine mit N = 5 Zuständen herausfinden sollte, die Rado's Anforderungen erfüllt. Rado selbst nannte dies Maschine 'busy Beaver - fleißiger Biber'.

Die Suche nach einem fleißigeren Biber kann aber zu einer wahren Sisyphusarbeit werden.

Der bisher fleißigste dieser Biber produziert nach 134.467 Schritten 501 Einsen, kann aber nicht den Anspruch erheben der wirklich fleißigste seiner Gattung zu sein. Die Suche nach einem fleißigeren Biber kann aber zu einer wahren Sisyphusarbeit werden. Bei N = 5 Zuständen existieren genau 63.403.380.965.376 mögliche Turing-Maschinen !

Das nachfolgende Programm 'BEAVER' erlaubt es 'nichtsdestotrotz' diese Maschinen zu untersuchen. Aber, wie gesagt, dabei muß schon einiges an Zeit investiert werden. Das Programm enthält fünf vordefinierte Biber die unterschiedlich viel bzw. wenig leisten und dafür unterschiedlich viel Zeit brauchen. Hier die Werte:

In der Zeitangabe der Tabelle 1 bedeutet die Zahl neben dem .. die volle Sekundenzahl, die Zahl rechts daneben die Hundertstel Sekunden. Diese Zeiten haben die entsprechenden Biber auf einem PC mit einer 8086 CPU mit einer Taktfrequenz von 9,65 MHz erreicht.

Wie aus der Zeit für die graphische Ausgabe des Bibers Nr. 5 zu ersehen ist - 29 Minuten, 13 Sekunden und 99 Hundertstel - kann das Untersuchen der diversen Biber schon wesentlich mehr als nur einen Abend füllen. Das gilt natürlich um so mehr, wenn man bei der Beobachtung des Bibers auf den implementierten Einzelschrittmodus zurück greift.

Der FORTH-Biber arbeitet de facto exakt so wie Rado ihn gefordert hat. Trotzdem benutzt er zum Beispiel ein anderes Alphabet. NULL und EINS sind hier die Zahlen 95 und 73, die ASCII-Werte für '_' und 'I'. Damit läßt sich bei der 'graphischen' Ausgabe der Arbeit des Bibers der Inhalt des Bandes via TYPE relativ schnell und ohne zusätzliche Übersetzungsarbeit auslesen und auf den Bildschirm bringen. Das 'leere' Band ist im FORTH-Biber bereits voller Nullen, bzw. voller '_'. Unser Biber soll sich nicht damit aufhalten, zusätzlich Nullen zu erzeugen, sondern sich auf die eigentliche Arbeit konzentrieren und Einsen produzieren. Das unendliche Band wird hier auch nur durch ein 10 KByte großes Feld simuliert. Bei dem schon angesprochenen Zeitbedarf zumindest einiger Biber, dürfte ein Biber, der dieses Array verläßt in den meisten Fällen ohnehin uninteressant sein. Ich gehe sogar davon aus, daß ein Biber, der schnurstracks aus diesem Array läuft, ganz sicher nicht regulär nach einer endlichen Schrittzahl terminiert.

Der FORTH-Biber ist menügesteuert und läßt, wie bereits erwähnt sowohl einen Einzelschrittmodus als auch verschiedene andere Ausgabemodi zu. Der Biberforscher kann wählen zwischen einer graphischen Ausgabe, die ihn die Arbeit des jeweiligen Bibers visuell verfolgen läßt oder einer Ausgabe, die nur die gerade aktuellen Werte wie Schrittzahl, Anzahl der erreichten Einsen, etc. auf den Bildschirm bringt oder einem Modus, der die erreichten numerischen Werte erst am Ende der gesamten Arbeit anzeigt. Welcher Modus gerade ausgewählt ist, wird innerhalb des Hauptmenüs in 'Highvideo' angezeigt.

Weiterhin kann sowohl einer von fertigen Bibern ausgewählt und auf die Reise geschickt, als auch ein eigener Biber definiert werden. Selbstverständlich ist es möglich, die vordefinierten Biber zu modifizieren und dann zu testen.

In diesem Sinne wünsche ich allen 'Biberforschern' viel Spaß.

Friederich Prinz, Moers

Der fleißige Biber

```
COMMENT:

                Der fleißige Biber
              ======================

          Eine Turing-Maschine nach Tibor Rado


-------------------------------------------------------------------

COMMENT;

( Variablen und Felder )

    VARIABLE band   10000 ALLOT     \ 10 KByte simulieren die Unendlichk.
    VARIABLE biber    30 ALLOT      \ 30 Byte für des Biber's Körper

    VARIABLE band_pos
    VARIABLE biber_pos              \ Positionshalter in band und biber
    VARIABLE b_pos                  \ entspricht biber_pos, wird wegen
                                    \ der sonst negativ auftretenden
                                    \ Seiteneffekte benötigt

    VARIABLE zustand                \ 'Tabellennummer' im Biber
    VARIABLE richtung               \ Arbeitsrichtung auf dem band

    VARIABLE schritte
    VARIABLE einsen                 \ was der Biber leisten soll...

    VARIABLE ende?                  \ ende? = 1 ist Abbruchflag
    VARIABLE trace?                 \ trace? = 1 ist Einzelschritt
    VARIABLE ausgeben?              \ hält den jeweils aktuellen
                                    \ Ausgabemodus fest
    VARIABLE spalte                 \ werden für den Test auf
    VARIABLE zeile          \ 'Highlight' gebraucht

( Biber- und Band-Operatoren )

: biber_lesen   biber_pos @  biber + C@ ; \ ein Byte aus dem Biber lesen
: inc_f biber_pos +! ;                    \ bequemer Incrementor
: b_inc 1 b_pos +! ;                      \ dito
: biber_schreiben b_pos   @  biber + C! ; \ ein Byte i. d. Biber schreib.

: band_lesen    band_pos @  band  + C@ ; \ entspr. Biber_lesen
: band_schreiben  band_pos @  band  + C! ; \ entspr. Biber_schreiben

( Bildschirm Kopf )
: scr_kopf                      \ Cursor ist eingeschaltet
  1 1 AT ." (C) f.e.p"     >BOLD  29 1 AT ." Der fleißige Biber "
         >NORM
  71 1 AT ." V.2.1/'89"          0 3 AT  80 0 DO ." -" LOOP
  30 2 AT ." BANDPOSITION : "      0 4 AT ." SCHRITTE : "
  64 4 AT ." EINSEN : "          29 4 AT ." RICHTUNG : "
   0 6 AT  80 0 DO ." =" LOOP ;

( Ausgabe Modi )

: band_ausgabe  1 ausgeben? !           \ der 'volle' Ausgabemodus wird in
   49 11 AT >BOLD ." AN " >NORM         \ ausgeben? gespeichert, der entspr.
   49 12 AT    ." AUS"                  \ Modus wird in HIGHLIGHT angezeigt
   49 13 AT    ." AUS"  ;

: num_ausgabe   2 ausgeben? !
   49 11 AT    ." AUS"
   49 12 AT >BOLD ." AN " >NORM
   49 13 AT    ." AUS"  ;

: end_ausgabe   3 ausgeben? !
   49 11 AT    ." AUS"
   49 12 AT    ." AUS"
   49 13 AT >BOLD ." AN " >NORM  ;

: trace    trace? @ 0 = IF 1 trace? !
       ELSE        0 trace? ! THEN ;

: trace_aus trace? @ 1 = IF 49 10 AT
         >BOLD ." AN " >NORM ELSE
               49 10 AT
            ." AUS"     THEN ;

( Ausgabe der Werte, entsprechend den gewählten Modi )

: ausgabe_2  ( Ausgabe aller numer. Werte und Richtungsanzeige)
  43 2 AT band_pos @ .  12 4 AT schritte @ .  73 4 AT einsen @ .
  40 4 AT richtung @ 0 = IF ." <--" ELSE ." -->" THEN ;

: ausgabe_3   ( ruft Ausgabe_2,aber erst,wenn der Biber regulär terminiert)
  ende? @ 0 = IF NOOP ELSE ausgabe_2 THEN ;

: ausgabe_1   ( komplette Ausgabe, auch 'graphische' Anzeige des Bibers)
  0 3 AT          \ Cursor auf Anfang 'graphische Ausgabe')
  band_pos @ 40 - band +
  80 TYPE
  ausgabe_2 ;

: ausgabe   ( steuert die gesamte Ausgabe )
   ausgeben? @ DUP 1 = IF ausgabe_1 DROP ELSE
           2 = IF ausgabe_2 ELSE
                  ausgabe_3 THEN THEN ;

( Bildschirm Hauptmenue )
: scr_menu
   32 6 AT >BOLD ." - HAUPTMENUE - " >NORM
   24 7 DO 0 I AT 80 SPACES LOOP  \ außer 'Kopf' Bildschirm löschen
  20 8 AT ."  fertigen Biber wählen          --> 1 "
  20 9 AT ."  Biber editieren                --> 2 "
  20 10 AT ."  TRACE / Einzelschritt          --> 3 "
  20 11 AT ."  Ausgabe inclusive Band         --> 4 "
  20 12 AT ."  Nur numerische Ausgabe         --> 5 "
  20 13 AT ."  Numer. Ausgabe am Ende         --> 6 "
  20 14 AT ."  Biber arbeiten lassen          --> 7 "
  20 15 AT ."  Programm Ende                  --> 8 " ;

( Hier arbeitet der Biber ! )
( ========================== )
: arbeite
   5000 band_pos !                      \ in Bandmitte aufsetzen
    0 einsen !   0 schritte !           \ noch ist nichts getan
    0 ende? !                           \ kein Abbruch
    0 zustand !                         \ Startzustand ist NULL
    band 10000 95 FILL                  \ band komplett mit '_' gefüllt
( ========================================== )
   BEGIN
    zustand @  6 * 1 + biber_pos !      \ Adresse im Biber berechnen !
( ========================================== )
    band_lesen  DUP                     \ Band lesen
     73 = IF 3 biber_pos +! THEN        \ Adresse im Biber korrigieren
( ========================================== )
    biber_lesen DUP                     \ Biber lesen
    ROT                                 \ für Vergleich zurecht legen
    < > IF                              \ prüfen: Band u. Biber versch.?
     DUP band_schreiben                 \ dann Biber auf Band übertragen
     DUP 95 = IF -1 einsen +! THEN      \ bei '93' Var. einsen decrement
       73 = IF  1 einsen +! THEN ELSE   \ sonst Var. einsen increment.
       DROP            THEN
( ========================================== )
    1 schritte +!                       \ in jedem Fall schritte incr.
( ========================================== )
    inc  biber_lesen DUP 47 = IF        \ HALT - Befehl gelesen ?
     1 ende? ! THEN                     \ dann Flag in ende? setzen
( ========================================== )
    DUP 0 = IF -1 band_pos +!           \ '_'- band_pos decrementieren
         0 richtung ! DROP ELSE         \ Richtung ist 'links'
    1 = IF  1 band_pos +!               \ 'I'- band_pos incrementieren
         1 richtung ! THEN THEN         \ Richtung ist 'rechts'
( ========================================== )
    inc biber_lesen zustand !           \ neuen Zustand auslesen
    trace? @ 0 = IF NOOP  ELSE          \ kein Einzelschritt? dann weiter
          KEY DROP THEN                 \ sonst auf belieb. Taste warten
    ende?  @ 1 = IF ausgabe_1 EXIT THEN  \ Abbruch_Flag gesetzt ? dann
                                        \ noch einmal alle Werte ausgeben
    ausgabe                             \ Ausgabe entspr. akt. Modus
    KEY? IF ausgabe_1  EXIT THEN        \ 'Einfach so' aussteigen ?
  AGAIN ;

( Biber Tabelle füllen, Werte aus Biber auf dem Bildschirm anzeigen  )
( Das Biber_ARRAY wird komplett ausgegeben um ein Editieren des Bibers )
( zu ermöglichen )
( Highlight? testet, ob Spalte und Zeile in Angabe mit dem Editor-Befehl )
( übereinstimmen und liefert ein 'Flag' für >BOLD                 )

: h_light?
   zeile @  #line @ = IF spalte @ #out @ = IF >BOLD THEN THEN ;


: tab_fuellen
   1 biber_pos !                \ Startpos. 1 im Biber
   10  10  0 DO
    DUP 38 SWAP AT    biber_lesen
               h_light?
               DUP 73 = IF 1 ." * DROP ELSE
               DUP 95 = IF 0 ." * DROP ELSE
                         ." * THEN THEN >NORM
    DUP 54 SWAP AT inc biber_lesen
               h_light?
               0 = IF ." links " ELSE
               biber_lesen
               1 = IF ." rechts" ELSE
                  ." HALT !" THEN THEN >NORM
    DUP 74 SWAP AT inc biber_lesen 1 +
               h_light? ." " >NORM  1 +
            inc LOOP   DROP ;

( Biber Tabelle anlegen )
: tab_anlegen
   24  7 DO 0 I AT  80 SPACES LOOP      \ Bildsch.-Bereich gelöscht
   32  6 AT >BOLD  ." - Biber Edit - " >NORM
    0  8 AT >UL
   ." Zust. alt      - lesen -      -schreibe-      -Richtung-      Zust. neu"
        >BOLD                       \ Tabellen-Überschrift
    4 10  6 1 DO  2DUP AT I . 1 + 2DUP AT  1 + LOOP
   21 10  6 1 DO  2DUP AT ." 0" 1 + 2DUP AT ." 1"  1 + LOOP
        >NORM  2DROP 2DROP
  tab_fuellen
   23 21 AT >BOLD
   ." 'H' oder 'h' für Richtung = 'HALT'"
   23 23 AT
    "     <ENTER>  = ENDE EDIT" >NORM ;
COMMENT:
  Hier folgen die Worte die die Steuerung der Eingaben in die Bibertabelle
  übernehmen. Nullen und Einsen, die eigentlich auf dem Band stehen sollen,
```

werden als 95 = '_' = '0' und 73 = 'I' = '1' auf das Band geschrieben. Diese Konvertierung ist vor allem deshalb sinnvoll, weil die 'graphische' Ausgabe des Bibers dann nur noch via TYPE einen jeweils definierten Bildschirmabschnitt zu lesen und darzustellen braucht. Das ist in Bezug auf die Ausgabegeschwindigkeit wesentlich effektiver, als wenn während des Auslesens diese Umwandlung vorgenommen und jedes Zeichen einzeln konvertiert werden müßte.
COMMENT;

```
( Wert für SCHREIBEN einlesen )
: schreiben?
  DUP 48 ( = 0 ) = IF 95 biber_schreiben tab_fuellen DROP ELSE
    49 ( = 1 ) = IF 73 biber_schreiben tab_fuellen THEN THEN ;

( Wert für RICHTUNG einlesen )
: richtung?
  DUP  72 ( = H ) = IF 47 biber_schreiben tab_fuellen DROP ELSE
  DUP 104 ( = h ) = IF 47 biber_schreiben tab_fuellen DROP ELSE
  DUP 108 ( = l ) = IF  0 biber_schreiben tab_fuellen DROP ELSE
  DUP  76 ( = L ) = IF  0 biber_schreiben tab_fuellen DROP ELSE
  DUP 114 ( = r ) = IF  1 biber_schreiben tab_fuellen DROP ELSE
  DUP  82 ( = R ) = IF  1 biber_schreiben tab_fuellen DROP ELSE
 DROP
  THEN THEN THEN THEN THEN THEN ;

( Wert für ZUSTAND einlesen )
: zustand?
  48 - DUP 1 < IF DROP NOOP ELSE
     DUP 5 > IF DROP NOOP ELSE
  1 - biber_schreiben tab_fuellen THEN THEN ;

( Editieren ermöglicht den Zugriff auf die einzelnen Zellen des BiberArray )
: editieren
  10 zeile !  38 spalte !            \ 'Highlight' Werte
  tab_anlegen  tab_fuellen
  1 b_pos !
  BEGIN
   tab_fuellen
   KEY DUP 13 = IF DROP EXIT ELSE      \ <CR> = Editor verlassen
         schreiben? b_inc \ sonst nach Schreiben fragen
         THEN
         54 spalte !          \ nächste Spalte speichern
   tab_fuellen
   KEY DUP 13 = IF DROP EXIT ELSE
         richtung?  b_inc
         THEN
         74 spalte !
   tab_fuellen
   KEY DUP 13 = IF DROP EXIT ELSE
         zustand?  b_inc
         THEN
   b_pos @     30 > IF 1 b_pos !  THEN   \ Array nicht verlassen !!!
         38 spalte !
   zeile @                    \ die Zeile neu berechnen
   1 +                        \ Zeile incrementieren
   DUP 19 > IF DROP 10 zeile ! ELSE     \ Tab.Ende ? zur. z. Anfang
   zeile !                THEN     \ sonst neue Zeile speichern
  AGAIN ;
```

COMMENT:
Fünf vordefinierte Biber:
==========
Die Aufteilung entspricht der Aufteilung in der Tabelle, b.z.w. im ARRAY Biber. Die oberste Zeile bedeutet: wenn NULL auf dem Band gelesen wurde, dann schreibe NULL, gehe nach RECHTS und nehme als nächsten Zustand den ZUSTAND 1 an (Beispiel aus dem Biber Nr. 1). Die nächste ist das Adäquat zu einer auf dem Band gelesenen EINS. Die beiden Zeilen darunter entsprechen biber_pos Anweisungen im Zustand Nummer 2 u.s.w.
COMMENT;

```
: biber_1  1 b_pos !
 95 biber_schreiben b_inc  1 biber_schreiben b_inc  1 biber_schreiben b_inc
 95 biber_schreiben b_inc  0 biber_schreiben b_inc  0 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  2 biber_schreiben b_inc
 95 biber_schreiben b_inc 47 biber_schreiben b_inc  0 biber_schreiben b_inc
 95 biber_schreiben b_inc  0 biber_schreiben b_inc  2 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  3 biber_schreiben b_inc
 95 biber_schreiben b_inc  0 biber_schreiben b_inc  3 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  4 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  0 biber_schreiben b_inc
 95 biber_schreiben b_inc  0 biber_schreiben b_inc  4 biber_schreiben b_inc ;

: biber_2  1 b_pos !
 95 biber_schreiben b_inc  1 biber_schreiben b_inc  1 biber_schreiben b_inc
 95 biber_schreiben b_inc  0 biber_schreiben b_inc  0 biber_schreiben b_inc
 95 biber_schreiben b_inc  1 biber_schreiben b_inc  2 biber_schreiben b_inc
 95 biber_schreiben b_inc 47 biber_schreiben b_inc  0 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  3 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  4 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  0 biber_schreiben b_inc
 95 biber_schreiben b_inc  0 biber_schreiben b_inc  3 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  2 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  4 biber_schreiben b_inc ;

: biber_3  1 b_pos !
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  1 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  0 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  2 biber_schreiben b_inc
 95 biber_schreiben b_inc  1 biber_schreiben b_inc  4 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  3 biber_schreiben b_inc
 95 biber_schreiben b_inc  1 biber_schreiben b_inc  0 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  1 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  3 biber_schreiben b_inc
 95 biber_schreiben b_inc 47 biber_schreiben b_inc  2 biber_schreiben b_inc
 95 biber_schreiben b_inc  1 biber_schreiben b_inc  1 biber_schreiben b_inc ;

: biber_4  1 b_pos !
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  1 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  0 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  3 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  2 biber_schreiben b_inc
 73 biber_schreiben b_inc 47 biber_schreiben b_inc  3 biber_schreiben b_inc
 95 biber_schreiben b_inc  0 biber_schreiben b_inc  1 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  4 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  2 biber_schreiben b_inc
 95 biber_schreiben b_inc  0 biber_schreiben b_inc  0 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  3 biber_schreiben b_inc ;

: biber_5  1 b_pos !
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  1 biber_schreiben b_inc
 95 biber_schreiben b_inc  0 biber_schreiben b_inc  2 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  2 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  3 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  0 biber_schreiben b_inc
 95 biber_schreiben b_inc  1 biber_schreiben b_inc  1 biber_schreiben b_inc
 95 biber_schreiben b_inc  1 biber_schreiben b_inc  4 biber_schreiben b_inc
 73 biber_schreiben b_inc 47 biber_schreiben b_inc  2 biber_schreiben b_inc
 73 biber_schreiben b_inc  0 biber_schreiben b_inc  2 biber_schreiben b_inc
 73 biber_schreiben b_inc  1 biber_schreiben b_inc  0 biber_schreiben b_inc ;


( Biber - Menue , die vorgefertigten Biber können gewählt werden )

: biber_waehlen
   24 7 DO 01 AT  80 SPACES LOOP       \ Bildsch.-Bereich gelöscht
   32  6 AT >BOLD ." - Biber Wahl - " >NORM
   15 12 AT ." Biber - 1 ,     67 Schritte , keine Einsen  --> 1"
   15 13 AT ." Biber - 2 ,    187 Schritte , keine Einsen  --> 2"
   15 14 AT ." Biber - 3 ,     52 Schritte , keine Einsen  --> 3"
   15 15 AT ." Biber - 4 ,  15.589 Schritte ,  165 Einsen  --> 4"
   15 16 AT ." Biber - 5 , 134.467 Schritte ,  501 Einsen  --> 5"
   15 18 AT >BOLD
         "          bitte einen 'Biber' auswählen ...          "
   BEGIN
    KEY
    DUP 13 = IF       DROP EXIT  ELSE  48 -
    DUP  1 = IF biber_1 DROP EXIT  ELSE
    DUP  2 = IF biber_2 DROP EXIT  ELSE
    DUP  3 = IF biber_3 DROP EXIT  ELSE
    DUP  4 = IF biber_4 DROP EXIT  ELSE
        5 = IF biber_5 DROP EXIT  THEN THEN THEN THEN THEN THEN
  AGAIN ;


( Das Hauptmenue... )

: beaver
   dark  scr_kopf  1 ausgeben? !
   BEGIN  scr_menu  trace_aus
            ausgeben? @
            DUP 1 = IF band_ausgabe ELSE
            DUP 2 = IF  num_ausgabe ELSE
              3 = IF  end_ausgabe THEN THEN THEN
   KEY 48 -
   DUP 1 < IF DROP            ELSE
   DUP 8 > IF DROP            ELSE
   DUP 1 = IF DROP  biber_waehlen ELSE
   DUP 2 = IF DROP  editieren   ELSE
   DUP 3 = IF DROP  trace       ELSE
   DUP 4 = IF DROP  band_ausgabe  ELSE
   DUP 5 = IF DROP  num_ausgabe  ELSE
   DUP 6 = IF DROP  end_ausgabe  ELSE
   DUP 7 = IF DROP  arbeite      ELSE
     8 = IF DROP  ABORT       THEN THEN THEN THEN THEN
                  THEN THEN THEN THEN THEN
  AGAIN ;
```

Neue Datentypen in FORTH

Konrad Scheller, Forchheim

Immer wenn ich mit einem Pascal-Fan diskutiere, dann kommt irgendwann im Verlauf des Disputs das Argument: "Außer Zahlen kannst du in FORTH ja gar nichts verarbeiten! Sieh dir dagegen mal meine Möglichkeiten an: Aufzählungstypen, Strings, Records,... "

Das ging mir nach einiger Zeit gehörig auf den Wecker. Ich setzte mich also an meinen Computer und das Ergebnis war der folgende Code für die Programmierung von Aufzählungstypen.

Was halten Sie davon, wenn Sie programmieren können

```
DECLARE  " grün"  " gelb"  "rot"  AS
Ampelfarben
' Ampelfarben  VAR  Ampel
" grün"  Ampel  T!

Ampel  T@  ". <ret>
grün ok
```

Vorsicht: Es handelt sich nicht um Stringvariablen! Das Abspeichern eines Typs (in diesem Fall "grün", es könnte aber auch »ultramarinblauzwitscheringrün« heißen, wenn man es vorher so deklariert) benötigt nämlich nur 4 Bytes, egal wie lang der Typenname ist. Eine Zuweisung eines Wertes, der zuvor nicht deklariert wurde, ist nicht möglich und führt zu einer Fehlermeldung.

Wollen Sie feststellen, ob ein Ausdruck (z.B. "gelb") in einem Typ enthalten ist, verwenden Sie das Wort *MEMBER?* Ein Beispiel.

```
 " gelb"  Ampelfarben MEMBER? .
-1 ok
```

Quelltext Service

Meistens will man jedoch nicht nur einzelne Daten erfassen, sondern ganze Gruppen davon. Auch dazu wieder ein Beispiel.

Ich programmiere eine Lagerverwaltung und möchte zu einem Artikel den Namen, den Preis, die Lagermenge und den Lieferanten abspeichern. Das geht nun so:

```
DECLARE  " Müller"  " Meier"
" Schmitt"  "Schulze"  AS Lieferanten

RECORD  Artikeltyp
    String  Name
    Integer       Bestand
    Double        Preis
    Lieferanten   Bezugsquelle
END-RECORD
```

Bis jetzt habe ich aber noch keine Variable definiert - nur Datentypen!! Eine Variable bastle ich mir nun so...

```
'  Artikeltyp   VAR  Artikel
```

(Beachten Sie bitte das »Tick«!). Jetzt erst wird die Variable Artikel angelegt. Nun kann man auf einen Artikel als Ganzes zugreifen.

```
Artikel .
 (liefert die Grundadresse der
Variable)
Artikel Name "@ ".
 (druckt den Artikelnamen)
Artikel Preis 2@ D.
 (druckt den Preis)
Artikel Bestand @ .
 (zeigt den Bestand)
Artikel Quelle T@ ".
 (druckt den Lieferanten aus)
```

Wir sind aber noch nicht fertig mit unseren Möglichkeiten.

Um bei dem letzten Beispiel zu bleiben: Man will normalerweise nicht nur auf einen einzigen Artikel, sondern auf ein ganzes Lager zugreifen. Auch das ist möglich.

Erstmal den alten Artikel vergessen:

```
FORGET Artikel
```

Nun definieren wir uns ein Lager:

```
200  ' Artikeltyp ARRAY Lager
```

und nun die Variable *Artikel*:

```
Lager  VAR Artikel
```

Jetzt wird bei Ansprechen eines Artikels durch den Einfluß des Arrays noch eine Artikelnummer auf dem Stack erwartet. Das ganze funktioniert nun so:

```
1 Artikel Name "@ ".
(druckt den Namen des 1. Artikels)
33 Artikel Bestand @.
(gibt an, wieviel vom Artikel Nr. 33
noch vorhanden ist)
```

Bei der Implementierung habe ich darauf geachtet, daß man die Datentypen untereinander kombinieren und verschachteln kann. Man kann also Records bilden usw. Die Möglichkeiten sind nur durch Speicherplatz und Stacktiefe begrenzt - typisch FORTH!

Noch ein letztes Beispiel:

```
RECORD DATE
   INTEGER DAY   INTEGER MONTH
                 INTEGER YEAR
END-RECORD

DECLARE " roman"  " protestantic"
        " jewish" " islam"
        " other" " free"
AS  Religions
RECORD PERSON
   STRING       NAME
   DATE         BIRTH
   RELIGIONS    RELIGION
   DOUBLE       MONEY
END-RECORD
' PERSON  VAR  WORKER
```

Aber jetzt denken Sie sich gefälligst selber was aus.

Stichworte

- » Datentypen,
- » Records,
- » Aufzählungstypen

DATE	adr-record	6	link	
DAY	adr-integer	2	0	link
MONTH	adr-integer	2	2	link
YEAR	adr-integer	2	4	0

Abbildung 1

Wie funktioniert's?

Nun, so ganz einfach ist das natürlich nicht mehr zu erklären. Schon mal die Datenstrukturen sind etwas schwieriger.

Betrachten Sie einmal Abbildung 1. Hier sehen Sie einen typischen, einfachen Record namens DATE: Er besteht aus DAY, MONTH und YEAR.

Diese sind alles normale Integers, da für Tag, Monat, und Jahr im allgemeinen keine Zahlen gebraucht werden, die größer als 32768 sind. Auf der Abbildung bedeuten »DAY«, »MONTH« und »YEAR« sowie »DATE« den jeweiligen Header des Wortes, sind also für die Funktion des Records ohne Bedeutung. Hinter diesem Header folgt erstmal der *Typ* dieses Wortes. Bei *Date* also die Adresse von *Record*, bei den anderen Worten jeweils die Adresse von *Integer*. Die zweite Zelle im Parameterfeld enthält die Länge, die der Variablentyp benötigt. Das sind hier, da *Day*, *Month* und *Year* jeweils Integers sind, 2 Bytes. In der nächsten Zelle steht also jeweils eine 2.

Im Parameterfeld des neuen Records (Date) steht analog dazu die Länge des Records, also eine 6 (2+2+2).

Die dritte Zelle im Parameterfeld enthält den *Offset* des Wortes innerhalb eines Records. Unter Offset versteht man eine Zahl, die zu einer Adresse dazuaddiert wird, um die Endadresse zu erhalten. In diesem Fall ist es so, daß man eben eine Null addieren muß, um von der Grundadresse zum Tag zu kommen, eine zwei um auf den Monat, und eine vier, um auf das Jahr. Nochmal: Wenn später die Variable tatsächlich im Speicher angelegt wird, so werden entsprechend der Länge von *Date* sechs Bytes im Speicher reserviert. Um nun innerhalb dieses reservierten Bereichs den Tag, den Monat und das Jahr zu finden, wird dieser Offset zu der Grundadresse hinzuaddiert.

In der letzten Zelle jeder Zeile findet man ein sogenanntes Link-Feld. Das ist nichts anderes als eine Adresse, die auf eine Speicherzelle zeigt, die wiederum auf eine Speicherzelle zeigt, usw. Das ist notwendig, weil ja die einzelnen Worte auch gefunden werden müssen!

Konkret ausgedrückt heißt das: Das Linkfeld von *Date* zeigt auf die Parameterfeldadresse von *Day*, dessen Linkfeld auf die PFA von *Month*, dessen Linkfeld auf die PFA von *Year*, und das Linkfeld von *Year* enthält eine Null, da es das letzte in dieser Reihe ist.

Wie wird diese Struktur aufgebaut?

Das Wort *RECORD* macht eigentlich nicht viel. Erstmal erstellt es einen neuen Eintrag im Wörterbuch - mit *CREATE* - und dann schreibt es seine eigene Codefeldadresse zwecks späterer Identifizierung ins Parameterfeld. Mit *HERE 0* , wird eine Speicherzelle freigelassen, die später von END-RECORD mit der Länge des Records aufgefüllt wird. Die 9, die jetzt im Listing steht, dient nur der Fehlerkontrolle. Dann wird auf die gleiche Weise eine Speicherstelle für das erste Linkfeld freigehalten. Die Null am Ende des Wortes ist die anfängliche Länge des Records.

Wird nun ein Wort wie *Integer*, *Double*, *String* oder auch ein mit AS selbstdefiniertes Typwort, aufgerufen, so erledigt das Wort *HEADER* den Aufbau des Parameterfelds. Zuerst wird die »Typadresse« des Wortes einkompiliert, also die von Integer usw. Dann wird über die PFA der Typenadresse die Länge dieses Typs geholt und ebenfalls einkompiliert. Jetzt wird die bisherige Recordlänge (= jetziger Offset) im Wörterbuch abgelegt. Diese Recordlänge wird nun mit der Typlänge von vorhin zusammengezählt, um die neue Recordlänge zu erhalten. Schließlich sorgt das Wort *LINK* für den korrekten Aufbau des Linkfeldes.

Nun haben wir die Datenstruktur, die nötig ist, um einen solchen Record vollständig zu beschreiben. Das Problem reduziert sich darauf, diese Angaben in eine Variable umzuwandeln.

Diese Aufgabe wird zum größten Teil von *MAKE-ROOM* erledigt. Leider ist dieses Wort ein wenig groß geraten und deshalb nicht einfach zu verstehen. Deshalb will ich hier keine Schritt-für-Schritt Beschreibung vornehmen, sondern den Vorgang nur kurz in Worten beschreiben.

Make-Room geht von der PFA des Datentyps aus, zu dem die Variable gebildet werden soll. Hier steht als erstes der Variablentyp. Ist dieser einer der bekannten (INTEGER, DOUBLE, STRING usw.) ist der Fall ganz einfach. Dann wird der entsprechende Platz reserviert - bei den Strings noch ein »link« gesetzt - und die Sache hat sich. Weit weniger lustig geht's zu, wenn ein Record auftaucht. Ein Record besteht ja wieder aus anderen Datentypen! Zuerst wird das erste Linkfeld gelesen. Das zeigt nun wieder auf die PFA des ersten Bestandteils des Records. Dieser Bestandteil muß nun genauso von *Make-Room* untersucht werden wie der Record selbst. Wie ist das einfach zu lösen?

Make-Room ruft sich selber auf - Rekursion. Ist dann dieser Bestandteil erledigt, wird von dessen Linkfeld aus der nächste unter die Lupe genommen und so weiter. Das geht solange, bis das Linkfeld gleich Null ist, soll heißen, hier ist das Record zu Ende. Was passiert, wenn *Make-Room* auf ein Array stößt? Zuerst wird die Anzahl der Elemente gelesen, und dann wird *Make-Room* wieder sooft mittels *DO..LOOP* und *RECURSE* aufgerufen, wie es dieses angibt. Auf diese Weise wird der gleiche Vorgang mehrmals ausgeführt.

Wenn es keiner dieser bekannten Datentypen ist, was dann ? Tja, dann kann es sich nur um einen mit AS selbstdefinierten handeln. Der jedoch

enthält in seiner PFA die Adresse von AS, so daß ein *RECURSE* das Problem für uns löst.

Genau darin liegt jedoch die Gefahr. Wird *Make-Room* auf eine »wilde« - sprich auf eine beliebige Adresse - angesetzt, so wird immer wieder ein *RECURSE* ausgeführt, und eines schönen Tages läuft mindestens ein Stack über...dann kann man beobachten, wie sich das eigene FORTH beim Absturz verhält.

OK. Nun ist also die Variable auch angelegt. Was tut nun diese, wenn man ihren Namen aufruft? Nicht viel. Sie legt lediglich die Anfangsadresse des reservierten Bereichs auf den Stack. (Ausnahme: ARRAY's multiplizieren nochmal...). Auf die einzelnen Bestandteile einer Variablen wird dann mit Hilfe der durch Integer usw. definierten Worte zugegriffen. Die zählen zu der Grundadresse nämlich noch den Offset (ich erklär's nicht schon wieder, was das ist) hinzu.

Das war alles. Einfach, nicht????

Noch ein paar Einbauhinweise. Bedauerlicherweise gibt es verschiedene FORTH-Standards, also...

Fig-FORTH Benutzer, aufgepaßt! Es gilt folgendes:

- ['] durch ' ersetzen.
- >BODY weglassen
- CREATE durch <BUILDS ersetzen.
- EXIT durch ;S ersetzen.
- ?DUP durch -DUP ersetzen.

FORTH79 Benutzer: Im wesentlichen gilt dasselbe wie bei Fig-FORTH, nur ist das ganze nicht so sicher, da FORTH79 das LAST usw. nicht spezifiziert. Die meisten 79er-Systeme arbeiten jedoch noch so wie Fig-FORTH.

Andere Systeme: Woher soll ich denn das wissen?

Also dann... Bis zum nächsten Mal!

Möget Ihr nie von FORTH genug haben!

Beschreibung der einzelnen Worte

'LAST (-- addr)
Gibt die Parameterfeldadresse des zuletzt definierten Wortes an, z.B. die gerade kompilierte.

[LAST] (--) (-- addr)
setzt die Adresse des letzten definierten Wortes als Zahl in die Definition ein.

RECURSE(--)
Setzt die Adresse des letzten definierten Wortes als Wortaufruf in die Definition ein - entspricht einem rekursiven Aufruf.

PRESET (-- addr)
Eine Variable, die eine voreingestellte Adresse enthält, deren Inhalt später zum Offset hinzugezählt wird, wenn sie nicht Null ist.

OFFSET (addr pfa -- addr2)
Berechnet aus der Grundadresse die Offsetadresse.

LINK (linkadr1 --- linkadr2)
Baut das Linkfeld auf.

HEADER (link offset pfa -- link2 offset2)
Bestimmt das Laufzeitverhalten eines Datentypes. Kreiert ein neues Wort mit einem Header, das dann später den Offset zur Grundadresse hinzuzählt.

NEEDS (n --)
Bestimmt die Länge eines Datentypes. Diese Worte sollten vom User nicht verwendet werden

", (--) (" s -- ")
Kompiliert einen String vom Stringstack ins Wörterbuch.

DECLARE(-- addr n)
Gebraucht in der Form:
DECLARE "string1" ... "stringn" AS name
Wobei name ein neuer Datentyp ist, der jeweils Werte von 'string1' bis 'stringn' annehmen kann, sonst nichts.

AS (addr n --)
Siehe DECLARE

T@ (typvaraddr --) (" -- s ")
Holt einen Datentyp aus der angegebenen Typvariable. Der Typ wird als String auf dem Stringstack abgelegt.

MEMBER?(typaddr -- f) (" s -- ")
Stellt fest, ob der angegebene Datentyp den String s enthält.

T! (typvaraddr --)
Bringt den String s in die Typvariable.

NEXT (typvaraddr --)
Schaltet eine Typvariable auf den nachfolgenden String um. Achtung: Die Strings müssen beim Deklarieren in umgekehrter Reihenfolge - wegen des Stringstacks - angelegt werden, also z.B. Declare " grün" " gelb" "rot" as Ampelfarben

END-RECORD
(recaddr n len link --)
Beendet eine Recorddefinition.

ARRAY (n 'typ --)
Definiert ein Array aus einem Datentyp. Das Array besteht aus n Elementen sowie aus dem Datentyp, dessen Adresse angegeben wird.

VAR ('typ --)
Legt eine Variable an. Der Datentyp muß auf dem Stack liegen.

Anmerkung der Redaktion:

Der Quelltext dieses Artikels benutzt String-Funktionen eines Beitrages des Autors, der noch nicht abgedruckt wurde. Der fehlende Quelltext ist in der Müncher Mailbox zusammen mit dem hier abgedruckten Sourcecode abgelegt oder kann bei der Redaktion angefordert werden. Der angesprochene Beitrag wird in einer der nächsten 'Vierten Dimensionen veröffentlicht.

die Red.

Neue Datentypen in FORTH

Screen # 1

```
// Neue Datentypen in Forth
// Einige Vorbereitungen
: 'Last ( -- adr ) // gibt die Pfa-Adresse des zuletzt
Last @              // definierten Wortes, z.B. des
Link>  >Body        // gerade definierten.
// FigForth und Forth79 Definition:
// : 'Last  Latest Pfa ;
// VolksForth Definition:
// : 'Last   Current @ @ 2+ name> ;

: [Last] ( -- )         ( Compiling )
         ( -- addr )  ( executing )
'Last [Compile] Literal  ;  Immediate

: Recurse ( -- )        ( Ermöglicht rekursive Programmierung )
  'Last 2- ,                ; Immedate
```

Screen # 2

```
// Neue Datentypen

Variable Preset   Preset off

: Offset  ( [addr] pfa -- addr2 )
  4 + @  Preset @ ?dup drop + ;

: link   ( link -- 'link )
  Here 6 - swap ! Here 0 , ;

: Header  ( link offset pfa -- 'link 'offset )
  dup , 2+ @ dup , over , + >r link r> ;

: Second   ( link offset pfa -- 'link 'offset )
  Create Header  does> Offset ;
```

Screen # 3

```
// Neue Datentypen

: needs  ( n -- )
  Create Here , , does> Second ;

1 needs Byte   2 needs Integer
4 needs Double 4 needs String


// Compiliert String vom Stringstack ins Dictionary

: ",  (" s -- ")
  "pop dup >r Here Place  r> 1+ Allot  ;
```

Screen # 4

```
// Aufzählungstypen

: Declare ( -- addr n )
  "ptr @ 8 ;

: as       ( addr n -- )
  8 - ABORT" No 'declare' before 'as' "
  "ptr @ - 2/ dup 0= ABORT" No types declared"
  Create [Last] , 4 ,
         0 DO ", LOOP 0 c,
  does>  Second ;

: t@       ( taddr -- )  (" -- s ")
  2+ @ dup 0= ABORT" Empty Type"
       count "push  ;
```

Die nächste 'Vierte Dimension' erscheint im Dezember

Screen # 5

```
// Aufzählungstypen

: (Member?)  ( Typ -- taddr f )  (" s -- ")
  false swap 6 +
  BEGIN dup count "push  "over
        "= IF swap not swap  "drop EXIT THEN
         "skip  dup c@  0=
  UNTIL  "drop  ;

: Member?    ( Typ -- f )          (" s -- ")
  (Member?) drop ;

: t!          ( taddr -- )         ("s -- ")
 dup @ 2- (Member?) swap 0= ABORT" No required Type"
 swap 2+ ! ;
```

Screen # 6

```
// Aufzählungstypen - Records

: Next        ( tvar -- )
  dup  2+ @ "skip  dup c@ 0= IF drop dup @ 4 + THEN
  swap 2+ ! ;

: Record    ( -- recaddr n len link )
  Create [Last] , Here 0 , 9 Here 0 , 0
  does> Second ;


: End-Record
  rot 9 - ABORT" Record not correct"
  0 rot ! swap ! ;
```

Screen # 7

```
// Arrays

: Array  ( N  'typ -- )
  Create  [Last] ,
          >Body 2dup 2+ @ * ,  ( Länge * n )
          swap , ,             ( Anzahl, LängeElement )
  does>   Create Header
          does>  ( [addr] n pfa -- addr2 )
                 dup 4 + @ >r
                 @ 6 + @ 2+ @ *
                 r> +
                 Preset @ ?dup drop + ;


// Leider etwas unübersichtlich! Tschuldigung!!
```

Screen # 8

```
// Variable anlegen

: make-Room  ( pfa --)
  dup @ dup ['] Integer >body = IF 2drop 0 ,     EXIT THEN
        dup ['] Byte    >body = IF 2drop 0 c,    EXIT THEN
        dup ['] Double  >body = IF 2drop 0 , 0 , EXIT THEN
        dup ['] String  >body = IF 2drop here "link @ ,
                                   "link ! 0 ,   EXIT THEN
        dup ['] as      >body = IF drop , 0 ,    EXIT THEN
        dup ['] Record  >body = IF drop 2-
                                   BEGIN 6 + @ ?dup WHILE dup
                                   Recurse REPEAT EXIT THEN
        dup ['] Array    body = IF drop dup 6 + swap
                                   4 + @ 0 DO dup Recurse LOOP
                                   drop EXIT THEN
   recurse drop ;
```

Screen # 9

```
// Variable anlegen

: var  ( typ -- )
  create  >body , 0 ,
          here 4 - make-room
  does> dup 4 + swap @
        @ ['] array >body = if dup 4 - @ 6 + @
                               2+ @ rot * + then ;
```

Read-only Stringfelder in FORTH

von Karsten Konrad,
Detzelstr.33, 6670
St.Ingbert

In der Praxis tauchen oft zusammengehörige Stringdaten auf, die von einem Programm nicht mehr geändert werden müssen. Das können z.B. Monatsnamen oder die Bezeichnungen für chemische Elemente sein. Die hier vorgestellten Routinen ermöglichen die platzsparende Übergabe solcher Daten im Quelltext.

Die Realisierung im Prinzip

Üblicherweise werden Stringfelder, wie alle Felder in FORTH, auf das größtmögliche Element normiert. Das kann notwendig sein, wenn die Strings veränderlich sind und man somit immer Reserveplatz braucht (etwa bei einer Adressenverwaltung). Bei read-only Strings läuft Reserveplatz jedoch auf Platzverschwendung hinaus, da sich die Länge der Strings ja nie ändert.

Bei meiner Lösung werden die Strings im Speicher direkt hintereinander abgelegt. Natürlich können die Adressen der Strings dann nicht mehr über einfache Multiplikationen berechnet werden, wie etwa in Superstrings. Das Lesen der Adressen muß über spezielle Wörter erfolgen, die sich praktisch durch die Strings hindurchhangeln.

Stichworte

- Read-only Stringfelder,
- unveränderliche Daten,
- volksFORTH

Die Realisierung in volksFORTH'83

Quelltext Service

In Screen 1 sehen Sie ein typisches Beispiel für ein read-only Stringfeld (eigentlich ist es ja kein Feld, aber den passenden Namen kenne ich nicht). Das Wort *"DATA* definiert ein Wort, das seine eigene Parameterfeldadresse auf den Stack legt. Diese Adresse wird von allen Operationen auf ein Stringfeld gebraucht; die Anzahl der Felder und der darin enthaltenen Strings ist übrigens nur vom freien Speicher begrenzt.

Dann folgen die Strings, die mit ," eingetragen werden. Dieses Wort gibt es im volksFORTH bereits, es legt einen String ab *HERE* ins Wörterbuch. Ich habe es in Screen 2 neu definiert, damit es auch noch die Anzahl der eingetragenen Strings mitzählt. Den Abschluß bildet *;DATA*. Es initialisiert einen Pointer, der im ersten Integer des Stringfeldes steht, schreibt die Anzahl der Strings in das zweite Integer und schließt das Feld mit einem Nullbyte als Countbyte ab.

Das Lesen der Stringadressen erfolgt jetzt sequentiell mit *READ* oder wahlfrei mit *ITEM*. Sequentiell bedeutet, daß die Adressen aus einem Feld nach ihrer normalen Reihenfolge ermittelt werden.

READ benutzt dazu einen Pointer, den jedes Datenfeld separat besitzt. Dieser Pointer ist in jedem Stringfeld vorhanden und zeigt auf den gerade zu ermittelnden String. Jedes *READ* verändert den Pointer so, daß er auf den nächsten String zeigt.

Das Verfahren bei *ITEM* gleicht dem normalen Vorgehen bei anderen Feldern: eine Adresse wird einfach über die Nummer des Feldelements ermittelt. Auch hier wird der Pointer verändert: anschließende Stringadressen können dann mit *READ* gelesen werden.

```
Syntax:  <feldname> READ
         ( adr -- adr )

         n <feldname> ITEM
           ( 16b adr -- adr )
```

Übrigens führen sowohl *READ* als auch *ITEM* eine Überprüfung der Feldgrenzen anhand des abschließenden Nullbytes durch. Ein »Überlesen« eines Feldes ist damit unmöglich.

Weitere Benutzerworte

RESTORE setzt den Pointer eines Feldes wieder auf den ersten String. Das Lesen mit *READ* kann dann von vorne anfangen.

#ITEMS ergibt die Anzahl der Strings in einem Feld; wird vor allem als Schleifenindex gebraucht.

```
Syntax: <feldname> RESTORE
        ( adr -- )

        <feldname> #ITEMS
        ( adr -- 16b )
```

Tips und Tricks

Der Pointer eines Stringfeldes steht immer im ersten Integer eines Feldes. Er kann somit wie jede andere Variable mit *PUSH* gerettet werden (*<feldname> PUSH*).

Die Strings müssen nicht unbedingt »read-only« sein. Da ja die Adresse bekannt ist, können schon Änderungen vorgenommen werden. Die Länge eines Strings darf aber dabei nicht verändert werden.

READ ist unter Umständen schneller als eine Routine für normierte Strings. Wer die Routinen für Maschinensprache optimieren will, der sollte mit *NEXT_ITEM* anfangen. Die Definition ohne Errorcheck sieht so aus:

```
I : next_item (adr1 -- adr2) count + ;
```

Quellen

[1] Stringstack von K. Schleisiek, VD II/1

[2] Starting FORTH von Brodie

Read-only Stringfelder in FORTH

Screen # 1

```
\ Loadscreen

Onlyforth

1 2 +thru

\\ kleines Beispiel, ohne großen Gehalt

"Data  maße ," Tera 10^12" ," Giga 10^9 ," Mega 10^6"
            ," Kilo 10^3 ," Hekto 10^2 ," Deka 10^1" ;data

: Sag ( -- ) [compile] Ascii capital maße restore
             maße #items 0
             DO maße read 2dup 1+ c@ =
                IF count type ELSE drop THEN
             LOOP drop ;
```

Screen # 2

```
\ Die "Data-Struktur

| Variable #strings

: "Data  ( -- dat_adr ) Create here 4 allot #strings off ;

: restore  ( dat_adr -- ) dup 4+ swap! ;

: ;data  ( dat_adr -- ) dup #strings @ swap 2+ ! restore
                        0 c, ;

: ,"   ," 1 #strings +! ;

: #items ( dat_adr -- n ) 2+ @ ;
```

Screen # 3

```
\ Lesen der Stringadressen

| : next_item ( adr1 -- adr2 )
              count dup 0= abort" End of Data." + ;

:read ( dat_adr --adr ) dup @ dup next_item rot ! ;

:item ( # dat_adr )
      under 4+ swap 0 ?DO next_item LOOP
      dup next_item rot ! ;
```

Screen # 5

```
\ Loadscreen

Die Definitionen kommen ins Forth

...laden

\\ kleines Beispiel, ohne großen Gehalt

So wird's gemacht.


Sie sehen eine Datenbank für Maße mit positiven Zehnerpotenzen.
Einfach SAG <letter> eingeben, und die Maßeinheit mit dem
Anfangsbuchstaben <letter> wird ausgegeben.

Fast schon wie dBase III.
```

Screen # 6

```
\ Die "Data-Struktur

#STRINGS dient zum Mitzählen der eingetragenen Strings.

"DATA definert ein Stringfeld.

RESTORE setzt den Pointer eines Feldes auf den ersten String.

;DATA schließt eine Felddefinition ab.


," kann wie üblich weiterverwendet werden.
#STRINGS wird nur zur Compilezeit von ;DATA benötigt.

#ITEMS ergibt die Anzahl der Strings in einem Feld.
```

Screen # 7

```
\ Lesen der Stringadressen

NEXT-Item dient zur Berechnung von Adressen in einem Stringfeld.
Zusätzlich werden aber auch die Feldgrenzen überprüft.

READ liest Stringadressen sequentiell aus einem Stringfeld. Der
Pointer wird dabei jeweils einen String weitergesetzt.

ITEM erlaubt einen quasi wahlfreien Zugriff auf einzelne
Strings. Der String mit der Nummer # (0 bis #items-1) wird
im Feld mit der Adresse dat_adr ermittelt und seine
Adresse auf den Stack gelegt.
Da der Pinter hier ebenfalls verändert wird, können darauf
folgende Stringadressen mit READ gelesen werden.
```

Hinweise für Autoren

Auch in Zukunft möchten wir Beiträge veröffentlichen, die Sie uns hoffentlich in großer Zahl liefern werden. Schicken Sie Ihre Manuskripte bitte an die Redaktion der 'Vierten Dimension' D.LUDA Software, Gustav-Heimann-Ring 42, 8000 München 83, Tel. 089/6708355 oder legen Sie sie in der FORTH-Mailbox München 'Konferenz Vierte Dimension' ab (8N1 Tel. 089/7259625).

Am liebsten hätten wir die Manuskripte auf einer Diskette 5 1/4" (360 Kbyte oder 1,2 Mbyte) im IBM-Format oder einer 3 1/2" Diskette (Atari-Format oder 720 Kbyte IBM-Format). Ist Ihnen das nicht möglich, können Sie auch normale Texte auf Papier einsenden. Bei Bildern sollte allerdings darauf geachtet werden, daß ein möglich guter Kontrast vorliegt. Die Arbeiten sollten in dieser Reihenfolge enthalten:

- Kurzer Titel,
- Autor,
- Zusammenfassung (ca. 50 Worte),
- Schlüsselworte (ca. 5), Text,
- Quellenangaben,
- Illustrationen,
- Tabellen,
- Quellcode.

Die Beiträge werden überarbeitet. Falls ein ausführliches Lektorieren erforderlich ist, erhalten Sie vor der Wiedergabe den Beitrag zur Korrektur und Zustimmung zurück. Layouts werden nicht mehr zur Prüfung durch die Autoren vorgelegt. Autoren erhalten auf Wunsch ein kostenloses Exemplar der 'Vierten Dimension' mit ihrem Artikel.

High-Level Interrupts im IBM-volksFORTH

von Frank Stüss

Bei meinen Arbeiten mit FORTH tauchte des öfteren das Problem auf, Interrupts zu benutzen und zu verbiegen. Nicht zuletzt auch deswegen, weil ein Interrupt eine bequeme Schnittstelle zu anderen Sprachen ist. Aus dieser Motivation heraus schrieb ich ein paar Worte, die sich bis jetzt einigermaßen bewährt haben. Es handelt sich um eine von vielen Möglichkeiten FORTH-Worte als Interrupt-Routinen zu benutzen.

Konkret sind die Routinen für's IBM-volksFORTH gedacht. Beim IBM PC sind die Interrupts allgemein in zwei Gruppen zu Unterteilen:

- solche, die per Software oder sonst wie (Prozessor, NMI, ...) aufgerufen werden
- und solche, die von den Interrupt-Controllern erzeugt werden (IRQ's).

Beide Arten erfordern eine unterschiedliche Art der Behandlung. Bei der ersten Sorte brauch man zur Rückkehr vom Interrupt nur IRET . Die letzteren haben die Besonderheit, daß man ein EOI (End_Of_Interrupt hex 20) zu dem Interrupt-Controller schicken muß. Dies läßt sich mit einem OUT Befehl bewerkstelligen. Bei einem XT ist nur ein Interrupt-Controller vorhanden; bei einem AT existieren jedoch zwei solcher Bausteine. Wenn man einen Interrupt vom zweiten Controller abarbeitet, muß man nicht nur dem 2. Controller ein EOI senden, sondern auch dem ersten, weil der zweite an einen IRQ-Kanal des ersten Controllers angeschlossen ist.

Um nun einen Interrupt in High-Level-FORTH auszuführen, müssen am Anfang des Wortes alle eventuell im Wort sich ändernden Register gerettet, und bei der Rückkehr vom Interrupt wiederhergestellt werden, um alle Fälle des Aufrufs abzudecken - auch die reentranten. Die erste Aufgabe erfüllt das Wort INTSTART: welches zusätzlich direkt in den Compile-Mode schaltet. Es erzeugt ein Headerloses Wort, das als erste Befehlsfolge das Retten der Register enthält. Danach kann man nach Herzenslust FORTHen (netter Ausdruck!), jedoch muß das Wort mit einem Returnwort abgeschlossen werden. Es stehen drei solcher Worte zur Verfügung:

- SOFTRET als Abschluß eines Software-Interrupts bzw. für alle Interrupts, bei welchen der Controller nicht seine Beinchen im Spiel hat.

Quelltext Service

- HARD1RET für Interrupt-Controller 1
- HARD2RET für Interrupt-Controller 2

Um Interrupts zu verbiegen steht zusätzlich das Wort ONINT zur Verfügung, welches die Interrupt-Nummer und die Adresse der Interrupt-Routine erwartet. Der jeweilige Interrupt wird auf diese Adresse im FORTH-Segment verbogen. Damit läßt sich nun schon einigermaßen komfortabel arbeiten. Ein Beispiel:

```
$ff here ( Interrupt $ff soll auf
'here' verbogen werden )

Intstart: ." Hallo, ich bin ein
Interrupt !!! " bell softret ;

onint ( jetzt wird er verbogen )
Code testint $ff int next end-code (
ein Test-wort )
testint <ret>
Hallo, ich bin ein Interrupt !!!
{PIIIIEPS}
ok
```

Als Anwendung stehen einige Sachen im Raum: Man kann jetzt sehr bequem von anderen Sprachen aus FORTH-Routinen aufrufen, die als Interrupts realisiert sind.

Über den Autor:

Ich bin 20 Jahre alt und Physikstudent im vierten Semester. Zu FORTH kam ich vor etwa 4 Jahren durch ZECH's sagenhaftes 'erstes' (fig und so). Jedoch waren die ersten Jahre relativ theoretisch, weil ich keinen eigenen Computer hatte, den ich für FORTH begeistern konnte. Die aktive Arbeit mit FORTH begann, als ich vom volksFORTH gehört hatte und endlich einen XT besaß. Ich bin hauptsächlich an FORTH + Physik interressiert. Dies schließt vor allem die Numerik und die Grafik ein. Jedoch interessieren mich auch andere Sprachen, weil ich glaube, daß FORTH in vieler Hinsicht (noch) nicht perfekt ist.

Stichworte

» IBM-volksFORTH,
» High-Level Interrupts

High-Level Interrupts im IBM-volksFORTH

Screen # 0

```
*****************************************************fnk 31mär89


  High-Level-Interrupts in volksFORTH


  Frank Stüss
  An der Turnhalle 6
  6369 Schöneck 2

  Tel.: 06187-5019


*****************************************************************
```

Screen # 1

```
\  Loadscreen Interrupts                                fnk 01apr89

  2 4 thru

  cr .( Interrupt-Behandlung geladen ) cr
```

Screen # 2

```
\ Interrupt-Handler                                     fnk 04apr89

  Code getint  ( n -- seg addr )   \ holt Interrupt-Adresse
          R push  D A mov  53 # a+ mov  $21 int
          E: A mov    R D mov   R pop  A push
          C: A mov   A E: mov   Next   end-code

  Code onint   ( n addr -- )  \ verbiegt interrupt n auf addr
         A pop   37 # a+ mov  \ im FORTH-Segment
         $21 int  D pop  Next    end-code


  : Intstart:  [ ASSEMBLER ]  A push R push C push D push
               U push I push   W push E: push D: push ;c: ;
```

Screen # 3

```
\  Interrupt-Return  SOFTRET                            fnk 31mär89
\ Alle Return-Worte sind der Abschluß für die Einleitung
\ mit  Intstart:
\ für Software-Interrupts:


  Code softret   D: pop E: pop W pop I pop U pop D pop C pop
                 R pop A pop iret
```

Screen # 4

```
\  Interrupt-Return  HARD1RET   HARD2RET                fnk 31mär89

\  Für Hardware-Interrupts:

  Code hard1ret  D: pop  E: pop W pop I pop U pop D pop C pop
                 R pop
                 $20 # A- mov  $20 #) byte out  \ EOI senden
                 A pop iret

  Code hard2ret  D: pop  E: pop W pop I pop U pop D pop C pop
                 R pop
                 $20 # A- mov  $A0 #) byte out   \ EOI
                 $20 #) byte out  \  für beide Controller
                 A pop iret end-code
```

Screen # 5

```
\                                                       fnk 31mär89

 Beispiel:

  $ff here  ( n addr )

  Intstart: ." Hallo, ich bin der Interrupt ! " bell
            softret ;
  onint

  Code testint  $ff int Next end-code

  ok
  testint <ret> Hallo, ich bin der Interrupt ! <PIIIEPS>
  ok
```

Screen # 6

```
\                                                       fnk 31mär89

  GETINT     liest die Interrupt-Adresse des Interrupts n
             aus der Int.-Tabelle.

  ONINT      verbiegt den Interrupt der Nummer n auf die
             Adresse addr im FORTH-Segment

  INTSTART:  ist ein Makro welches Interrupt-Worte einleitet
             Diese 'Worte' besitzen keinen Header. Die Adresse
             muß sich also mit 'here' gemerkt werden !
```

Screen # 7

```
\                                                       fnk 31mär89

  SOFTRET    ist das Return-Wort für Software-Interrupts.

  HARD1RET   ist das Return-Wort für Interrupts, welche der
             Interrupt-Controller 1 ausgelöst hat.
             Es wird ein EOI an den Controller geschickt.

  HARD2RET   ist das Return-Wort für Interrupts, welche der
             Interrupt-Controller 2 ausgelöst hat (nur AT).
             Das EOI wird an beide Controller gesendet.
```

Ein weiteres Beispiel wäre die Auslagerung des Editors oder des Debuggers in ein anderes Segment, so daß er bei Bedarf nachgeladen und PER INTERRUPT aufgerufen wird, was Platz sparen hilft (eigentlich sollte man ja in FORTH mit weniger als 20 KB auskommen, jedoch hatte ich da einige Probleme mit numerischen Anwendungen). Wer macht's ?

Schließlich und endlich ist da ja auch noch eine Domäne von FORTH, nämlich das Messen, Steuern und Regeln, was ohne Interrupts in vielen Fällen ziemlich brotlos ist. Wer hier zwar ein Hardware-As ist, jedoch mit Assembler auf Kriegsfuß steht, kann ja jetzt in FORTH. Es dürfte auf einem AT auch in zeitkritischen Situation öfter anwendbar sein. Wenn's halt gar nicht geht, muß halt Assembler her, aber das ist jetzt nicht das Thema.

Frank Stüss
An der Turnhalle 6
6369 Schöneck 2
Tel.: 06187-5019

FORTH-Bibliothek, Teil 4

Dies ist eine auszugsweise Übersicht der FORTH-Bibliothek der Münchner Gruppe. Bei Interesse an einem Artikel kann man sich an Christoph Krinninger wenden. Diesesmal werden wiederum eine Reihe von Anwendungen vorgestellt. (verw. Abkürzungen: DDToF = Dr. Dobb's Toolbook of FORTH, DD Journal = Dr. Dobb's Journal)

Einblicke in FORTH	Decompiler in FORTH-79	. Jürgen Schmidt	c't
Elements of a FORTH Database Design	Grundlegende Worte für eine Datenbank	Glen B. Haydon	DDToF
Entwicklung einer digitalen Oszilloskop-kamera	Eine umfangreiche Diplomarbeit in FORTH	Michael Stenzel	
Error Trapping and Local Variables	Der 'Mißbrauch' des Returnstack für Fehlerbearbeitung und Lokale Variablen	Klaus Schleisiek	DDToF
Error-Traps	Definition von Error-Routinen, Rücksprung zu einer definierten Stelle	Klaus Schleisiek	
Escaping FORTH	Bessere Lesbarkeit von Sequenzen, Selektionen und Iterationen in FORTH	Wil Baden	
Evolution of a Video Editor	Implementation eines Mini-Editors	Wendall C. Gates	DDToF
Extended Control Structures	IF-THEN-ELSE etc. mit allerlei Ausnahmebearbeitung	George W. Shaw II	DDToF
F83 Word Usage	Statistische Analyse des Dictionary	C.H. Ting	DDToF
Factoring in FORTH	Der massive Gebrauch von CREATE DOES> und andere Tips zur besseren Lesbarkeit	Michael Ham	DDToF
FORTH and the EMS	Einbindung der Expanded Memory Specification	Ray Duncan	DD Journal
FORTH decompiler	Decompiler in fig-FORTH	Ray Duncan	DDToF
FORTH in Rehabilitation Applications	Artikel über den Einsatz von FORTH in der Entwicklung von Geräten für Behinderte	David L. Jaffe	DDToF
FORTH in Space	Artikel über den Einsatz von FORTH bei biologischen Experimenten im Weltraum		euroFORML '88
FORTH Shifts Gears	The next FORTH generation has syntax that allows object-like multiple code fields	George W. Shaw	Computer Language
FORTH Turtle-Grafik für GDP EF9365	Verwendung des EF9365 für Turtle-Grafik	Rolf Schöne	65xx Micro Mag
FORTH Windows for the IBM-PC	Window-Paket in F83	Craig A. Lindley	DD Journal
FORTH-Disassembler	Universeller Decompiler für 6502/fig-FORTH	Dr. Helmut Mörtl	65xx Micro Mag
FORTH-Editor	Universeller Editor für Screens	Klaus Flesch	65xx Micro Mag
Fractal Landscapes	Fraktale Berge und Landschaften in FORTH	Phil Koopman	Jr.
GO in FORTH	Das berühmte Spiel...	C. H. Ting	DDToF
H-19 Screen Editor	Implementation für das Heath H-19 Terminal	Albert S. Woodhull	DDToF
Hacking FORTH	Bessere Lesbarkeit durch Flußdiagramme	Wil Baden	DDToF
Install a VBL Task with MACH1	Installing a FORTH VBL Task - another CRT saver	Jörg Langowski	The Complete MacTutor
Interfaces for a Mouse	Einbindung einer Maus für IBM-PC	Ray Duncan und Rick Wilton	DDToF
Keyboard Re-Mapper Utility	Keyboard configuration and reconfiguration (Macintosh)	Jörg Langowski	The Complete MacTutor
Leaping FORTH	Flußdiagramme zur besseren Darstellung von FORTH-Programmen	Wil Baden	DDToF
LIST: A Generator for Object Oriented, Cyclic Linked Lists,	Implementation eines objektorientierten Kernes	K.-Dietrich Neubert	
LOGO in FORTH	A Role for Stack Frames and Local Variables, Implementation von Logo in FORTH	Lance Collins	
Lokale Variablen	Lokale Variablen in fig-FORTH	A. Kochenburger	mc 01/88
Mengen in FORTH	Die Verwendung von Mengen (Sets) bietet effiziente Speicherausnutzung	Hans-Georg Lange	65xx Micro Mag
METHODS: Object oriented extensions redux	Implementation eines objektorientierten Kernes	Terry Rayburn	
Multiple Inheritance Object Systems	Beschreibung von ForthTalk, einer objekt-orientierten Erweiterung	Stephen D. Lindner	
Nondeterministic Control words in FORTH	Zufallsgesteuerter Aufruf von Worten	Louis L. Odette	DDToF
Object-oriented FORTH	Einleitender Artikel über objektorientierte Worte	Dick Pountain	BYTE
Principals of Text Editing	TextEdit Manager unter FORTH	Jörg Langowski	Best of MacTutor
Quicksort and Swords	Diskussion von Sortiermethoden in FORTH	Wil Baden	DDToF Vol. 2
Recovering Lost Files	Disk Editor unter NEON	Jörg Langowski	Best of MacTutor

Lokale FORTH-Gruppen, die sich regelmäßig treffen:

1000 Berlin	Claus Vogt, Tel.: 030/2168938. Treffen am letzten Donnerstag des Monats um 19.30 Uhr in der Technischen Universität Berlin, Mathematikgebäude, 6.Stock im Raum MA 621
2000 Hamburg 13	Karsten Roederer, Tel. 040/4104446, tagsüber 412 329 84, Treffen im Geomatikum Raum 1438 (14. Stock), Bundesstr. 55, am 28. Juni, 27. September, 25. Oktober jeweils um 19.30 Uhr
4130 Moers 1 Rhein-Ruhr	Friederich Prinz, näheres Tel: 02841/583 98 Jörg Plewe, Tel: 0208/423514, Treffen nach Absprache. Der nächste Termin kann bei Jörg Plewe erreichbar unter obiger Telefonnummer erfragt werden.
6100 Darmstadt	Andreas Soeder, Tel. 06257/2744. Treffen an der VHS an einem Mittwoch in der Mitte des Monats.
6800 Mannheim	Lokale Gruppe Rhein-Neckar, Thomas Prinz, Tel.: 06271/2830. Treffen jeden ersten Mittwoch im Monat im Vereinslokal des Segelflugvereins Mannheim e.V. Flugplatz, Mannheim-Neuostheim.
8000 München	Heinz Schnitter, Tel. 089/3103385 oder Christoph Krinninger 089/7259382. Treffen jeden 4. Mittwoch im Monat 19 Uhr 30 im Vereinsraum 2 im Bürgerhaus Unterschleißheim am Rathausplatz (S-Bahnhaltepunkt S1 Unterschleißheim).

FORTH-Fachgruppen:

8000 München	RTX 2000 Gruppe, Koordinator Max Diez, Treff- und Zeitpunkt wie oben bei der lokalen Münchner Gruppe.
6800 Mannheim	FIS (FORTH Integriertes System) - Datenbank, Textverarbeitung, Kalkulation, Postadresse: Dr. med. Elemer Teshmar, Danziger Baumgang 97, 6800 Mannheim 31

Es möchten in ihrer Region eine Gruppe gründen:

7000 Stuttgart 31	Wolf-Helge Neumann, Huttenstr. 27, Tel. 0711/882638.
8500 Nürnberg 20	Thomas G. Bauer, Fichtestr. 31, Tel. 0911/538321.
5000 Köln 60	Michael Heycke, Boltensternstr.
4830 Gütersloh 1	Ludwig Röver, Holzheide 145A
4900 Herford	Andreas Findewirth, Im Großen Vorwerk 48, Tel.: 05221/23504

Eine Fachgruppe will gründen:

7000 Stuttgart 80	Grafik/Arithmetik, Jörg Tomes, Anweilerweg 56, Tel. 0711/7802293.
8000 München 70	Btx u. FORTH, Christian Schwarz, Lindenschmitstr.30, 8000 München 70

Hier kann man um Rat fragen:

02103/556 09	Jörg Staben, Dienstag und Freitag, 20.00 - 22.00 Uhr
02845/28951	Karl Schroer

Ansprechpartner zu bestimmten Interessengebieten:

volksFORTH/ultraFORTH:	Klaus Kohl, Tel.: 08233/30524 Bernd Pennemann, Tel. 0228/640979 und Klaus Schleisiek-Kern, Tel. 040/2202539.
32-Bit Systeme:	Robert Jones, Tel. 02434/4579
Künstliche Intelligenz:	Ulrich Hoffmann, Tel. 0431/678850
NC4000 Novix Chip:	Klaus Schleisiek, Tel. 040/6449412
Realtime relationale Netze:	Wigand Gawenda, Tel. 040/446941
Gleitkomma-Arithmetik:	Andreas Döring, Tel. 02631/52786
32FORTH	Rainer Aumiller, Tel. 089/6708355
PostScript/FORTHscript	Christoph Krinninger, Tel: 089/725 93 82

FORTH-Gesellschaft e.V. - Postfach 1110 - D-8044 Unterschleißheim
Tel.089/3173784, FORTH-Mailbox Tel.: 089/7259625
Postgiroamt Hamburg, Kontonr.: 563211-208 BLZ 20010020

Ergänzungen, Änderungen bitte dem Büro der FORTH-Gesellschaft e.V. mitteilen.

FORTH-SYSTEME

Software-Vertrieb
Beratung und Schulung

ANGELIKA FLESCH

Software-Entwicklung
Hardware-Entwicklung

UR/FORTH

- Forth-83 Standard
- Für MS-DOS, OS/2, 80386, 68000 UNIX und XENIX
- Direkt gefädelte Code Implementationen mit dem obersten Stackwert im Register um größtmögliche Ausführungsgeschwindigkeit zu erreichen
- Segmentiertes Speichermodell mit Programm, Daten, Headers und Dictionary Hash Table jeweils in einem getrennten Segment
- Komplett gehashtes Dictionary führt zu extrem schneller Übersetzung
- Mächtige neue String Operatoren (Suche, Extraktion, Vergleich und Addition) sowie einen dynamischen String-, Speichermanager
- Kann mit Objektmodulen, die in Assembler oder anderen Hochsprachen erzeugt wurden, gelinkt werden
- Native Code Optimizer zur direkten Umsetzung in 80 x 86 Code im Lieferumfang

HARRIS RTX 2000

Informieren Sie sich über diesen Prozessor, der auch von uns unterstützt wird.

DSP APPLIKATIONEN

DSP Anwendungen mit dem AT & T DSP-32. Informieren Sie sich über unser Angebot.

FORTH MAIL BOX

Für alle FORTH-Interessierten hat unsere Firma eine Mailbox eröffnet. Sie ist unter der Nummer 0 76 67 556 zu erreichen und akzeptiert 300, 1200 und 2400 Baud, 8N1. Außer einer offenen Hauptkonferenz und einigen Fileareas enthält sie auch Supportkonferenzen für unsere komplette Produktlinie.

LMI FORTH-83 Metacompiler

Der LMI Forth Metacompiler wird mit komplettem Quellcode für ein ausführlich ausgetestetes, Hochgeschwindigkeits Forth 83 Kern ausgeliefert, wobei Sie die Auswahl aus folgenden Zielprozessoren haben:

8086/8088	8096/97
Z80	HD64180
8080/8085	8031/32/535
68000	6303
Z8	6502
1802	6802
6809	68HC11
65816/65802	RTX 2000

Sie erzeugen schnelle und kompakte Anwendungen, indem Sie Ihre Quellprogramme mit unserem Forth Nucleus zusammenstellen und ihn mit dem LMI Forth Metacompiler übersetzen.

Forth Programme, die mit einem LMI interaktiven Forth System z. B. PC/FORTH oder Z80 Forth geschrieben und getestet wurden, werden im Normalfall mit nur geringen Änderungen übersetzt.

Serieller ROM/RAM Simulator

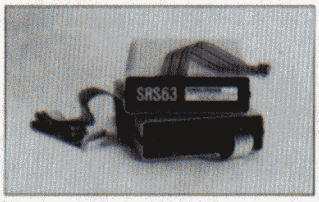
Entwickeln Sie romfähige Programme ?

Müssen Sie neu entwickelte Einplatinencomputer testen ?

Setzen Sie 2764, 27128, 27256, 27512 oder 4364, 43256 oder kompatible ROM/RAM-Bausteine ein ?

Wollen Sie diese Bausteine mit bis zu 38 400 Baud über die serielle Schnittstelle laden ?

Können Sie eine zusätzliche serielle Schnittstelle über den Speichersockel zum interaktiven Programmieren gebrauchen ?



Dann ist unser SRS63 die optimale Ergänzung Ihres Arbeitsplatzes.

Sie werden vom Preis-Leistungsverhältnis überrascht sein.

Unsere ROM-Compiler liefern direkt verwendbare Dateien, wir akzeptieren auch Intel-Hex oder Motorola-S-Formate.

Bitte fordern Sie unseren Produktkatalog und Preisliste an. FORTH-Gesellschaftsmitglieder erhalten bis zu 10 % Rabatt (artikelabhängig).

Kühnheimerstraße 21 · Postfach 1103 · D-7814 Breisach · Tel. (0 76 67) 5 51 · Fax 07667 555