VIERTE DIMENSION

8. Jahrgang Nr. 1 März 1992

HP48SX

CRC Test

Yerk

Prinz Forth

FORTH MAGAZIN

7,50 DM

Hier könnte Ihre Anzeige stehen:

Gültige Anzeigenpreise und -formate bei:

FORTH-Gesellschaft e.V., W-8044 Unterschleißheim, Postfach 1110

Tel. 089-317 37 84 oder DFÜ 089-871 45 48 secretary

Postgiroamt 2000 Hamburg, Kontonummer : 56 32 11-208
Bankleitzahl : 200 100 20

und

Editor und Redaktion:

Peter Dinies, Metzstraße 38, W-2300 Kiel 1, Tel. 04 31-1 32 39

FORTH-Magazin
Vierte Dimension

Inhalt

Editorial

Liebe Leser, die Vierte Dimension, unser Forthmagazin, besteht nun schon seit etlichen Jahren. Das Magazin soll weiter bestehen. Dazu braucht es vor allem Sie, die Leser und Autoren. Kommen Beiträge, kommt die Zeitung. Sie selbst entscheiden durch Ihre Arbeit über die Zukunft des Heftes. Es ist halt im wesentlichen eine Zeitschrift für die Veröffentlichung von Forthanwendungen geworden. Die Forthgesellschaft und der Editor sind lediglich Sponsor und Herausgeber der Zeitschrift.

Herausgeber zu sein ist einfach in Zeiten reger Aktivitäten der Forthler - und äußerst mühsam aber, wenn diese gerade andere Dinge zu tun haben, als ausgerechnet an das Forthmagazin zu denken. Glücklicherweise gab es in all den Jahren doch immer wieder genug Material zu veröffentlichen - obwohl sicher jeder Editor diese Situation erlebt hat: Redaktionsschluß war gestern, eingetroffene Beiträge = null; was nun?

Vier Generationen von Editoren hat die Gesellschaft inzwischen so nach und nach schon gehabt. Die ersten Hefte aus Hamburg und dann Wuppertal wurden noch ganz in ehrenamtlicher Arbeit und mit bescheidenen Mitteln erstellt. Layouten war wörtlich gemeint: Blätter auslegen, schnippeln und kleben. Schon maschineller gearbeitet wurde dann in München und jetzt in Kiel von Peter Dinies. Maschinenlesbare Texte wurden mit Ventura Publisher verarbeitet. Ein weiteres Novum seit Kiel: Es gab schon eine kleine Redaktion drumherum mit Ulrich Hoffmann, Michael Kalus, Rolf Kretschmar und Jens Storjohan.

Und nun ist erstmals in der Geschichte des Blattes die Situation eingetreten, daß sich gleich mehrere Teams um die Herausgabe der Zeitschrift bewerben zu einer Zeit, da diese Redaktion auseinandergeht. Verschiedenen persönlichen und berufliche Gründe gebieten es, die redaktionelle Arbeit abzugeben. Nun, wohin geht die Redaktion diesmal? Vom Norden ging es über den Westen in den Süden der Republik und dann zurück in den Norden. Nun nach Osten? Das stand bei Redaktionsschluß noch nicht fest. Lassen wir uns überraschen.

Die Mitgliederversammlung in Rostock Ende März ist gewesen, wenn dieses Heft erscheint. Mit dem nächsten Heft dürfte sich das neue Team dann bereits vorstellen. Neue Ideen, frische Kräfte und weiter verbesserte technische Voraussetzungen sollen geboten werden.

Damit das auch wahr wird, wünschen wir der neuen Redaktion viele gute Beiträge aus der Forthwelt und viel Glück und den Lesern auch in Zukunft viel Spaß bei der Lektüre.

Wir bedanken uns bei den Autoren, Lesern, Kritikern und der Forthgesellschaft und wollen uns mit diesem Heft verabschieden.

Tschüß, Eure Redaktion aus Kiel.

IMPRESSUM

FORTH-*Magazin "Vierte Dimension"*

Herausgeber:
FORTH-Gesellschaft e.V.

Editor, Satz und Layout:
Peter Dinies, Metzstraße 38, W-2300 Kiel 1, Tel. 0431/1 32 39

Beirat:
Ulrich Hoffmann, Jens Storjohann, Michael Kalus

Illustrationen
Rolf Kretzschmar

Kontaktadresse:
FORTH-Büro, Postfach 1110, W-8044 Unterschleißheim, Tel. 0 89/3 17 37 84 oder FORTH-Mailbox, München, Tel. 0 89/871 45 48 8N1 "Konferenz Vierte Dimension".

Quelltextservice:
Der Quelltext von Beiträgen, die mit dem Disketten-symbol gekennzeichnet sind, ist auf der Leserservice-Diskette zur jeweiligen Ausgabe oder in der FORTH-Mailbox zu finden.

Redaktionsschluß:
Erste Woche im mittleren Quartalsmonat
Erscheinungsweise vierteljährlich

Auflage:
Ca. 1.000

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

Preis:
Einzelheft DM 7,50, Abonnementpreis DM 40,-, bei Auslandsadresse DM 45,- inklusive Versandkosten

Open-Boot-Firmware

Author: Mitch Bradley, Chairman

Übersetzung: Ulrich Hoffmann

IEEE P1275 Open Boot
Working Group
Sun Microsystems MTV17-08
2550 Garcia Ave.
Mountain View, CA 94043
Phone: +1 (415) 336-3608
FAX: +1 (415) 962-0927

Open-Boot ist ein portables, CPU und BUS-unabhängiges Firmware-System, das auf der Programmiersprache Forth beruht. Open-Boot wird z.Zt. in über 400.000 Workstations mit dem SPARC-Prozessor erfolgreich eingesetzt und kann leicht auf nahezu jedes Allzweck-Computer-System angepaßt werden.

Die IEEE P1275 Open-Boot Arbeitsgruppe entwickelt gegenwärtig einen Vorschlag zur Standardisierung von Boot-Firmware basierend auf dem Open-Boot-System.

Open-Boot bietet die Möglichkeit, Hardware und Software automatisch zu konfigurieren, das Betriebssystem zu laden und Optionen zu verwalten. Zusätzlich bietet es ausführliche Unterstützung bei der Fehlersuche in Hardware, Software und in der Firmware.

Hersteller von Systemen auf Basis von Futurbus+, VME-D, Sbus und ComBus prüfen ggw. die Eignung, Open-Boot als Firmware-Architektur einzusetzen. Sun Microsystems benutzt Open-Boot in allen ihren gegenwärtigen Systemen. Force Computer hat die Benutzung von Open-Boot für ihre zukünftigen Produkte, auf 3 CPUs und mit 4 unterschiedlichen Bus-Systemen, angekündigt.

Die Standardisierung von Open-Boot wird auf dem Arbeitsmarkt Plätze für Forth-Programmierer schaffen.

Open-Boot Eigenschaften

- Maschinen eigenen Einsteckkarten mit Gerätetreibern
- Automatische Konfiguration mit Geräten, die sich selbst identifizieren
- Programmierbare Benutzerschnittstelle basierend auf Forth
- Erweiterbare, eindeutige und hierarchische Gerätenamen
- Werkzeuge zur Fehlerbeseitigung in Hardware, Software, Firmware und in Gerätetreibern
- Patch- und Erweiterungs-Skripts in nichtflüchtigem RAM
- Inspektion des Gerätebaums
- Objekt-orientierte Verwaltung der Konfiguration
- Text- und Font-Unterstützung für bit-mapped Grafik-Geräte
- TFTP/UDP/IP Netzwerk-Protokolle für das Booten
- Anforderung von virtuellem und physikalischem Speicher
- Schnittstelle zur Unterstützung des Betriebssystems
- Kommandozeilen-Editor mit Kommando-Vervollständigung und interaktiv manipulierbarer Kommando-Geschichte
- On-Line Hilfestellung
- Terminal-Emulation

Gerätebaum

- Hierarchische Repräsentation der Hardware
- Interne Knoten sind Busse oder Adreßumsetzer
- Blätter sind Geräte
- Geräte werden durch einen Namen und eine physikalische Adresse identifiziert
- Physikalische Adressen sind relativ zum "Elternadreßraum"

Beispiel:

/sbus/scsi@1,40000/disk@2,1

Geräte-Knoten

- Jeder Eintrag im Gerätebaum ist ein Geräte-Knoten, der entweder einem Gerät, einem Bus oder einem Software-Paket zugeordnet ist.

- Jeder Geräte-Knoten hat eine Liste von Methoden (Forth-Worte) und eine Liste von Eigenschaften. Jede solcher Listen ist ein Forth Vokabular.

- Geräte-Methoden sind Forth-Worte, die einen Treiber für das Gerät realisieren.

- Geräte-Eigenschaften sind Name/Wert-Paare, die das Gerät beschreiben.

- Der Geräte-Name wird als Name eines Forth Wortes abgelegt. Der Wert ist ein Byte-Array, das codierte Informationen, wie Strings, Zahlen oder Adressen enthält.

Beispiel:

- "model" - Die Modellnummer eines Gerätes

- "reg" - Liste der Register-Adressen

- "intr" - Interrupt-Ebenen, die das Gerät verwendet.

FCode

- Neue Treiber können einem Open-Boot-System hinzugefügt werden, z.B. indem eine neue Grafik-Karte auf den SBus gesteckt wird. Die Karte enthält ein ROM, welches das Gerät identifiziert. Zusätzlich enthält das ROM ein Forth-Programm, codiert in FCode.

- FCode ist byte-orientierter Forth-Quellcode. Jedem Standard-Forth-Wort ist eindeutige eine Zahl zugewiesen. Statt des Quellcodetextes sind nun diese Zahlen im FCode-ROM abgelegt. Einige Zahlen stehen dem Programm zur Verfügung, sodaß es selbst Forth-Worte erzeugen kann. Das FCode Codierung-Schema führt zu sehr kompakten Treibern.

- Um einen FCode Treiber zu benutzen, erzeugt das Open-Boot-System einen neuen Knoten im Gerätebaum und interpretiert das zugehörige FCode-Programm. Alle neuen Worte, die der FCode definiert, werden als Methoden dieses Geräte-Knotens ins RAM kompiliert.

- FCode-Treiber sind CPU-unabhängig, d.h. der gleiche Treiber kann zusammen mit Maschinen benutzt werden, die unterschiedliche CPU-Instruktionssätze haben.

Fehlersuche

- Open-Boot kann mit Hilfe der folgenden Werkzeuge Fehler in Hardware, Betriebssystem-Software und in sich selbst aufspüren:

- Symbolischer Disassembler

- Strukturierter Decompiler (Decompiliert Kontrollstrukturen ordnungsgemäß)

- Einzelschrittausführung und Breakpoints

- Forth Quelltext Debugger

- Interpretierte Kontrollstrukturen

- Da FCode Treiber wie normale Forth-Definitionen in den Speicher kompiliert werden, können die gewöhnlichen Forth-Werkzeuge auch ebenso für FCode eingesetzt werden. Die interaktive Forth-Umgebung ist besonders beim Debuggen von Hardware hilfreich.

Vorteile von Forth

Die Benutzung von Forth als Grundlage von Open-Boot hat viele Vorteile:

- leistungsfähige Benutzerschnittstelle

- gleiche Schnittstelle für alle Funktionen

- einfach erweiterbar, um neue Möglichkeiten zu unterstützen

- programmierbar

- effizienter Speicherplatzbedarf

- hochgradig portabel

- einfach zu debuggen, insbesondere im Feldeinsatz

- unterstützt maschinenunabhängige FCode-Treiber

Leserbriefe

Die zwei wichtigsten FORTH-Bücher:

Thesaurus-Leitfaden
G. Wersig
Deutsche Gesellschaft
für Dokumentation
Schriftenreihe Bd.8
K.G.Saur 1985

Indexing, the Art of
A guide to the indexing
of books and periodicals
G. Norman Knight, MA
G. Allen & Unwin Ltd. 1979

Ein Gedicht in Englisch aus dem obigen Werk:

"Without a key we search
and search in vain,
but a good index is
amonstrous gain."

Obwohl ich schon die Tomaten und Eier auf mich zufliegen sehe, betrete ich mutig die Bühne, rufe: "Wider die Mülltonne!!" und übertrage in's Deutsche:

"Wenn man nicht weiß, wie, sucht man sich tot; aber das WIE ist eine Heidenarbeit." (Applaus für den Übersetzer, bitte.)

(FORTH-)Programmierer schreiben gerne für die Mülltonnne. Hier ein Compilerchen, da ein Entwicklungssystemchen... Ist die Mülltonne voll, begibt man sich an die tägliche Neuerfindung des Rades. Währenddessen erstellen TurboPASCAL-Programmierer die ultra-scharfen Applikationen, ehrlich - ich hab' die starken Anwendungen selbst gesehen!

Das wissen wir aber schon alles; die Frage ist nur: Machen wir weiter so, bis sich die letzten drei FORTH-Programmierer allmonatlich zum Skat-Abend treffen oder tun wir was dagegen?

Gut - machen wir also was dagegen: Wie fangen wir an? Wir sortieren erst mal, was da ist und beurteilen dann, ob man es gebrauchen kann. Kann man es gebrauchen, schreiben wir es uns auf und überlegen, wie wir es finden, wenn wir es brauchen. Ja bitte?

Dann wird ein Ding SOLANGE OHNE ÄNDERUNG BENUTZT, wie es seinen Zweck erfüllt; erst wenn sich herausstellt, daß es nicht leistungsfähig genug ist, dann erst wird es geändert! Sollten Dinge fehlen, werden sie gemacht. Vorher wird aber erst beschrieben, was fehlt und was die Dinge machen sollen.

Tja - jetzt hab' ich den Faden verloren... egal, auf jeden Fall habe ich die einschlägige FORTH-Literatur (Beilstein, McCabe) nach diesem Thema durchsucht und bin fündig geworden – bei uns in der örtlichen öffentlichen Leihbücherei.

Die Mitarbeiter dort klassifizieren ihre Bücher deshalb, weil man sie in Regalen geordnet aufstellen will. Und damit die unseligen Leser die Bücher in den Regalen wiederfinden, sind alle Bücher NACH EINER ***SCHLAGWORT-NORM-DATEI*** *BESCHRIEBEN WORDEN.*

So einfach ist das; ich finde, wenn wir eh' alle soviel Zeit haben - siehe oben - dann sollten wir auch mal sowas machen.

Kleiner Schwenk: In der Zeitschrift TOOLS beginnt gerade eine neue Serie: WIR BAUEN EIN EXPERTENSYSTEM ZUR PROGRAMMIERSPRACHE. Damit soll man fragen können:"Wie mach' ich denn 'nen Interrupt (nein, nicht was Sie denken!)?" Das paßt wohl auch so'n bißchen in dieses Thema; meinen Sie, wir packen sowas auch?

In der Diskussion um die Klassifizierung und Katalogisierung von Programmen und Programm-Modulen habe ich das Argument gehört: "Lieber schreibe ich das schnell neu, bevor ich 'ne Viertelstunde suche."

Die Aussage ist natürlich FALSCH!, denn sie geht davon aus, das die auf die schnelle geschriebene Funktion sofort fehlerfrei und nicht dokumentiert wird. Dadurch bleibt auch diese Funktion wieder nicht nutzbar, sondern wird immer wieder und wieder neu erstellt, wann immer das Problem ernaut auftaucht.

Betrachten wir nun den Fall, daß diese "auf-die-Schnelle-gestrickte"-Funktion hier und da noch Tipp- und Denkfehler enthält, so kommt sehr rasch der Punkt, wo sich auch eine halbstündige Suche nach einem fehlerfreien, getesteten Modul als rentabel erweist!

Zugleich wird eine solche Modul-Datenbank das veraltete Wissen über FORTH sichten helfen; von Lösungen, die mehrfach enthalten sind, kann die leistungsfähigste aufgenommen und damit der Stand der Dinge festgestellt werden.

Für wen nun die ganze Arbeit?

Einen solchen Zugriff auf FORTH werden in erster Linie Amateure zu schätzen wissen, denn kein Profi wird - in der ersten Zeit - die Ergebnisse seiner Arbeit von Modulen fremder Leute abhängig machen. Ein Vorbehalt, den Anwender von C- oder PASCAL-ToolBoxen wohl nicht haben, sonst gäbe es den Markterfolg von diversen Zusatzprodukten für Standardcompiler nicht.

Dagegen wird nur die Verfügbarkeit einer solchen Modul-Datenbank dafür sorgen, daß Anwender daran GLAUBEN!, ihre Aufgabenstellung mit FORTH lösen zu können - ein Glaube, der zusehends in's Wanken gerät.

Wo gerade das Wort VERFÜGBARKEIT gefallen ist:
Verfügbar heißt: Es ist da, ich weiß davon und ich kann es mir kaufen. So gilt der SuperDuper-FORTH-Compiler irgendwo in einer einsamen Berghütte - obwohl es vielleicht der beste aller Compiler ist - als NICHT verfügbar; ein Compiler, dessen Basisversion 1000DM kostet, wenn die Konkurrenz die doppelte Leistung für 250

Leserbriefe und Nachrichten

Mark anbietet, gilt ebenfalls als nicht verfügbar.

Was heißt das? Es muß leistungsfähige FORTH-Produkte zu marktgerechten Preisen geben. Man könnte den Satz auch herumdrehen und marktfähige Produkte zu leistungsgerechten Preisen fordern; hieran läßt sich die nächste Aussage fest machen: Der Preis eines Produktes ist relativ - für das Ingenieur-Büro sind 1000DM für einen Compiler im Vergleich zum Auftragsvolumen von einer viertel Million schlicht geschenkt; für einen Hobbyisten sind 1.000 DM für sein 200 DM Budget einfach untragbar.

Allerdings muß ich auch sagen: Hier arbeitet der falsche Mann am richtigen Thema. Dieses (erste und letzte?) Projekt der FORTH-Gesellschaft muß von Leuten begleitet werden, die die Ausbildung, das Gewußt Wie und Lust haben.

Jawohl - immer noch FORTH!

Euer
Jörg Staben

Im Nachrichtenbrett der Forthgesellschaft FORTHEV/FORUM, das bundesweit Nachrichten verteilt, schrieb Johannes Teich folgende Zusammenfassung über eine Diskussion im internationalen Forth-Netz (comp.lang.forth):

Im Brett /FORTHEV/NEWS (Newsgroup comp.lang.forth) gab es eine Diskussion über den erforderlichen Forth-Minimal-Wortschatz, die ich ganz interessant fand. Milan Merhar schlug vor:

Siehe rechts

```
> Stack ops:
>  DUP ( create a stack element)
>  DROP ( destroy a stack
        element)
>  SWAP ( move stack element)
>  >R and R> (stack exchange)
> Arithmetic/logic:
>  LITERAL ( constants, etc.)
>  NAND ( sounds silly,
   but you can
   synthesize anything
    else out of it!)
> Address space access:
>  ! and @ ( or C! and C@,
     if you wish)
>  P! and P@ ( for i/o space
    port access,
    your CPU has such a thing...)
> Dictionary extension:
>  CREATE
> This is a _VERY_ sparse list!

  Uwe Kloss antwortete ihm:
> And to me it seems to be
  too sparse!
>
> If you consider the stacks
   as entities your
  stack ops are ok!
> But then you can NOT a
  ccess stack pointers!
> If you prefer to have
  the stack pointers as part
  of the virtual
> machine implemented you
  should write:
>
>    SP!
>    SP@
>    RP!
>    RP@
>
> And write the stack ops
  as colon definitions.
>
> This means you assume (!)
 that your stacks are
 memory mapped!
 ...
> But you should definitely
       add EXECUTE to that list.
> How do you implement
  EXECUTE for primitives
  as a colon definition?
>
> And a NAND operation
   is not enough. If you want
   to do arithmetics you
  > need shift operations
   as well!
> Left AND right!
>
> On the other hand you
  can easily replace LITERAL
  by a colon definition.
 >
```

Der HP48SX - Ein Taschenrechner, Forth-ähnlich programmierbar

Ulrich Hoffmann

Das ist er nun, der HP48SX, der Taschenrechner, der die Nachfolge des legenderen HP-41c antreten soll. Für Forth-Programmierer ist dieser kleine Zwerg mit der riesigen Funktionalität ganz besonders interessant, da er sich Forth-ähnlich programmieren läßt.

Der HP48SX hat eine mehrzeilige alphanumerische Flüssigkristall-Anzeige, in der im Normalbetrieb der typische HP-Stack dargestellt ist. Zusätzlich dient die Anzeige aber auch zur Darstellung von Formeln und zur graphischen Repräsentation von Funktionsverläufen. Eine spezielle Rolle nimmt die unterste Zeile der Anzeige ein. Sie stellt, verbunden mit den 6 Tasten der obersten Tastenreihe, anwendungsspezifische Menüs dar. Je nach eingestelltem Modus erscheinen dort Funktionen, die durch Druck auf die entsprechende Taste ausgelöst werden können. Alle Funktionen, die nicht direkt auf einer Taste liegen, können entweder über diese Menüs erreicht werden, oder direkt alphanumerisch eingegeben werden. Um bei der Vielzahl der Funktionen nicht den Überblick zu verlieren, haben die Entwickler des HP48SX eine Indexfunktion integriert, in der die einzelnen Funktionen nach Funktionalität sortiert aufgesucht werden können. So hat man die Möglichkeit eine gute übersicht über die existenten Befehle zu bekommen.

Zunächst zu den nichtprogrammierbaren Funktionen. Schon dort bietet der HP48SX all das, was das Herz eines "richtigen Rechners" höher schlagen läßt. Neben den Grundrechenarten, logarithmischen und trigonometrischen Funktionen, neben den hyperbolischen und Standard-Statistik-Funktionen bereiten dem HP48SX auch Rechnungen mit komplexen Zahlen, Vektoren und Matrizen keine Probleme. Alle Datenobjekte, dazu gehören auch Formeln und Programm-Definitionen (s.u.), mit denen der HP48SX arbeitet, lassen sich als ein Element auf den Stack legen, oder in benannten Variablen sichern.

Damit ähnelt das Daten-Konzept des HP48SX dem von Systemen wie ASYST oder POSTSCRIPT.

Zusätzlich zum reinen numerischen Rechnen bietet der HP48SX die Möglichkeit symbolische Ausdrücke zu verarbeiten. Einfachstes Beispiel ist die reelle Zahl PI, etwa als Ergebnis von ("1 ARCTAN 4 *"), die nicht angenähert (numerisch) sondern genau (symbolisch) dargestellt wird, und nur auf Wunsch in einen Zahlenwert überführt wird. Zu den Manipulationsmöglichkeiten für Formeln gehören alle klassischen Termumformungen, wie Ausmultiplizieren, Ausklammern, Zusammenfassen, usw., aber auch symbolisches Differenzieren und Integrieren. Der übergang vom numerischen Rechnen zur Symbolmanipulation ist fließend, da Ausdrücke auch gemischt numerische und symbolische Teilausdrücke enthalten dürfen.

Obwohl ein Forth-Anhänger keine Probleme damit haben wird, sich an die HP-spezifische Postfix-Eingabe zu gewöhnen, steht neben einer Infix-Eingabe für Formeln auch ein Eingabe-Editor zur Verfügung, der die entstehenden Formeln in klassischer mathematischer Notation auf dem Display graphisch darstellt. Für die Eingabe von Rechnungen ergeben sich damit insgesamt drei verschiedene Formen:

1) Postfix-Notation

```
z.B.     3  6  'A  / +
```

2) Infix-Notation ohne Eingabe-Editor

```
z.B.     '3+(6/A)'
```

3) Infix-Eingabe mit Eingabe-Editor

```
z.B.          6
          3+  -
              A
```

Mit Hilfe von Formeln lassen sich Funktionen definieren, die in einem speziellen Modus graphisch dargestellt werden können. Die Koordinaten des zu zeichnenden Ausschnitts können entweder automatisch bestimmt, oder in einem ausgefeilten interaktiven Prozess manipuliert werden. Mit Hilfe eines Graphik-Cursors, können markante Stellen des Graphs untersucht werden. So lassen sich beispielsweise Nullstellen oder Maxima einer

Funktion ganz einfach durch Draufzeigen (numerisch) bestimmen.

Ein weiterer Modus des HP48SX gestattet es, bei Angabe von beliebigen n-1 Variablen einer Formel mit n Variablen, die n-te Unbekannte zu bestimmen. Ein ausdrückliches Auflösen der Formel nach der gesuchten Unbekannten ist dazu nicht notwendig. Während der Eingabe der Variablen ist die Menüzeile mit den Variablennamen belegt, sodaß sich die Eingabe auf das Eintasten des Wertes und anschliessendes Drücken der zugehörigen Menütaste beschränkt. Wählt man eine Menütaste über einen speziellen Tasten-Präfix, so signalisiert dies das gewünschte Auffinden des Wertes für diese Variable, der dann in der Anzeige erscheint.

All die bisher vorgestellten Fähigkeiten des HP48SX machen ihn schon zu einem leistungsfähigen Werkzeug im Taschenformat, das sich im Punkte Vielseitigkeit durchaus mit Paketen wie Eureka, Derive, o.ä. messen kann.

Beim Stöbern im Funktionsindex stößt man unter der Rubrik Stackmanipulation auf Funktionen wie SWAP", "DROP" oder "DUP", die offensichtlich ihren Ursprung in Forth haben. Andere Funktionen, wie etwa "DUPN", haben die gleiche Funktionalität wie andersnamige Forth-Worte ("PICK"). Bei genauerer Betrachtung stellt sich heraus, daß die Programmiersprache, mit der der HP48SX programmiert wird, sich zwar in einigen Punkten an Forth anlehnt, aber nicht alle Konzepte verwirklicht, die in Forth zur Verfügung stehen.

Programm-Funktionen werden von "<<" und ">>" umfaßt. Sie werden benannt, indem sie in einer Variablen abgelegt werden.

Beispiel:

```
<< dup * >> 'QUADRAT STO
```

definiert die Funktion Quadrat. Im Variablen-Menü kann die Funktion direkt durch Druck auf die entsprechende Taste angewendet werden, ansonsten muß die Funktion EVAL zur Hilfe genommen werden.

Funktionsdefinitionen dürfen lokale Variablen enthalten, die ihre Anfangswerte aus dem Stack beziehen.

Beispiel:
Mittelwert zweier Zahlen

```
<  -> A B  < A B * 2 / > >
'MITTELWERT STO
```

Definierte Funktionen können wie in Forth einfach durch Nennung ihres Funktionsnamens angewendet werden:

```
<< dup QUADRAT * >> 'KUBIK STO
```

Als Kontrollstrukturen stehen die normalen Forth-Kontrollstrukturen zur Verfügung, nur daß sie vollkommen andere Syntax bekommen haben: (Vergleiche Übersicht)

Zusätzlich hat der HP48SX noch "funktionale" Kontrollstrukturen, wie etwa IFT bei denen sich zwei Funktionen und eine Bedingung auf dem Stack befinden, und je nach Bedingung die eine oder die andere Funktion angewendet wird.

Der wesentliche Unterschied des HP48SX zu einem traditionellen Forth-System besteht darin, daß der HP48SX kein Dictionary im Forth-Sinn zur Verfügung stellt. Die zugehörigen Manipulationsmöglichkeiten etwa durch "Defining Words" oder Vokabulare sind dann natürlich auch nicht notwendig. Die Folge davon ist, daß es keine "immediate" Worte und damit auch keine "Compiling Words" gibt, Konzepte, die u.a. zur besonderen Stärke von Forth gehören.

Die mitgelieferte Dokumentation besteht aus einem Handbuch in zwei Bänden und einer Kurzreferenz. Im ersten Band des Handbuchs wird die Benutzung des HP48SX beschrieben, ohne auf die Programmierung einzugehen. Der zweite Band widmet sich dann ausführlich der Programmierung. Die Kurzrefenz stellt die einfachsten Benutzungsregeln und eine Beschreibung aller HP48SX-Funktio-

Forth	HP-48
`<cond> IF <true-action> THEN`	`IF <cond> THEN <true-action> END`
`<cond> IF <true-action> ELSE <false-action> THEN`	`IF <cond> THEN <true-action> ELSE <false-action> END`
`<to> <from> DO <body> LOOP`	`<from> <to> FOR <body> NEXT`
`<to> <from> DO <body> +LOOP`	`<from> <to> FOR <body> STEP`
`<to> <form> DO ... I ... LOOP`	`<from> <to> START <var> ... <var> ... NEXT`
`BEGIN <body> <cond> UNTIL`	`DO <body> UNTIL <cond> END`
`BEGIN <cond> WHILE <body> REPEAT`	`WHILE <cond> REPEAT <body> END`
`CASE <expr1> OF <body1> ENDOF <expr2> OF <body2> ENDOF ... ENDCASE`	`CASE <expr1> THEN <body1> END <expr2> THEN <body2> END ... END`

nen zusammen, und ist so klein, daß sie bequem in der Tragetasche Platz findet.

Die gesamte Dokumentation des HP48SX ist in deutsch. Sie ist kompetent und gut verständlich übersetzt worden, wie es von Hewlett Packard zu erwarten ist. Für die HP48SX-Programmiersprache existiert ein eigenes Handbuch, das separat bestellt werden muß.

Von HP angebotenes Zubehör umfaßt eine serielle Schnittstelle nebst Treiberssoftware für IBM-PCs, einen infrarot-verbundenen Drucker und Speichererweiterungen.

Insgesamt ist der HP48SX der leistungsfähigste Taschenrechner, den ich bis dato zu Gesicht bekommen habe. Den Einfluß den Forth auf seine Entwicklung genommen hat, kann er nicht leugnen. Eine tragbare Forth-Maschine ist er aber dann doch nicht, dazu ist seine Programmiersprache doch zu weit von Forth entfernt.

Denjenigen, die viel interaktiv Rechnen müssen und die gerne Forth-ähnlich programmieren, sei er dennoch wärmstens empfohlen, somal er im Preis vergleichbar mit entsprechenden PC-Paketen liegt.

Ich möchte mich herzlich bei Firma Kiessling/Hamburg bedanken, die mir freundlicherweise ein HP48SX Testgerät zur Verfügung gestellt hat.

Fehlererkennung mit dem Cyclic Redundancy Code (CRC)

Test von Speicherinhalten in batteriegepufferten RAM oder EEPROMs

Claus Kühnel

In Mikrocontrollern üblich, so auch bei der Super8-Platine vorgesehen, ist eine Batteriepufferung des RAM. Die Lebensdauer jeder Batterie und auch eines ordnungsgemäß geladenen Akkumulators ist aber irgendwie endlich. Bei der Super8-Platine wird ein Datenerhalt im RAM bei Batteriepufferung von 1 bis 2 Monaten angegeben [1]. Als Maß für die Unversehrtheit der Daten kann der Cyclic Redundancy Code (CRC) verwendet werden, dessen besondere Bedeutung bei der Kommunikation im weitesten Sinne zu suchen ist. Zu den Hintergründen der Fehlererkennung muß auf die nicht einfach erschließbare Literatur [2][3] verwiesen werden. Hier sollen das Verfahren und das resultierende Programm im Vordergrund stehen.

Einige Vorbemerkungen

Um das ganze Verfahren der Bildung des CRC vereinfacht darzustellen, soll vorerst eine ganz normale Division betrachtet werden. Ein Dividend N wird durch den Divisor D dividiert, das Ergebnis ist ein Quotient Q und ggf. noch ein Divisionsrest R. Das ist allen bekannt und der folgende Zusammenhang kann notiert werden:

```
N = Q * D + R
```

Wird der Divisionsrest R vom Dividend N subtrahiert, dann muß sich eine restfreie Division möglich sein.

CRC in der Kommunikation

Bei der Datenübertragung nutzt man nun dieses Prinzip, wobei die Operanden nicht mehr durch einfache Zahlen sondern durch Polynome gebildet werden. Die zu übertragenden Daten werden, wie auch an anderen Stellen üblich, als Koeffizienten von Polynomen zur Basis 2 aufgefasst.

Nach der Theorie der Fehlererkennung existieren verschiedene Polynome, die als Divisor D verwendet werden können. Die zu übertragenden Daten werden als Dividend N betrachtet und es kann die Division folgen. Das Ergebnis sind wiederum Quotient Q und Divisionsrest R. Dieser Rest wird zusätzlich zu den Daten gesendet.

Auf der Empfängerseite werden die gesendeten Daten durch den bekannten Divisor D geteilt. Erhält man jetzt einen von Null verschiedenen Divisionsrest, dann liegt ein Fehler bei der Datenübertragung vor.

Test auf Unversehrtheit von gespeicherten Daten

Der Test auf Unversehrtheit von Daten in RAM oder EEPROM kann nun auf dieser Basis erfolgen. Die abgespeicherte Datenmenge wird nun byte- oder wortweise dieser Prozedur unterzogen. Von Interesse ist am Ende nur noch der verbleibende Divisionsrest R, der zur Kennzeichnung der abgelegten Daten dient. Ändert man auch nur ein Bit im untersuchten Speicherbereich, dann ändert sich auch der betreffende Divisionsrest R. Dadurch kann dieser Rest zum Zeitpunkt der Abspeicherung ermittelt auch weiterhin für die Unversehrtheit der abgelegten Daten herangezogen werden. Es sind dann nur der Test zu wiederholen und der neue Divisionsrest mit dem ursprünglichen zu vergleichen. Sind diese beiden Divisionsreste gleich, dann sind die Daten unversehrt.

Programm: s. S. 13

In Screen Scr # 1 sind die Worte cells und >< definiert. Cells ist im ANSI- Standard enthalten und wurde aus Kompatibilitätsgründen hier mit definiert. Das Vertauschen von Hi- und Lo-Byte besorgt das Wort ><. Bei Codedefinitionen mit dem Super8-FORTH ist zu beachten, daß der Top-of-Stack (TOS) im Register AX liegt. Die ansonsten erforderlichen PUSHs und POPs werden

so reduziert, was sich für die Laufzeiteffizienz positiv auswirkt.

In Screen Scr # 2 wird die Tabelle crc_table mit einer Länge von 256 Worten im RAM definiert. Anstelle des üblichen allot steht hier vallot , da bei ROM-fähigen Systemen eine genaue Unterscheidung zwischen ROM- und RAM-Speicherbereich getroffen werden muß. Desweiteren erfolgt die Definition des verwendeten CRC-Polynoms crc_polynomial als Konstante. Es sind verschiedene Polynome angegeben, die hier nicht benötigten sind auskommentiert. Schließlich folgt noch das Wort accumulate , welches byteweise die CRC-Berechnung durchführt.

Screen Scr # 3 beinhaltet die Definition des Wortes setup. Mit Hilfe dieses Wortes wird während der Compilierung die Tabelle crc_table mit den 256 möglichen CRC-Einträgen versehen. Der Vorteil dieses Verfahrens liegt in der schnelleren Berechnung mittels Tabelle, der Kaufpreis liegt bei einem Speicherbereich von 512 Byte für diese Tabelle. Ist die Tabelle initialisiert, dann muß das Wort accumulate dem auch Rechnung tragen. Nachdem alle nichtmehr benötigten Worte vergessen werden dürfen, erfolgt auch die Redefinition von accumulate.

In Screen Scr # 4 ist schließlich noch das Testwort test_crc definiert. Dieses Wort dient nur dazu, die Funktionsfähigkeit des Programmes zu verifizieren und damit eventuelle Tippfehler aufzuspüren.

Literatur:

[1] Singleboardcomputer mit ZILOG Super8 und ROM-Forth. Dokumentation zur Super8-Platine 23. 9.91

[2] Völz, H.:
Fehlerkorrektur - Möglichkeiten und Grenzen. J. Inf. Rec. Mater. 14 (1986) 4, S.289-296

[3] Peterson, W. W.:
Prüfbare und korrigierbare Codes. Oldenbourg-Verlag München, Wien 1967

Programm

Das resultierende FORTH-Programm für den Super8 ist in den folgenden fünf Screens gelistet. Die Anpassungen an andere Implementierungen sind minimal.

```
\ CRC Polynomials Made Plain                          ( 01.12.91/ck )

\ Der folgenden Quellcode ist dem Beitrag mit o.a. Titel
\ von Wil Baden zur FORML 1989 entnommen und an das Super8-
\ FORTH angepasst worden.


\ Zwei CRC-Polynome haben sich international durchgesetzt:
\ CRC-16 ist in der IBM-Welt verbreitet
\ CRC-CCITT ist spezifiziert durch ISO und CCITT

\ Claus Kühnel, Schlyffistr. 14, CH-8806 Bäch

\ cells ><
                                        ( 01.12.91/ck )
decimal
: cells   ( n -- 2*n )              \ for compatibility
: 2*
;
code ><   ( 16b1 -- 16b2 )          \ exchange Hi-Byte and Lo-Byte
       bl , al ld,
       al , ah ld,
       ah , bl ld,
       next,
end-code
-->


\ crc_table crc_polynomial accumulate              ( 02.12.91/ck )

variable crc_table 255 cells vallot

\ hex a001 constant crc_polynomial decimal ( ModBus )
\ hex 8005 constant crc_polynomial decimal ( CRC-16 )
 hex 1021 constant crc_polynomial decimal ( CRC-CCITT )

: accumulate   ( n c -- n )           \ accumulate CRC Remainder
       >< xor
       8 0 do
              dup 0< if 2* crc_polynomial xor else 2* then
       loop
;
-->


\ setup accumulate                                 ( 02.12.91/ck )

: setup   ( -- )
       256 0 do 0 i accumulate i xor i cells crc_table + ! loop
;
setup                                  \ creating the crc-table
forget crc_polynomial                  \ and forget the rest

: accumulate   ( n c -- n )           \ table driven accumulate
       >< xor
       >< dup 255 and cells crc_table + @ xor
;
-->


\ Test                                             ( 02.12.91/ck )
hex

\ Die CRC-"Summe" des Zeichens CR ( 0D ) lautet D1AD
: test_crc   ( -- )
       0 d accumulate u.
;
```

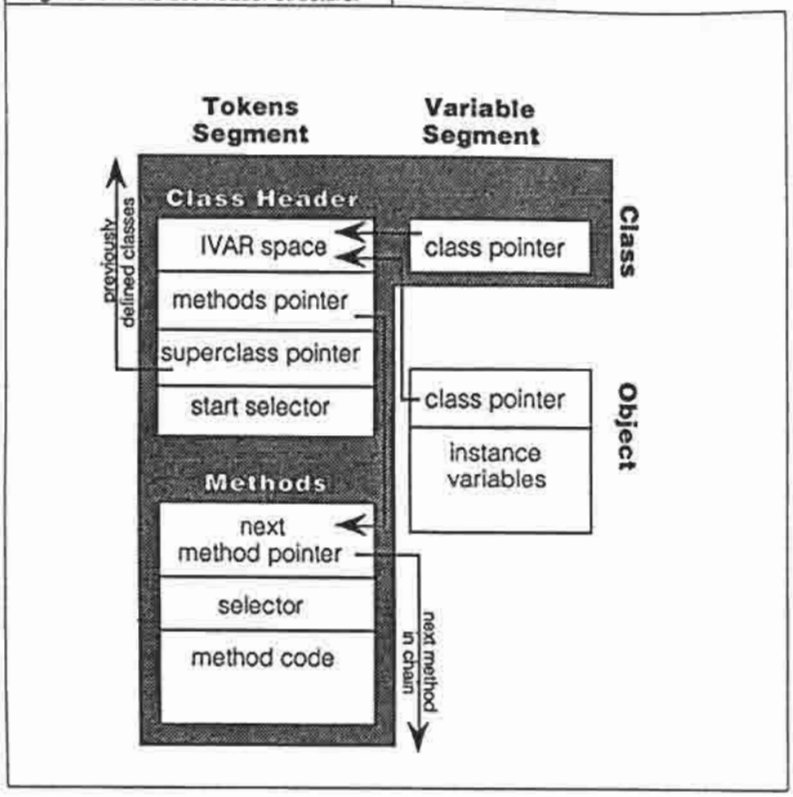
Yerk kommt zum PC

Rick Grehan

Peterborough, New Hampshire

(Übersetzung: Michael Kalus)

Figure One. Class header structure.



Listing Three. Building a headerless object.

```
:class polygon <super object
 2 ivar ^vertex_list

\ Allocate space for vertex list
:m init:  ( n -- )
     2*                \ 2 coordinates per vertex entry
     here dup ^vertex_list !  ['] ldarray >body @
     instantiate
;m

init: <<init-method

:m ->vertex:   ( y x i -- )
     2*             \ Index * 2 for x and y coordinates
     tuck           \ Get a second copy of the index
     to: ( ^vertex_list @ )   \ Store x
     1+             \ Advance index to next slot
     to: ( ^vertex_list @ )   \ Store y
;m

;class
```

Vor längerer Zeit schon entdeckte ich Neon, ein objektorientiertes Programmiersystem auf der Basis von Forth für den Macintosh. Aus irgendwelchen Gründen wurde Neon nicht weiter entwickelt. Nun tauchte diese Sprache als "Yerk" kürzlich in der public domain wieder auf - Bob Loewenstein vom Yerks Observatorium hatte daran gearbeitet, daher der Name.

Die objektorientierten Fähigkeiten von Yerk haben mich immer wieder beeindruckt. Und obschon ich häufig nur am Macintosh arbeite, habe ich doch auch genug in MS-DOS zu tun und wünschte mir daher soetwas wie Yerk auf dem PC. Darum schrieb ich PCYerk, ein annähernd kompletter Nachbau der Yerksyntax für den PC. "Annähernd" komplett, weil PCYerk auf einige der Eigenschaften von Yerk verzichten muß - natürlich auch auf all die Mac Toolbox Routinen.

PCYerk ist in Upper Deck Forth geschrieben, ein 16-bit multisegmentiertes Forth unter MS-DOS. Wer einigermaßen erfahren in F-PC ist, wird es leicht portieren können. Ich benutze PCYerk hier, um einige objektorientierte Konzepte vorzuführen.

Objekte und Klassen

Ein *Objekt* ist für dein Programm eine unteilbare Einheit aus Code und Daten. Der Code darin besteht aus einer Sammlung von Routinen für den Umgang mit solchen Daten, anderen Objekten, I/O

Treibern und sonstigen Systemteilen.

Eine *Klasse* ist so etwas wie eine Schablone für ein Objekt. Alle so erzeugten Objekte arbeiten entsprechend der Klassendefinition, bestehen im Grunde aber nur aus zwei Teilen: Den *Methoden* (das was die Objekte tun können) und den *Instanzvariablen* (lokale Daten der Objekte).

Wenn du ein Objekt angelegt hast, sagen wir "das Objekt wurde *erzeugt (instanziiert):* Aus der Klassenschablone wurde damit etwas reales (so real wie ein Stück Code halt werden kann). Sobald du ein Objekt erzeugt hast, kannst du Botschaften dorthin schicken damit es etwas tut. Formal gesehen besteht ein Objekt aus einem *Selektor* und einigen zugehörigen Daten. Der Selektor ist eine ID-Nummer und das Objekt entscheidet damit, welche Methode ausgeführt werden muß. In PCYerk gibt es keinen besonderen Unterschied zwischen der Botschaft und dem Selektor. Die Botschaft ist der Selektor und wird wie alle Daten auf dem Parameterstack übergeben.

In PCYerk definiert man Klassen mit dem defining word :CLASS. Listing 1 zeigt den Quellcode für die einfache Klasse INTEGER. Beachte, daß die Klassendefinition von dem Pärchen :CLASS und ;CLASS eingefasst ist (ich erkläre die anderen Teile der Definition später). Diese beiden Worte dienen zur Abgrenzung der Definition.

Instanzvariablen

Wir wissen schon, Instanzvariablen legen den lokalen Speicher für das Objekt fest. Wenn du nochmal auf Listing 1 schaust, siehst du, daß Objekte der Klasse Integer nur eine einzige Instanzvariable verwenden: LOCALDATA.

Das Wort IVAR erwartet den Platzbedarf für die Instanzvariable als Anzahl Bytes oben auf dem Stack - den Namen der Variable hingegen wird IVAR sich aus dem input stream holen. Jedesmal wenn ein Objekt des Typs Integer erzeugt wird, weiß jetzt das Sytem, daß zwei Bytes Platz für eine Variable angelegt werden müssen.

Listing Four. PCYerk source code.

```
\ PCYerk
\ Object-oriented extensions to Forth
\ a la Yerk (once, NEON)
\ Written for Upper Deck Forth, version 2.0
\ R. E. Grehan

\ Sorry, I can't stand "then"
: endif [compile] then ; immediate

\ *************
\ ** STORAGE **
\ *************
34 $variable tstring            \ Used in parsing names

\ Method stack
20 constant METH_STACK_SIZE
create mstack
      METH_STACK_SIZE allot

\ Objects stack
20 constant OBJ_STACK_SIZE
create ostack
      OBJ_STACK_SIZE allot

\ Instance variable names segment
variable ivar_seg               \ Segment
20 constant IVAR_SEG_SIZE       \ Segment size in paragraphs
variable ivar_next              \ Offset to next free loc.

\ Methods names segment
variable methname_seg           \ Segment
100 constant METHNAME_SEG_SIZE  \ Segment size in paragraphs
variable methname_next          \ Offset to next free slot
variable curr_meth#             \ Current method #
variable my_meth#               \ Method # about to be defined

\ Class definitions
variable    curr_class_off      \ Offset to current class
variable    curr_meth_tail      \ Current method tail

\ ****************************
\ ** METHOD AND OBJECT STACKS **
\ ****************************
\ NOTE: Neither stack do any bounds checking (for speed's
\ sake).  If bounds checking is added, the stack manipulation
\ words should be written in machine language.

\ Initialize the method stack...stores selector ids
: mstack-init     ( -- )
  mstack dup !
;

\ Push top word onto method stack
: mpush           ( n -- )
  mstack @ 2+ !                 \ Save item
  2 mstack +!                   \ Increment
;

\ Copy top of mstack to dstack
: mstack->dstack  ( -- n )
  mstack @ @
;
```

(Listing continues.)

(PCYerk Listing Four, continued.)

```
\ Pop top word from method stack
: mpop            ( -- n )
  mstack->dstack                    \ Fetch
  -2 mstack +!                      \ Decrement
;

\ Initialize the object stack
: ostack-init     ( -- )
  ostack dup !
;

\ Push top word onto object stack
: opush           ( n -- )
  ostack @ 2+ !                     \ Save item
  2 ostack +!                       \ Increment
;

\ Copy top of ostack to dstack
: ostack->dstack  ( -- n )
  ostack @ @
;

\ Pop top word from object stack
: opop            ( -- n )
  ostack->dstack                    \ Fetch
  -2 ostack +!                      \ Decrement
;

\ Dup top of object stack
: odup            ( -- )
  ostack->dstack                    \ Fetch top
  opush                             \ and push
;

\ Drop top of object stack
: odrop           ( -- )
  opop drop
;

\ Clear both stacks.
\ Use this if something aborts and you don't want the
\ stacks growing forever.
: clear-o&mstacks
  ostack-init
  mstack-init
;

\ ************************
\ ** TEMPORARY SEGMENTS **
\ ************************
\ The method names and instance variable names are kept in
\ temporary segments.  These segments are allocated
\ from DOS.  When you're done defining things and
\ its time to make an executable, just free those
\ segments. (The word 'end-objects', defined later,
\ does all that.

\ Compare two counted strings. seg1 addr1, seg2 addr2 point to
\ segment and addresses of two strings with preceding count
\ bytes. Returns 0 if equal, else nonzero
: ccompl    ( seg1 addr1 seg2 addr2 -- n )
  \ First check byte counts
  count1 >r 2swap count1 r@ -
```

Die Namen der Instanzvariablen exsistieren nur innerhalb der Klassendefinition. Anders ausgerückt, wenn der Code des Listing 1 abgelaufen ist, wird der symbolische Name LOCALDATA wieder verschwunden sein (das Wort ;CLASS sorgt dafür).

Botschaften und Methoden

Listing One. Source code for a simple class

```
:class integer
2 ivar localdata

:m get:    ( -- n )
   localdata @
;m

:m put:    ( n -- )
   localdata !
;m
;class
```

Objekte der Klasse Integer verstehen die zwei Botschaften GET: und PUT:. Jede Botschaft passt zu einer Methode gleichen Namens. Ich kann ein Objekt des Typs Integer erzeugen und dort hinein eine 12 legen mit

```
INTEGER MYINT
12 PUT: MYINT
```

Du ahnst es schon, mit der Botschaft GET: kann ich den Inhalt von MYINT holen. GET: und PUT: sind der einzig legale Weg um die Instanzdaten vom Programm aus zu manipulieren. Ich sage hier ausdrücklich "legal", weil ein cleverer Forthprogrammierer natürlich alles manipulieren kann. Die Richtlinien für objektorientiertes Programmieren verbieten es aber, das Innere der Objekte über andere Wege (low level) zu verändern.

Gelegentlich brauchst du schon bei der Objekterzeugung den automatischen Ablauf einer Methode. Ein gutes Beispiel dafür ist eine Feldklasse, deren Größe erst festgelegt weden kann, wenn sie erzeugt wird.

Die Initialisierungsmethode soll dann den Speicherplatz reservieren

und in einer Instanzvariablen die Anzahl der Feldelemente ablegen.

Listing 2 zeigt die Syntax mit der so eine initialisierende Methode aufgebaut wird. Ich definiere dort die Methode CLEAR: um eine Null in die lokalen Daten zu schreiben. Das Wort < INIT METHOD muß vor dem Wort ;CLASS stehen. Das Listing 2 zeigt auch wie man SELF gebraucht. Damit ruft das Objekt sich bei der Methodendefinition selbst auf. Das wirkt als ob die Methode CLEAR: sagen würde: " Okay, Objekt, hier ist noch eine Null auf dem Stack. Sende dafür nun die Botschaft PUT: an dich selbst."

Listing Two. Initialization and selferencing

```
:m clear: ( -- )
   0 put: self
;m

clear <init-method
```

Vererbung

Der Begriff "Vererbung" gehört zu den großen Schlagworten der objektorientierten Programmierung. Das Konzept ist simpel: Du beginnst mit einfachen Klassen und baust daraus nach und nach Definitionen für Klassen höherer Ordnung und stärkerer Spezialisierung. Jede neue Klasse innerhalb des Erbgangs beinhaltet dann das gesammte Wissen ihrer Vorfahren (d.h. deren Methoden und Instanzvariablen).

Nehmen wir einmal an, ich hätte eine Klasse 1DARRAY definiert, mit der man eindimensionale Felder erzeugen kann. Objekte dieser Klasse haben Methoden um dem Feld Speicher zuzuweisen, die Indexstelle mit einen Wert zu laden und diesen auch wieder herauszuholen.

Nun definiere ich die Klasse $PTR_ARRAY um Zeiger für Zeichenketten in einem Feld zu bearbeiten. Diese neue Klasse soll wissen, wie man mit eindimensionalen Feldern umgeht (so wie 1DARRAY), soll aber auch eine Zeichenkette ausgeben können. Anders ausge-

```
  if        r> compl          \ Lengths match...try comparison
  else      4drop r>drop 1    \ Show mismatch
  endif
;

\ Advance to next item past the current counted string.
\ seg:addr points to counted string.  Takes into account
\ trailing integer.
: nextstr   ( seg addr -- seg addr' )
  count1 + 2+
;

\ Search one of the temporary segments.
\ seg1:addr1 points to string to search for
\ seg2 is temporary segment to search
\ max is maximum current offset in segment.
\ n is returned associated integer; -1 means
\ the string was not located.
: search-tseg     ( seg1 addr1 seg2 max -- n )
  dup                                  \ Anything to look for?
  if
      >r                               \ Save max
      0                                \ Start search at zero
      begin
            2over 2over                \ Dup seg/address
            ccompl                     \ Look for match
            0=
            if    2swap r> 3drop       \ Clear stack
                  count1 + @1          \ Fetch value
                  exit
            endif
            nextstr                    \ Advance to next string
            dup r@ >=                  \ Topped out?
        until
        r>drop                         \ Clear return stack
  endif
  4drop                                \ Clear stack
  -1                                   \ Show error
;

\ **
\ ** Instance variable segment handling
\ **
\ The ivar segment is a temporary region where the system
\ keeps a list of the current class definition's instance
\ variables.  Each entry is composed of a length byte, the
\ name, and a 2-byte value that indicates that instance
\ variable's offset into an instance of the class

\ Allocate space for the IVAR segment.  Place the segment
\ in global variable ivar-seg
: alloc-ivar_seg  ( -- )
  IVAR_SEG_SIZE alloc
  error                                \ Fetch error
  if  abort" Ivar allocation error"
  endif
  ivar_seg !                           \ Save pointer
;

\ Clear the ivar segment
: clear-ivar_seg  ( -- )
  0 ivar_next !
;
```

(Listing continues.)

(PCYerk Listing Four, continued.)

```
\ Given that addr points to a counted string that represents
\ an instance variable name, return the associated offset.
\ If you can't find that variable, return a -1.
: search-ivar     ( addr -- n )
  vars swap                \ String is in vars segment
  ivar_seg @               \ Search through ivars segment
  ivar_next @              \ Max. to look for in ivar segment
  search-tseg              \ Search a temp. segment
;

\ addr points to a counted string that represents an instance
\ variable name.  n is the offset to attach to that instance
\ variable.  Add this name to the list.
: add-ivar  ( n addr -- )
  dup c@ >r                          \ Save byte count
  vars swap                          \ Source address
  ivar_seg @ ivar_next @             \ Destination
  $!l                                \ Copy the string in
  ivar_seg @ ivar_next @ r@ + 1+ !l  \ Store associated value
  r> 3 + ivar_next +!                \ Advance next
;

\ **
\ ** Methods name segment handling
\ **
\ The methods segment looks a lot like the IVARS segment.
\ it holds the list of methods defined within the system.
\ Associated with each method name is a unique 2-byte
\ id.


\ Allocate space for the method name segment
: alloc-methname_seg    ( -- )
  METHNAME_SEG_SIZE alloc
  error                                \ Fetch error
  if  abort" Methname allocation error"
  endif
  methname_seg !                       \ Save pointer
;

\ Clear the method segment
: clear-methname_seg    ( -- )
  0 methname_next !
;

\ Search for a method in the methods segment.  Return -1 if
\ not found.  Else return method #
: methname-find   ( addr -- n )
  vars swap                \ String is in vars segment
  methname_seg @           \ Search through ivars segment
  methname_next @          \ Max. to look for in ivar segment
  search-tseg              \ Search a temp. segment
;

\ Add a new method to methods segment.  Associate n with that
\ method as the method's id
: add-methname    ( n addr -- )
  dup c@ >r                            \ Save byte count
  vars swap                            \ Source address
  methname_seg @ methname_next @       \ Destination
  $!l                                  \ Copy the string in
  methname_seg @ methname_next @ r@ + 1+ !l
                                       \ Store associated value
```

drückt: Die Klasse $PTR_ARRAY soll die Eigenschaften von 1DARRAY erben, hinzugefügt wird dann noch etwas für die Verarbeitung von Zeichenketten.

PCYerk bietet Vererbung über das < SUPER Wort an. Das geht so:

```
: CLASS
PTR_ARRAY
<< SUPER 1DARRAY
```

Es wurde die Klasse PTR_ARRAY gebildet, deren Superklasse die 1DARRAY Klasse ist.

Das << SUPER Wort macht mehrere Dinge. Zuerst wird im Kopfbereich der Klassendefinition ein Bindeglied (link) eingetragen. Dieser Zeiger weist zum Kopf der Superklassendefinition. Immer wenn das Ojekt eine Botschaft erhält, durchsucht es zunächst die Methodenliste seiner eigenen Klasse. Findet es dort keine passende Methode, durchsucht es über das Bindeglied auch die Methodenliste seiner Superklasse - solange bis eine Methode gefunden ist oder alle Klassen dieses Erbgangs untersucht sind. Das ist der Mechanismus der Methodenvererbung. (Merke dir folgendes: Weil die Suche zuerst durch die lokalen Methoden führt, kann das Objekt seinen Erbgang redefinieren.)

Danach initialisiert << SUPER den Akumulator für Platzbedarf von Instanzvariablen mit dem Befarf der Superklasse. Die Klasse erbt also den Instanzvariablenplatzbedarf seiner Superklassen.

Schließlich wird die Initialmethode der Superklasse mit Hilfe von << SUPER als Initialmethode der eigenen Kasse eingetragen. Die Klasse erbt die Initialmethode der Superklasse. Wie schon weiter oben beschrieben, kannst du diese Initialmethode mit Hilfe von <<INIT-METHOD überschreiben. Vererbung ist die Basis für *Polymorphismus*- ein weiterer objektorientierter Begriff. Polymorphismus meint, daß unterschiedliche Abkömmlinge einer Superklasse auf gleiche Botschaften doch unterschiedlich reagieren. So könnte ich eine Subklasse zu INTEGER definieren als INTEGER_ARRAY. Ein Integerobjekt liefert dann auf die Botschaft GET:

hin den Wert seiner einzigen Instanzvariablen ab. Dagegen nimmt ein Objekt der Klasse INTEGER_ARRAY bei gleicher Botschaft erst einmal einen Index vom Stack und liefert danach den Inhalt des zugehörigen Feldelementes aus. Gleiche Botschaft, andere Aktion.

Binden

Den Vorgang, mit dem Selektor einer Botschaft die ausführbare Adresse zu bestimmen (d.h. die Methode), nennt man "binden". Denn die Methode hängt ja ab von der Objektklasse, an die die Botschaft ging. Es gibt zwei Arten zu binden: Frühes und spätes Binden (early and late binding).

Frühes Binden hast du schon kennengelernt, es sieht so aus:

GET: MY_INTEGER

liefert den Inhalt aus MY_INTEGER. Das System kennt hier schon die Botschaft (get:) und das Objekt (my_integer) und findet so die passende Ausführungsadresse für die Botschaft bereits zur Compilezeit.

(Der innere Ablauf ist der: Wenn das System den Namen der Botschaft erkennt, legt es eine 16bit-ID auf einen Spezialstack, den Methodenstack - kurz MSTACK. Außerdem legt es seine Objektadresse auf einen weiteren Spezialstack, den Objektstack - OSTACK - und gibt die Kontrolle an das Wort EXEC-OBJ ab, das nun mit dem Id vom MSTACK bindet. Die Instanzvariablen gewinnen ihre Laufzeitadressen zurück mit Hilfe der Offsets, die an Hand der Objektadressen auf dem OSTACK kalkuliert wurden.)

Späte Bindung wird auch aufgeschobene Bindung genannt (deferrd binding). Einfach ausgedrückt heißt das, dein System weiß noch nicht an welches Objekt du die Botschaft senden willst. Also kann es zur Compilezeit die Botschaft noch nicht an die Ausführungsadresse binden.

Hier ein Beispiel für PCYerk's spätes binden:

GET: { INTEGEROBJ @ }

```
  r> 3 + methname_next +!           \ Advance next
;

\ ***************
\ **  METHODS  **
\ ***************
\ Methods are kept on singly-linked list. That list is anchored
\ in the class definition structure defined below.  Each entry
\ on a method list looks like this:
\      Token segment
\      [  link to next  ]
\      [  Method id #   ]
\      [ ...tokens      ]

\ Attach a new method to tail. addr on top of stack is assumed
\ to be pointer into token segment
: new-method-tail ( addr -- )
  \ See if we are first method added. If so, attach to parent.
  curr_meth_tail @ ?dup 0=
  if  dup                                   \ Copy ourselves
            curr_class_off @ 2+
            !t
  else      over swap !t                    \ Fix link
  endif
  curr_meth_tail !                          \ We are new tail
;

\ (>super)
\ This routine looks 'up the chain' to an object's super object.
\ Used when searching for methods to execute.
\ addr1 is the current object's address in the token seg.
\ addr2 is the super object's address or 0 if none found
: (>super)  ( addr1 -- addr2 )
  4 + @t                        \ Fetch the super object address
;

\ (domethod)
\ Following code word vectors execution to a method.
\ Assumes that the value on top of the stack is offset
\ into token space for the method.
code (domethod)          ( off -- )
      bp    dec                 \ Make room on return stack
      bp    dec
      si 0 [bp] mov             \ Push IP
      bx si mov                 \ Get method address in IP
      bx    pop                 \ Pop stack
      next                      \ Take off!
end-code

\ domethod
\ Calls (domethod) and clears the object stack.
\ Off is the address of the method code.
: domethod        ( off -- )
  (domethod)
  odrop
;

\ (methid->addr)
\ Given a method id, this finds that method's address in the
\ token segment. addr1 is the address of the object (in the
\  token segment) whose method
\ list we'll search.  addr2 is the method address, or 0 if the
\ method wasn't found.
```

(Listing continues.)

(PCYerk Listing Four, continued.)

```
: (methid->addr)   ( addr1 n -- addr2 )
  swap 2+                              \ Advance to method pointer
  begin
      @t                               \ Fetch pointer
      dup
  while
      dup>r                            \ Save copy
      2+ @t                            \ Fetch id number
      over =                           \ Match?
      if    drop                       \ Clear method id #
            r> 4+                      \ Point to code
            exit
      endif
      r>                               \ Ready for next loop
  repeat
  2drop 0                              \ Show failure
;

\ find-method-code
\ Expects a method id # atop the method stack and an object
\ pointer atop ostack. Locates the method code and
\ leaves it on dstack. In so doing, the method stack is popped.
: find-method-code       ( -- code )
  mpop                                 \ Get method id
  ostack->dstack                       \ Fetch the object
  @                                    \ Address in token seg
  begin
      2dup swap (methid->addr)         \ Get address
      ?dup                             \ Didja find it?
      if    -rot 2drop                 \ Clear the stack
            exit                             \ Bug out
      endif
      (>super)                  \ Not found...go to super object
      ?dup                      \ Any super object??
    0=
  until
  clear-o&mstacks                                \ Clear the stacks
  abort" Method not found"
;

\ Define a method.  This word doesn't do a create...it
\ loads the method name in the method segment (unless
\ its already there), then compiles the code at the
\ end of the object definition.  The code is linked to
\ the preceding method for that object.
: :m
  \ We must be defining a class
  curr_class_off @ 0=
  if  clear-o&mstacks
            abort" Method def. ouside class"
  endif

  \ We are the new method tail...so fix the link
  \ code.
  here-t new-method-tail
  0 ,t

  \ See if the method is in the method seg.  If it is,
  \ return the method #...if not, add this method in and
  \ assign a number.
  blword                               \ Parse the name
  tstring $!                           \ Put it in tstring
  tstring methname-find                \ Look for the method
  dup -1 =                             \ Found?
```

Hierbei nehme ich an, daß die Variable INTEGEROBJ eine Objektadresse beinhaltet. Zur Compilezeit gibt es keinen Weg für das System zu erfahren, welches Objekt in INTEGEROBJ sein wird, wenn das Programm später einmal läuft. Das System kann daher erst zur Laufzeit von GET: die Ausführungsadresse bestimmen.

Mit den Worten { und } erreicht man spätes Binden. (Yerk benutzte [und], aber diese Worte waren in UDForth schon vergeben.) Die geschweiften Klammern sollen direkt hinter der Botschaft stehen und dürfen jeden Forthausdruck enthalten, der eine Objektadresse liefert.

Natürlich läuft Code mit später Bindung langsamer ab als Code mit früher Bindung. Das liegt daran, daß spätes Binden Codeabläufe in die Laufzeit verschiebt, die sonst zur Compilezeit abgelaufen wären.

Mit später Bindung kann man aber folgende Schwäche des PCYerk umgehen, ohne allzuviel Extracode zu brauchen: Du kannst alle Instanzvariablen nur mit dem Wort IVAR definieren. Ein Objekt kannst du nicht als Instanzvariable einsetzen. Angenommen, du definierst die 1DARRAY Klasse für eindimensionale Felder und eine Klasse für polygone Objekte. In der Polygonklasse wäre es schön, wenn das Objekt 1DARRAY eine der Instanzvariablen sein könnte. Dann wärst du in der Lage, die Liste der Polygonekken in das eindimensionale Feldobjekt einzuspeichern (polygon's vertex list).

Und eben das geht, wenn man eine der Instanzvariablen als Zeiger auf ein Objekt aus der 1DARRAY Klasse anlegt. Eigentlich baust du damit ein Objekt ohne Kopf (headerless); ohne Kopf im Forth Wörterbuch. Das Wort INSTANTIATE erzeugt so ein kopfloses Objekt. Dazu braucht INSTANTIATE auf dem Stack nur die Startadresse in der Variablenzone (wo das neue Objekt sein wird) und einen Zeiger auf die Klassendefinition. Der Code in Listeing 3 zeigt dies in der Annahme, daß du zuvor schon 1DARRAY definiert hast.

Beachte das ich INIT: so definiert habe, daß die Eckenanzahl mit zwei multipliziert werden muß um

die Anzahl der Einträge zu erhalten, denn die Eckenliste beinhaltet für jede Ecke sowohl X- als auch Y-Koordinaten. Die folgende Zeile läd die Adresse für den nächsten freien Platz im Variablensegment nach ^VERTEX_LIST hin, und wird so zum Zeiger auf unser eindimensionales Feld. Dann benutze ich ['] um die Codeadresse der 1DARRAY Klasse zu gewinnen, und hole die Adresse dieser Klassendefinition. Das INSTANTIATE Wort erzeugt dann das eigentliche Feld. Merke, wenn INSTANTIATE läuft, wird die Initialmethode des Objekts 1DARRAY ausgeführt: Speicherplatz zuweisen.

Nun können wir mit der Methode TO: eine XY-Koordinate in das 1DARRAY Objekt packen; ich nehme dabei an, daß diese Methode für die 1DARRAY Klasse schon definiert worden ist. Du kannst diese Idee nun ausbauen soweit du willst, und Zeiger auf Objekte schaffen die Zeiger auf Objekte haben und so weiter.

Zum Kern der Sache

Betrachte Listing 4, die komplette Quelle für PCYerk. CLASS: ist das Herzstück von PCYerk. CLASS: ist aus verschachteltem CREATE ... DOES> gemacht - die Sorte von der man Kopfschmerzen bekommt. CLASS: erzeugt beim compilieren einen Klassenkopf wie den in Bild 1.

An dieser Stelle muß ich etwas zur Struktur von UDForth sagen. Da es auf dem IBM PC arbeitet ist es segmentiert angelegt. Im Codesegment liegt ausführbarer Code, Im Variablensegment werden die Werte der Variablen gespeichert, das Tokensegment hält die Fädelzeiger und die Namen werden im Headersegment abgelegt. Namen, die mit Forth's CREATE erzeugt worden sind, liefern einen Parameterfeldzeiger auf das Variablensegment, die Klassendefinition liefert einen Klassenzeiger in das gleiche Segment.

Das erste Feld im Kopfbereich - Platz für IVAR - teilt dem System mit, wieviel Platz es für Instanzvariablen bereitstellen muß, wenn ein Objekt erzeugt wird. Das zweite Feld ist der Anfang einer verkette-

```
  if  drop                                \ Clear stack
        curr_meth# @ dup                  \ Fetch current method ID #
        tstring add-methname              \ Add method to the class
        1 curr_meth# +!                   \ Bump current method ID #
  endif

  \ Store method # for ;m and set aside space in token seg
  my_meth# !
  0 ,t

  \ Now go ahead and compile the method code.
  [compile] ]
;

\ End of method definition
: ;m
  \ Store the method # so the system can find it
  my_meth# @
  curr_meth_tail @ 2+ !t

  compile unnest                          \ Do a semicolon
  [compile] [                             \ Set interpret state
; immediate

\ *************************
\ **  INSTANCE VARIABLES  **
\ *************************
\ Define an instance variable.
\ Used in the form:
\  n ivar <name>
\ n indicates # of bytes for this instance variable.
: ivar      ( n -- )
  blword                                  \ Parse the name
  tstring $!                              \ Put it in tstring

  \ See if ivar already exists
  tstring search-ivar -1 <>
  if  abort" Ivar already defined"
  endif

  \ Fetch current offset--add it and ivar to ivar space
  curr_class_off @ dup @t dup
  tstring add-ivar

  \ Update ivar space for next offset
  rot +
  swap !t
;

\ This code does the actual instance variable processing.
\ When he executes, he expects the offset of an instance
\ variable on the data stack.  He also expects an object
\ address (in variable segment) on the ostack.
\ The returned addr is the offset to the instance variable.
: (do-ivar)       ( off -- addr )
  ostack->dstack                          \ Get object address
  2+                                      \ Skip pointer to token seg
  +                                       \ Add offset
;

\ ***************
\ **  CLASSES  **
\ ***************
```

(Listing continues.)

(PCYerk Listing Four, continued.)

```
\ exec-obj
\ This fellow expects an object pointer (in vars segment) atop
\ the object stack and a method # atop the methods stack.
\ Executes an object's method
: exec-obj   ( -- )
  \ Find the method's executable code
  find-method-code
  domethod
;

\ instantiate
\ addr1 is current pointer in var seg
\ addr2 is object's token pointer
\ Stores that pointer in the
\ variable segment, then allocates ivars space.
: instantiate      ( addr1 addr2 -- )
  dup                     \ Make copies of token pointer
  dup ,                        \ Store token pointer in var seg
  @t                           \ Fetch ivar space
  allot                        \ Allocate variable storage
  6 + @t                       \ Fetch startup method
  dup -1 <>                    \ Anything there?
  if  mpush                    \ Push method
            opush              \ Push object
            exec-obj           \ Execute stuff
  else
            2drop              \ Drop -1 and object pointer
  endif
;

\ **
\ ** Class definition
\ **
\ The contents of a defined class are:
\    Token segment:       Vars segment:
\    [  Ivars space ]<---[ token ptr  ]
\    [  Meth list   ]
\    [  Super ptr   ]
\      [  start meth  ]
\    [ ..tokens     ]
\ Note that the code following does> can do a @ and
\ retrieve the offset into token space for the class
\ definition structure.
\
\ Once instantiated, an object looks like this:
\     Token segment:     Vars segment:
\     [  (;code)     ]    [  token ptr  ]   <<<< To parent class
\     [  here-c      ]    [ ...ivars    ]
\     [  ..tokens    ]
: :class
  0 curr_meth_tail !              \ No methods yet
  clear-ivar_seg                  \ No instance variables
  create                          \ Build the name field
      here-t dup curr_class_off !  \ Set current class
      ,                           \ Build pointer in vars seg.
      0 ,t                        \ Size of ivars region
      0 ,t                        \ Pointer to list of methods
      0 ,t                        \ Pointer to superclass
      -1 ,t                       \ Initial method
  does>
      @                           \ Fetch token pointer
      here swap                   \ Get current object pointer
      create                      \ Make a header
            instantiate           \ Instantiate the object
```

ten Liste aller Methoden einer bestimmten Klasse. Als nächstes kommt das Superklassenzeigerfeld das durch <<SUPER gesetzt wird, womit die Klasse die Methoden der Superklasse erbt. Das letzte Wort im Klassenkopf ist der Startselektor; darin wird die Methode bestimmt, die automatisch bei der Objekterzeugung ablaufen soll.

Die Objekte tragen Ausführungsadressen definiert vom Code hinter dem zweiten DOES> in CLASS:. Wenn du eine Botschaft an ein Objekt abgegeben hast, folgt das System dem Zeiger zum Klassenkopf und durchsucht danach die Liste der Methoden nach dem gewünschten ausführbaren Codestück. Beachte, daß der Code einer Methode absolut ohne Kopfeinträge dasteht; eine Methode hat nicht einmal eine Codefeldadresse. Ein spezielles Wort - (DOMETHOD) - führt durch Nachahmung der Laufzeitaktion eines Colonwortes die Methode aus. An (DOMETHOD) wird die Startadresse der Methode übergeben und um den Rest kümmert es sich selbst.

PCYerk erzeugt keine üblichen Forthkopfeinträge (á la create) für Instanzvariablen und Methoden. Die Namen der Instanzvariablen sollen wieder verschwinden, wenn die Definition einer Klasse fertig ist. Die Methoden benötigen keinen Extraspeicher für Namensköpfe, weil sie umgehend zu zwei Byte langen ID-Nummern verarbeitet werden.

PCYerk belegt zwei Speicherblöcke (das UDForthwort ALLOC weist über die DOS-Funktion ein Speichersegmente zu): Ein Block für die Namen von Instanzvariablen, der andere für die Methodennamen. Jedem Namen wird eine Integerzahl zugeordnet.

Bei einer Instanzvariablen befördert diese zugeordenete Integerzahl ein Offset der Instanzvariablen in den lokalen Datenspeicher des Objekts. Wenn das System beim compilieren dann einer Instanzvariablen begegnet, schaut es den Offset der Variablen nach und compiliert diesen als LITERAL gefolgt vom Wort (IVAR) in das Objekt. Zur Laufzeit nimmt (IVAR) also den Offset vom Stack und fertigt daraus die Adresse.

Im Fall der Methoden ist die assoziierte Integerzahl einfach die ID-Nummer, der Selektor somit. Wenn du den Namen einer neuen Methode definierst, wird im System intern ein Methodenzähler erhöht und die neue Methode bekommt diese Nummer. Damit hat jede Methode eine eigene Nummer.

(Yerk erzeugte solche ID-Nummern durch 'hashing'. Ich wählte einen anderen Weg, weil der Code dafür bei den Instanzvariablen und den Methodennamen ganz ähnlich ist.)

Aber halt! Wenn die Methodennamen und die Namen der Instanzvariablen abwechselnd in jenen Segmenten abgelegt werden, wie findet Forth die Namen bei der Compilation wieder? Du mußt INTERPRET anpassen.

UDForth sucht ein Wort zuersteinmal im Wörterbuch. Gelingt das nicht, versucht INTERPRET das Wort als Zahl zu behandeln. Wenn die Umwandlungsroutine keine Zahl draus machen kann, führt INTERPRET das Wort DO-UNDEFINED aus (das unbekannte Wort wird ausgegeben und sodann QUIT ausgeführt). Ich überschreibe den Aufruf von DO-UNDEFINED mit einem Zeiger auf <INTERP-PATCH>. Dieses Wort (siehe Listing 4) schaut zuerst im Methoden- und dann im Instanzvariablensegment nach. Findet es in einem der Segmente das gesuchte Wort, wird die passende Aktion ausgeführt. Sonst fällt <INTERP-PATCH> einfach hindurch und landet wieder in DO-UNDEFINED. Damit man überhaupt Klassen definieren kann, mußt du zu erst einmal START-OBJECTS ausführen. Das Wort START-OBJECTS stellt für die Instanzvariablen- und Methodennamen Segmente bereit und initialisiert diese, flickt dann INTERPRET und räumt schließlich noch den Methoden- und Objektstack auf. Wenn du alles fertig hast um eine eigenständige Anwendung (standalone application) zu schaffen, führe zum guten Schluß END-OBJECTS aus. Dadurch wird INTERPRET in den Urzustand zurückgestezt und die nun überflüssigen Speicherblöcke werden freigegeben.

```
        immediate               \ Make the object immediate
    does>
        opush                   \ Get object ptr. on ostack
        state@
        if                      \ We are compiling
            compile (lit)       \ Compile obj ptr. as literal
            ostack->dstack      \ Get object pointer
            ,t                  \ There's the pointer
            compile opush       \ Compile an object push
            compile (lit)       \ Another literal is
                                \ method code pointer
             find-method-code   \ Get method's code pointer
             ,t                 \ Compile that
             compile domethod   \ Code to execute method
             odrop              \ Don't need object anymore
        else                    \ We are interpreting
             exec-obj           \ Execute the object
        endif
;

\ Complete a class definition
: ;class
  clear-ivar_seg                \ No ivars segment
  0 curr_class_off !            \ No current class
;

\ Special word that returns current object so object
\ can send a message to itself.  Use 'self' inside
\ the methods definitions to refer to the current object.
: self      ( -- )
  compile (lit)
  curr_class_off opush                  \ Get current object
  find-method-code                      \ Locate method code
  ,t                                    \ Store as literal
  compile odup                          \ Dup object
  compile domethod                      \ Execute method
  odrop                                 \ Clear object stack
; immediate

\ Define a class's super class.
\ A class will inherit instance variable space, methods, and
\ startup methods from the super class.  A class can override
\ methods and startup methods.
: <super
  \ Find the object and resolve code address to token address
  blword find 0=
  if  abort" Super object not found"
  endif
  >body @ dup
  \ Store token address into super pointer of current class
  curr_class_off @ 4 +
  !t
  \ Copy ivars into local ivars
  dup @t curr_class_off @ !t
  \ Copy initial method
  6 + @t curr_class_off @ 6 + !t
;

\ Define initialization method.
\ This routine expects a method id on the top of the method
\ stack.  It stores that method id as the object's startup
\ method.
```

(Listing continues.)

(PCYerk Listing Four, continued.)

```
: <<init-method         ( -- )
  mpop
  curr_class_off @ 6 +
  !t
;

\ **********************
\ ** DEFERRED BINDING **
\ **********************
\ Deferred binding allow you to specify the object at runtime,
\ rather than at compile time.

\ { Starts deferred binding.  He assumes there's a method # on
\ top of the method stack.  He copies that as a literal into
\ inline code (along with an mpush).
: {
    state@
    if                                      \ We are compiling
            compile (lit)                   \ Compile literal
            mpop ,t                         \ Get method #
            compile mpush                   \ Compile mpush code
      endif                           \ Interpreting--do nothing
; immediate

\ } Concludes a deferred method.  He assumes there will be
\ (at runtime) a method # on top of the method stack and an
\ object pointer atop the data stack.  He pushes the object
\ pointer onto the object stack, finds the method, and executes
\ it.
: }
  opush                                     \ Push object pointer
  exec-obj                                  \ Execute it
;

\ ************************
\ ** PATCHES AND MISC. **
\ ************************
\ Following code is the patch to interpret.
\ Allows system to recognize methods and instance variables.
\ NOTE: When we get here, literal? has left 2 zeros on stack.
\ For uniformity's sake...we pass them on along.
: <interp-patch>
  2drop                                     \ Clear stack
  \ See if the item in question is a method.  If so, leave the
  \ method id # on the method stack
  here methname-find dup
  -1 <>
  if  mpush exit                            \ Push the method #
  else      drop
  endif

  \ Not a method -- see if it's an ivar
  here search-ivar dup
  -1 <>
  if  ?comp                         \ GOTTA be compiling
            compile (lit)           \ Compile ivar value
            ,t
            compile (do-ivar)       \ Compile ivar handler
            exit
  else      drop
  endif
```

Zum Autor:

Rick Grehan ist 'senior editor' beim BYTE Magazin und dort technischer Direktor des Laboratoriums. Er entdeckte Forth von sieben Jahren bei der Entwicklung eines Kontrollsystems für Musik basierend auf einem KIM-1.

Seither gebraucht er Forth auf 68000 Systemen einschließlich Macintosh und noch immer auf AppleII und jetzt IBM PC. Er arbeitete viel mit dem SC32, einem stackorientierten Prozessor. Rick hat den 'B.S. degree' in Physik und angewandter Mathematik sowie einen 'M.S. degree' in Mathematik/Computerwissenschaft.

```
> Stack ops:
>  DUP ( create a stack element)
>  DROP ( destroy a stack
        element)
>  SWAP ( move stack element)
>  >R and R> (stack exchange)
> Arithmetic/logic:
>  LITERAL ( constants, etc.)
>  NAND ( sounds silly,
   but you can
   synthesize anything
    else out of it!)
> Address space access:
>  ! and @ ( or C! and C@,
     if you wish)
>  P! and P@ ( for i/o space
    port access,
   your CPU has such a thing...)
> Dictionary extension:
>  CREATE
> This is a _VERY_ sparse list!

  Uwe Kloss antwortete ihm:
> And to me it seems to be
  too sparse!
>
> If you consider the stacks
   as entities your
  stack ops are ok!
> But then you can NOT a
  ccess stack pointers!
> If you prefer to have
  the stack pointers as part
  of the virtual
> machine implemented you
  should write:
>
>    SP!
>    SP@
>    RP!
>    RP@
>
> And write the stack ops
  as colon definitions.
>
> This means you assume (!)
  that your stacks are
  memory mapped!
  ...
> But you should definitely
       add EXECUTE to that list.
> How do you implement
  EXECUTE for primitives
  as a colon definition?
>
> And a NAND operation
   is not enough. If you want
   to do arithmetics you
  > need shift operations
   as well!
> Left AND right!
>
> On the other hand you
   can easily replace LITERAL
   by a colon definition.
  >
> Uwe
>
> P.S.:      I claimed LITERAL
'easy'!
>          Ok! Here it is!
>
>    : LITERAL
```

```
  0 0                                  \ Look like literal?
  \ Let do-undefined handle things
  do-undefined
;

\ Following code patches interpret.  Do it AFTER you've
\ allocated methods and variable segments
: patch-interpret
  ['] <interp-patch>
  ['] interpret >body 40 + !t
;

\ Put interpret back the way it was.
: unpatch-interpret
  ['] do-undefined
  ['] interpret >body 40 + !t
;

\ Initialize the system.
\ One you've included [i.e., loaded] this code, you must
\ execute "start-objects" before you can begin defining
\ any objects.  When you're done defining and calling all
\ your objects [i.e., you're about to make an executable],
\ execute "end-objects".
: start-objects
  alloc-ivar_seg              \ Allocate ivars segment
  clear-ivar_seg
  alloc-methname_seg          \ Allocate method name segment
  clear-methname_seg
  1 curr_meth# !              \ Start method #'s
  patch-interpret             \ Fix interpret
  clear-o&mstacks             \ Initialize the stacks
;

\ Clean things up
: end-objects
  unpatch-interpret                \ Put interpret back
  ivar_seg @ free                  \ Ditch ivars segment
  methname_seg @ free              \ Ditch method name segment
;
```

(End listing. Next issue contains code for basic & storage classes, byte & word arrays, strings, & string arrays.)

"Die Schweinekiste"

Forth hat den Ruf eine sog. WOL (Write Only Language) zu sein, also eine Programmiersprache, in der man zwar Programme schreiben, sie aber nie wieder lesen kann.

Nun, ganz objektiv betrachtet ist das so natürlich nicht war. Trotzdem - wenn man sich keine Mühe gibt, Forth-Programme lesbar zu gestalten, sind sie in der Tat unlesbar. Gibt man sich hingegen Mühe, so sind sie so leicht verständlich wie nur möglich - der bekannte "Forth Verstärker Effekt".

In der Praxis sollten NIEMALS unleserliche Programme erstellt werden.

Um denen, die es dennoch nicht lassen können, eine Möglichkeit zu geben, ihre Triebhaftigkeit ausleben zu können, ist die Schweinekiste erfunden worden.

Hier kann nach Herzenslust ausprobiert werden, was in der Praxis verboten ist. Unlesbarer, unstrukturierter, nicht zu durchschauender und kryptischer Code, über den andere Programmierer Minuten, Stunden, ja Tage und Wochen grübeln, müssen, um ihn zu verstehen. Ganz nach dem Motto "Schwein sein ist schön!"

So abreagiert kann dann die im Alltag notwendige Software erstellt werden: Übersichtlich, leicht verständlich, mit klaren Konzepten, prägnant und elegant.

Als Einstieg in die Schweine-Kiste hier ein Forth-Rätsel, das im USENET (comp.lang.forth) von Antti Karttunen (karttu@mits.mdata.fi) aus Helsinki gestellt wurde:

Hier ist das notorische rekursive Rätselwort, das ich vor Jahren erfunden habe:

```
: MYSTERY
    DEPTH 0= IF
     ELSE DEPTH 1 = IF
      ELSE DEPTH 2 = IF SWAP
       ELSE
           RECURSE
           SWAP
           RECURSE
           RECURSE
       THEN
      THEN
     THEN
;
```

Nun, es gilt zu verstehen, was das Wort tut:

1) Zuerst, indem es nur angesehen wird und vielleicht einige Stackdiagramme aufgemalt werden. Kein Erfolg? Bist du durch die Wirren der Rekursion schrecklich verwirrt? Dann probiere doch die nächste Methode:

2) Lade das Wort einfach in Dein Forth-System, lege einige Werte auf den Stack und probiere es aus. Es sollte eigentlich offensichtlich sein, was es tut. (Hinweis: .S ist in diesem Zusammenhang sehr hilfreich.)

3) So - nun zurück zu Punkt 1 und den Code nocheinmal erforschen. Nach alldem sollte es eigentlich vollkommen klar sein, was MYSTERY tut und auch WIE es das macht.

Was ist der Unterschied zwischen Punkt 1 und 3?

Das Geheimnis liegt darin, daß Du, nachdem Du MYSTERY in Punkt 2 getestet und seine Wirkung ermittelt hast, nun schon den richtigen Namen von MYSTERY kennst. Den RECURSE-Aufrufen im Rumpf von MYSTERY kannst Du dann schon eine Bedeutung geben. Vorher war da nur ein Haufen Rekursion. (Das ist der eigentliche Grund, warum ich das Wort in MYSTERY umbenannt habe. Erst dadurch wird's zum Rätsel ;-)

Ich frage mich, was wohl Mathematiker/Logiker dazu sagen würden, wenn sie dieser Art Schlüsse zu ziehen gegenübergestellt würden.

Punkt 3, obwohl selber deduktiv, basiert auf der Annahme, die in Punkt 2 ermittelt wurde. Sie aber ist induktiv oder eignetlich nur geraten, da man in Punkt 2 nicht sicher sein kann, daß MYSTERY wirklich immer so arbeitet, wie es den Anschein hat (Ich meine hier keine Ausnahmesituationen wie Stacküberlauf oder so.) Also vielleicht handelt es sich hier um soetwas wie zirkuläres Schließen? Im Moment bin ich auf der Suche nach einer generellen Methode rekursive Funktionen zu beweisen.

(Wahrscheinlich benutze ich hier nicht die richtigen Logik-Fachausdrücke. Ich bin selber kein Logiker, tut mir leid ;-)

Wir haben also etwas allgemeines über rekursive Funktionen gelernt. Laßt Euch nicht von Rekursionen ("depth-first") beeindrucken. Zuerst einmal probiere, dem rekursiven Aufruf eine konzeptuelle Bedeutung zu geben ("breadth-first"), entweder indem Du dem Namen der Funktion traust (wenn er selbsterklärend ist) oder indem Du einfach die Bedeutung rätst. Dabei ist es eine wichtige Sache, herauszufinden, was die Funktion macht wenn sie NICHT rekuriert.

Forth-Kurs in Moers

Autor: Friederich Prinz

Die Moerser Forthgruppe versucht, seit gut 4 Jahren vor allem Einsteigern eine Möglichkeit zu bieten, sich sowohl mit dem komplexen Gerät/System Computer vertraut zu machen, als auch den Umgang mit diesem Gerät gleich 'richtig' zu lernen - eben via FORTH.

Die FORTH-GRUPPE MOERS bietet für Einsteiger einen Kurs an. Die Gruppe besteht aus Mitgliedern der Deutschen Forthgesellschaft e.V. und 'unabhängigen' FORTHern aus dem Raum Moers. In diesem Beitrag stellt der Autor Teile des Kurses vor.

Wir haben bewußt versucht, FORTH als eine Art Transportmedium zu 'verkaufen', das dem Anfänger die Türen zu dem WIE ebenso eröffnet, wie auch zu dem WARUM. Im 2. Halbjahr 1991 haben wir uns ausgiebig mit dem Dateihandling in DOS-Maschinen beschäftigt und die grundlegenden Möglichkeiten der Arbeit mit Dateien aufgezeigt.

Unser konzeptionelles Angebot, FORTH als Medium zu benutzen, um quasi über die Notwendigkeit des 'überhaupt Lernens' erste Kontakte mit FORTH nebenher zu vermitteln, hat sich unserer Meinung nach als richtig erwiesen. Aus den Anfängen der Moerser Forthgruppe, die vor cirka 6 Jahren aus lediglich vier interessierten FORTHern bestand, ist heute eine Gruppe von 16 Menschen verschiedener Altersklassen und mit ganz unterschiedlichen beruflichen Hintergründen geworden. Der Anfängerkurs wird regelmäßig, wöchentlich (!) von 8 bis 10 Menschen besucht.

Nicht ohne Stolz möchten wir an dieser Stelle einfügen, daß die Anzahl der Menschen, die sich zu unserer Gruppe zugehörig fühlen, deutlich höher ist. Die 'ehemaligen' Anfänger aus den ersten Tagen der Gruppe treffen sich wesentlich seltener und unregelmäßiger und verabreden in kleinen Gruppen Projekte und Arbeitstreffen, die wir bewußt nicht in den Vordergrund der Anfängergruppe lassen. Aber spätestens bei unserem alljährlichen Sommerfest kommen doch immer wenigstens 20 FORTHer zusammen.

Seit dem 2. Halbjahr 1991 hat sich die Konzeption der Kurse noch etwas weiter spezialisiert. Dieses Semester war ganz und gar dem Dateihandling gewidmet, als Reaktion auf den Wunsch der Kursteilnehmer, diese Dinge 'machen zu können'. In 1992 werden wir uns, wie Sie den Texten auf der Diskette entnehmen können, verstärkt dem ZF FORTH von Tom Zimmer zuwenden. Wir arbeiten in Moers seit vielen Jahren mit diesem System, das wir vor allem für den Ausbildungsbereich als ganz besonders geeignet ansehen. ZF ist von mehreren Mitgliedern der Moerser Forthgruppe 'eingedeutscht' und in der Funktionalität an die Bedürfnisse des Lernenden angepaßt worden. Dabei hat ZF selbstverständlich nichts von seiner Praxistauglichkeit verloren.

Auch "WORTE.TXT" ist aus den Wünschen der Einsteiger hieraus entstanden. In dieser Datei werden (fast) alle Worte des ZF Systems in deutscher Sprache beschrieben. Wir gehen davon aus, daß eben doch nicht jeder die englische Sprache perfekt beherrscht. Eine Computersprache zu erlernen ist, vor allem wenn der Lernaufwand auf die Freizeit beschränkt bleiben muß, Anstrengung genug. Sich zusätzlich auf eine 'direkte' Fremdsprache konzentrieren zu müssen, kann nicht im Sinne desjenigen sein, der FORTH vermitteln möchte.

Die Kurstexte für 1992, die inhaltlich bereits teilweise vorgetragen wurden, beschäftigen sich, dem zuvor Gesagtem entsprechend, gezielt mit dem ZF Entwicklungssystem. Wie uns die Kursteilnehmer gelehrt haben, reicht es nicht, die grundlegenden Dinge einer Sprache zu vermitteln. Mindestens ebenso wichtig ist es, gezeigt zu bekommen, wie man diese Sprache 'in den Computer rein kriegt'. In diesem Sinne diskutieren wir im laufenden Halbjahr Grundsätzliches und versuchen SPRACHE, ENTWICKLUNGSUMGEBUNG und die notwendigen WERKZEUGE sinngemäß voneinander zu trennen. Dabei wollen wir versuchen, dem System Computer ebenso wie dem System ZF die Komplexität zu nehmen und einzelne Module überschaubar zu machen. Ob uns dies gelingen wird, können wir Ihnen frühestens im Sommer dieses Jahres mitteilen.

Die 'groben' Lernziele für einen jeweils laufenden Kurs diskutieren wir dazu in einem kleineren Kreis und stellen das Ergebnis dem jeweils laufenden Kurs vor. Die Kursteilnehmer selbst sagen uns dann, was sie davon 'haben möchten' und/oder was sie in der nächsten Zeit zu benötigen glauben. Diese Handhabung sorgt für Freude und Spaß bei der Vermittlung 'unserer' Sprache, wobei der Spaß auf beiden Seiten gleich groß ist.

Die Disketten mit den gesamten Kurstexten und die Forth-Program-

me dazu können bei mir angefordert werden.

Meine Anschrift:
Friederich Prinz,
Hombergerstraße 335,
4130 Moers 1,
Tel. 02841/58398 p
Tel. 02842/572120 d

1. Kurstag

Ziel der Deutschen Forthgesellschaft e.V. ist es, die Programmiersprache FORTH gerade Programmierern aus dem Hobbybereich näher zu bringen. FORTH als vollständige Programm-Entwicklungsumgebung bietet nicht nur dem professionellen Programmierer eine Reihe von Vorteilen. Der 2.Teil des laufenden FORTH-Kurses, der sich inhaltlich mit fortgeschrittenen Lernzielen befasst, steht unter der Schirmherrschaft des Deutschen Paritätischen Wohlfahrtsverbandes (DPWV), der die anfallenden Unterrichtsstunden bezahlt. Das Moerser Arbeitslosenzentrum stellt der Forthgruppe Moers seit deren Gründung Räume zur Verfügung, in denen die Gruppe sowohl ihre Treffen, als auch ihre Kurse organisieren kann.

Kursleiter und 'Assistenten'

Die Kurse der Forthgruppe wurden (werden) von Friederich Prinz, Moers, Hombergerstraße 335, erarbeitet. Die Kursinhalte des 'Einsteigerkurses' werden in gemeinsamer Arbeit von F.Prinz, Bernd Feuerer und Michael Major vorgetragen und mit der Gruppe erarbeitet.

Lernziel

Das Lernziel des vorliegenden Kurses ist ein Einstieg in die hardwarenahe Programmiersprache FORTH. Die Kursteilnehmer sollen mit dem Compiler/Interpreter System 'ZF' vertraut gemacht werden und zum Kursende in der Lage sein kleinere Programme und Definitionen selbstständig zu erarbeiten.

Einstieg in die Programmiersprache FORTH

Der Einstieg in die Programmierumgebung FORTH fällt gewöhnlich um so leichter, je weniger an Vorkenntnissen von den Kursteilnehmern mitgebracht wird. Deshalb geht der Kurs grundsätzlich von keinerlei Vorkenntnissen im Bereich Programmieren aus. Zur allgemeinen Bedienung des Computers sollten aber Grundkenntnisse zum jeweiligen Betriebsystem vorhanden sein. Die Kursteilnehmer sollten in der Lage sein ihr jeweiliges Betriebssystem zu starten, Disketten zu formatieren und einzelne Programme aufzurufen.

Warum FORTH ?

FORTH zählt, unberechtigter Weise, unter den Programmiersprachen zu den 'Exoten'. FORTH ist eine Programmierumgebung, die sehr nahe an der Hardware arbeitet. Das schreckt vor allem 'Hobbyprogrammierer' häufig ab. Trotzdem ist FORTH, wie kaum eine andere Sprachumgebung, geeignet 'richtiges' Programmieren zu lernen und zu lehren.

FORTH erfüllt alle wesentlichen Forderungen an eine moderne Programmiersprache und führt 'automatisch' zu einem tiefen Verständnis der Maschine Computer und deren Möglichkeiten. Der versierte Forthprogrammierer behält die absolute Kontrolle sowohl über die Maschine, als auch über das Betriebssystem der Maschine und letztlich auch die Kontrolle über 'fremde' Programme.

FORTH ist keine 'Blackbox'. Fast alle anderen Sprach- oder Entwicklungsumgebungen stellen sich dem Benutzer weitgehend 'natürlich-sprachlich' dar. Das heißt, daß der Benutzer in fast natürlichem Englisch der Maschine Anweisungen geben kann, die diese dann ausführt - oder auch nicht. Solche komplexen Umgebungen sind in ihrer Arbeit vom Benutzer in den meisten Fällen nicht nachvollziehbar. Stößt der Benutzer an Grenzen 'seiner' Sprachumgebung, geht es einfach nicht mehr weiter. Nicht so in FORTH. FORTH lernt gewissermaßen mit dem Benutzer.

Geschichtliches

Entstanden ist FORTH vor mehr als 20 Jahren an einer amerikanischen Universität. Charles M. Moore, ein Physiker und Radioastronom, hat Forth aus der Not heraus entwickelt, eine Programmiersprache zu brauchen, die schnell ablaufenden Code erzeugt (ca 1:5 bis 1:7 gegenüber Assemblern), die leicht zu handhaben ist und in der sich schnell programmieren, aber auch schnell eventuelle Programmfehler finden und beheben lassen. FORTH sollte so einfach wie BASIC beherrschbar sein, ohne dessen Geschwindigkeitsverluste in Kauf zu nehmen.

Ch. M. Moore stellte sein erstes FORTH der Public Domain Bewegung zur Verfügung. Vor allem in den USA, und dort im wissenschaftlichen Bereich, sowie im Bereich von Steuerungen, gewann FORTH recht schnell an Popularität und kam Mitte der 70er Jahre über die Niederlande nach Europa und in die BRD.

Seit den ersten Tagen von FORTH arbeiten verschiedene Organisationen an der ständigen Weiterentwicklung dieser Entwicklungsumgebung. Eine dieser Organisationen ist die Deutsche Forthgesellschaft e.V., die es sich zum Ziel gemacht hat, FORTH gerade dem privaten Anwender näher zu bringen und auch Menschen aus dem Hobbybereich echte Kontrolle über Maschine und System zu ermöglichen.

Leistungsübersicht verschiedener PDS's

FORTH ist heute längst keine Sprache mehr die ihr Hauptgewicht auf den Bereich Steuerung und Regelung legt. Grundsätzlich sind aber alle Forthumgebungen zunächst in ihren Leistungsumfängen eher bescheiden. Diese 'Bescheidenheit bezieht sich vor allem auf den Umfang, bzw. auf die Menge an freiem Hauptspeicher die von FORTH be-

nötigt wird ! Dort, wo andere Umgebungen zum Teil MEGABYTE an RAM für sich selbst verbrauchen, arbeitet FORTH in einigen Fällen bereits mit weniger als 10 KILOBYTE ! Trotzdem sind moderne Forthsysteme extrem leistungsfähig. FORTH wird fast immer zunächst in Form eines 'Kerns' angeboten, der elementare Grundfunktionen der Sprache anbietet. Dieser Kern ist vom Benutzer beliebig erweiterbar, so daß letztlich keine nur denkbare Funktion anderer Umgebungen fehlt. So existieren 'FORTH-Aufsätze' zu Teilgebieten wie Tabellenkalkulation, Datenbankanwendungen etc.. Diese Aufsätze belasten den Computer aber immer nur dann, wenn der Benutzer das WILL und diese 'Aufsätze' aufruft !

Voraussetzungen

Entsprechend der großen Flexibilität von FORTH, sind die maschinellen Voraussetzungen sehr gering. Auf PC-Maschinen arbeiten viele FORTH-Systeme bereits dann, wenn weniger als 128 KByte Gesamtspeicher zu Verfügung stehen. Maschinen älterer Bauart, z.B. CP/M Rechner, 'fahren' noch wesentlich kleinere FORTHversionen.

Computer- im Unterricht und zu Hause

Die Forthgruppe Moers wird sich bemühen, zu den Kurstagen in ausreichender Zahl Rechner zu Demonstrations- und Übungszwecken zur Verfügung zu stellen. Trotzdem wird es den einzelnen Kursteilnehmern einfacher sein, dem Unterricht zu folgen, wenn sie über eigene Rechner verfügen und den Inhalt der jeweiligen Stunden zu Hause nachvollziehen können.

ZF-FORTH Compiler

Im Einsteigerkurs arbeitet die Gruppe mit dem Shareware Compiler ZF von Tom Zimmer. Dabei handelt es sich um einen sehr komfortablen Forthcompiler, der auf den F83-Standard aufbaut. ZF verfügt über einen sequentiellen Editor zur Erfassung und Bearbeitung der Quelltexte, über die Möglichkeit 'Langadressen' durch den gesamten RAM-Bereich anzusprechen, über Routinen zur Bearbeitung von Dateien und viele weitere Optionen. ZF ist, als Sharewareprodukt, frei benutzbar. Auch die Kursteilnehmer sind völlig frei darin, den Compiler oder Kopien des Compilers weiter zu geben.

Gedrucktes Material

'Gedrucktes' Material stellt die Forthgruppe Moers den Kursteilnehmern in Form von Fotokopien zur Verfügung. Dabei wird es sich um Fotokopien zu Texten der Lerninhalte handeln.

2. Kurstag

Was ist FORTH ?

Wie am ersten Kurstag bereits angesprochen, wird FORTH häufig zu den 'Exoten' unter den Programmiersprachen gezählt. Vor allem die große Nähe zur Hardware schreckt viele 'Hobbyisten', aber auch viele 'professionelle' Programmierer zunächst ab. In einer Zeit in der große Softwarehäuser Entwicklungspakete von zum Teil Megabyte Umfang anbieten, scheint es so, als gäben diese Pakete Maßstäbe vor, an denen alle anderen Entwicklungsumgebungen gemessen werden können. Diese 'großen' Sprachumgebungen führen ihren Anwender aber sehr weit von der eigentlichen Maschine fort und erlauben das bloße 'Benutzen' solcher Sprachen, ohne daß der Anwender zu wirklichem Verständnis von Maschine und System gelangen kann.

FORTH sollte von Anfang an keine 'Blackbox' sein, sondern seinem Benutzer deutlich machen, was der jeweilige Computer physikalisch leisten kann und was nicht. Durch die Arbeit mit FORTH wird auch dem Hobbyprogrammierer klar, daß Computer sehr universelle Werkzeuge darstellen, deren Arbeitsbereiche grundsätzlich nur von Geschick und Phantasie des Programmierers eingeschränkt werden. Wo andere Sprachen an künstliche Grenzen stoßen, beginnt Forth erst interessant zu werden. Ob eine Applikation aus dem Bereich 'Messen-Steuern-Regeln' geplant ist, oder eine Datenbankanwendung, ob es um graphische Aufbereitung von Daten geht, oder um ein Programm zum Musizieren - Forth kann alles, zumindest alles, was sein Programmierer kann, und was der Computer physikalisch 'hergibt'.

FORTH - eine interaktive Stackmaschine

FORTH wird oft als 'interaktive Stackmaschine' bezeichnet. Allein die Bezeichnung Maschine deutet bereits darauf hin, daß Forth etwas 'anders' ist als andere Sprachen. Tatsächlich verhält Forth sich zum Teil so, als sei es ein eigener Computer. Diese 'virtuelle Maschine' existiert als Software innerhalb der physikalischen Maschine. Was zumindest für den Einsteiger nicht so leicht einsichtig ist, wird durch den Umgang mit Forth meist sehr schnell deutlich. Forth benutzt bestimmte Teile der Hardware auf eine nur ihm eigene Art und 'reizt' die Hardware technisch aus. Der Begriff der 'Interaktivität' ist eine wesentliche Forderung an eine leistungsfähige, moderne Programmierumgebung. Interaktivität ist nur wenigen Sprachumgebungen zu eigen und hauptsächlich von BASIC her bekannt. Interaktivität bedeutet im grunde nichts anderes, als daß der Programmierer einen großen Teil, der ihm zu Verfügung stehenden Worte und Befehle, quasi von der 'Oberfläche' der Umgebung aus aufrufen kann, ohne diese Befehle erst in ein Programm packen zu müssen. So kann man FORTH zum Beispiel rechnen lassen, gerade so, als bediene man einen überdimensionierten Taschenrechner. Das ist sonst nur noch mit BASIC möglich. Andere Compiler als FORTH können so etwas nicht. Die Interaktivität bringt einen weiteren Vorteil mit sich, der vor allem von professionellen Programmierern sehr hoch bewertet wird - Die Suche nach Fehlern innerhalb eines Programmes wird sehr einfach. FORTH stellt seinen Anwendern eine ganze Reihe

von Werkzeugen zur Verfügung, die es in anderen Sprachen nicht gibt.

FORTH - ein hochwertiger Compiler

FORTH ist aber nicht nur ein Interpreter. Interpreter arbeiten grundsätzlich relativ langsam. Interpreter wie zum Beispiel BASIC arbeiten sich durch jede Programmzeile einzeln hindurch - und übersetzen diese Zeilen in für den Computer ausführbaren Maschinencode. Diese Übersetzung kostet Zeit. Wenn nun innerhalb eines Programmes eine sogenannte Schleife abgearbeitet wird, übersetzt der Interpreter die in der Schleife enthaltenen Zeilen immer wieder, so lange bis er die Programmschleife verläßt. Compiler übersetzen den Programmtext natürlich ebenfalls, aber nicht simultan. Ein Compiler übersetzt den gesamten Programmtext vor der ersten Ausführung des Programmes. Wenn ein Compilerprogramm aufgerufen wird, entfällt die Übersetzungsarbeit, die ja bereits zuvor erledigt wurde. Das Programm läuft wesentlich schneller als unter einem Interpreter.

FORTH kompiliert alle seine 'Quelltexte'. Der Compiler in FORTH ist relativ schnell. Das bedeutet, daß sowohl die Übersetzungsarbeit schnell vonstatten geht, als auch, daß der dabei erzeugte Code schnell abläuft. Gegenüber den jüngsten Versionen der 'C' und 'PASCAL'-Compiler ist FORTH etwas langsamer im ausführbaren Code. Diese Einbuße an 'Laufzeit' macht aber der ebenfalls in Forth enthaltene Assembler mehr als wett.

FORTH - Assembler

Typischerweise verfügen Forth-Systeme über einen eigenen Assembler. Dabei handelt es sich, von Rechner zu Rechner unterschiedlich, um einen Assembler für den jeweiligen Prozessor des Computers, in dem Forth arbeitet. Mit diesen Assemblern lassen sich die Prozessoren 'direkt' ansprechen. Das führt zu Maschinencode der mit der maximal möglichen Arbeitsgeschwindigkeit im Computer ausgeführt wird. Die Arbeit mit einem Assembler ist meist etwas unbequem. Forthprogrammierer benutzen den 'eingebauten' Assembler deshalb grundsätzlich nur dann, wenn sie Definitionen brauchen, die möglichst schnell ablaufen sollen. Dies ist relativ selten der Fall. Die allermeisten Definitionen sind nicht 'zeitrelevant'. Der besondere 'Clou' bei der Arbeit mit dem Forth-Assembler ist der, daß der Programmierer hier immer in der gleichen 'Umgebung'arbeitet. Ganz gleich, ob er interaktiv, mit dem Compiler oder mit dem Assembler arbeitet, der Programmierer braucht FORTH nie zu verlassen. In anderen Sprachen fehlen solche Optionen ganz. Dort muß der Programmierer sich an jede der drei grundsätzlichen Arbeitsumgebungen jeweils neu anpassen.

ZF - Ein vollständiges Entwicklungssystem

FORTH ist ein sogenanntes PDS, ein Program Develop System, ein Entwicklungssystem für Programme. Als Entwicklungssystem ist FORTH vollständig, eben eine 'Zusammenfassung' aus Interpreter, Compiler und Assembler. Dazu kommen die bereits erwähnten Werkzeuge zur Fehlersuche und Fehlerbehandlung, die in den meisten anderen Sprachen, wenn überhaupt, als externe Programme zu Verfügung stehen und in FORTH einen festen Bestandteil der Sprache darstellen. Trotz dieser komplexen Umgebung ist FORTH sehr klein. Wo andere Umgebungen mit geringerem Leistungsumfang bereits mit einem Megabyte an Programmen und Werkzeugen aufwarten, ist ein typisches FORTH-System immer noch kleiner als 50 KByte.

Der Editor in ZF

Zu einem vollständigen System gehört immer auch ein Editor. Der Editor ist nichts weiter als eine Textverarbeitung. Diese Textverarbeitung dient dazu, die Quelltexte der Programme zu erfassen. Mit einem Editor schreibt der Programmierer sein Programm. Wenn sich während des Kompilierens, was meistens der Fall ist, herausstellt, daß man zum Beispiel ein Wort falsch geschrieben hat, kann dieses Wort mit Hilfe des Editors einfach und bequem korrigiert werden, ohne daß andere Teile des Quelltextes davon berührt werden.

Das ZF-Forth besitzt, wie alle anderen Forthsysteme ebenfalls, einen 'eingebauten' Editor. Das bedeutet, daß der Programmierer zur Bearbeitung seiner Quelltexte kein externes Programm zur Textverarbeitung benötigt. Das ist auch heute noch keineswegs selbstverständlich. FORTH war, schon vor mehr als 20 Jahren, die erste Programmierumgebung die damit aufwarten konnte. Der Editor in ZF stellt in sofern eine Besonderheit dar, als es sich dabei um einen 'sequentiellen' Editor handelt. Fast alle anderen FORTH-Systeme verfolgen dagegen ein sogenanntes BLOCK- oder SCREENKonzept. Blockorientierte Editoren bieten dem Programmierer eine Reihe ganz eigener Vorteile, sind aber zumindest für den Anfänger mit einigem Lernaufwand verbunden. Der sequentielle Editor des ZF ist wesentlich einfacher und schneller beherrschbar als Block-Editoren und eignet sich darüber hinaus als 'ganz normale' Textverarbeitung. Um mit dem Editor arbeiten zu können, brauchen wir uns nur wenige Worte zu merken.

10. Kurstag

Werkzeuge in FORTH

Nachdem wir bereits eine ganze Menge an verschiedenen Worten, oder Befehlen in FORTH kennen gelernt haben, ist es an der Zeit ein wenig über 'Werkzeuge' zum Programmieren zu erfahren.

Bei den verschiedenen Übungen, Beispielen und Test's haben wir sicher schon gemerkt, daß nicht immer alles genau so funktioniert, wie wir uns das jeweils vorgestellt haben. Schreibfehler in den Quelltexten zu unseren Definitionen, Irrtümer in bezug auf den logischen Ablauf eines Wortes und viele andere Gründe können dazu führen, daß ein von uns definiertes Wort etwas anderes tut, als wir das wollen. Das

ist etwas ganz und gar 'Normales' und 'Natürliches'. Auch der Werkzeugmacher oder Maschinenbauer oder... bekommt sein jeweiliges Werkstück nicht immer auf Anhieb genau so hin, wie er es ursprünglich wollte. Das ist bei den Programmierern nicht anders. Anders ist hier allerdings die Arbeit bei der Suche nach Fehlern. Eine Motorwelle die unter Belastung 'ausleiert', kann man beobachten, vermessen und ersetzen. Bei falsch funktionierenden Definitionen im Computer kann man bestenfalls an einem unwahrscheinlichen Ergebnis erkennen, daß dort etwas aus dem Lot läuft. Wir können ja die 'Bitströme' weder sehen, noch anfassen noch irgendwie manipulieren, oder ? DOCH ! Wir können das durchaus, und wir können durchaus an einem Wort arbeiten wie ein Handwerker an einem Werkstück ! Dazu benötigen wir nur die richtigen Werkzeuge. Anders als die meisten anderen Sprachen stellt FORTH uns alles was wir hier brauchen auch zur Verfügung ! Was aber besonders angenehm an den Werkzeugen in FORTH ist, ist die Tatsache, daß wir, um mit diesen Werkzeugen zu arbeiten, die FORTH-Umgebung nicht zu verlassen brauchen ! Die meisten anderen Sprachen stellen diese Werkzeuge ausschließlich als 'externe' Programme zu Verfügung. In FORTH sind die folgenden Werkzeuge direkt eingebaut.

Dabei handelt es sich gerade um vier Worte, die wir jetzt lernen müssen. DEBUG, DUMP, VIEW und SEE heißen diese Worte, und wir werden in Zukunft recht häufig von Ihnen Gebrauch machen.

Einer der häufigsten Fehler bei der Arbeit mit einer 'Stackmaschine' wie FORTH ist der, daß man aus irgendwelchen Gründen die Übersicht über das Geschehen auf dem Stack verliert. Da braucht weder Nachlässigkeit im Spiel zu sein, noch Unkenntnis. Manchmal ist man sich einfach nicht darüber klar, was ein Wort vom Stack herunter nimmt, oder welche Werte es dort ablegt. Das beobachten zu können, ermöglicht uns DEBUG.

Mit dem Aufruf DEBUG <name> bereiten wir ein Wort für den Einzelschrittmodus vor. Das bedeutet, daß wir eine Definition anschließend Schritt für Schritt ausführen lassen können. Schritt für Schritt meint dabei, daß jedes FORTH-Wort, wie wir inzwischen wissen, aus anderen FORTH-Worten besteht. Wir können jedes dieser untergeordneten Worte einzeln ausführen lassen und dabei sehen, was sich auf dem Stack tut. DEBUG, das dabei einen eigenen Interpreter darstellt, hält die Arbeit nach jedem Wort an und gibt uns einen Überblick über den Inhalt des Stack. Dabei macht DEBUG die Fehlersuche sehr komfortabel für uns. DEBUG teilt den Bildschirm in zwei Bereiche auf. Im oberen Bereich bekommen wir die 'Quelle' angezeigt, den Text der unserer zu überprüfenden Definition zu grunde liegt. Im unteren 'Fenster' zeigt DEBUG uns jeweils das nächste auszuführende Wort, sowie den aktuellen Stand auf dem Stack an. Ein Druck auf eine (fast beliebige) Taste führt uns jeweils einen Schritt weiter - bis zum Ende der gesamten Definition. Wir können sehen, wie sich die Definition zu jedem Zeitpunkt ihrer Ausführung verhält.

Dieses Wort gehört zum Sprachumfang des F83-Standard. Leider bieten nicht alle FORTH-Versionen ein Werkzeug wie DEBUG. Zum Beispiel die ansonsten recht gute 'professionelle' Version des PC-FORTH von LMI kennt keinen solchen 'Fehlersuchinterpreter'. Dafür leistet das DEBUG 'unseres' ZF-Systems umso mehr.

Zwischen den beiden Bildschirmbereichen stellt DEBUG uns in einer Zeile ein paar zusätzliche Optionen zu Verfügung. Wenn wir die Taste (C)ontinue drücken, wird der Einzelschrittmodus teilweise aufgehoben. Die einzelnen Schritte 'flitzen' über das untere Ausgabefenster. Wir können diesen 'Schnellauf' jederzeit durch Drücken einer beliebigen Taste unterbrechen und so wesentlich schneller innerhalb einer umfangreicheren Definition an den vermutlichen Fehlerort gelangen. (Z)ip arbeit ähnlich wie (C)ontinue und führt zum Beispiel eine in dem Wort enthaltene Schleife 'am Stück' aus. Am Wichtigsten ist aber die Option (F)orth. Durch Drücken des Buchstabens 'F' erreichen wir, daß DEBUG seine Arbeit kurz unterbricht und uns wieder auf die FORTH-Oberfläche zurück läßt. Hier können wir jetzt zum Beispiel die Inhalte von Variablen 'von Hand' ändern, oder den Stack 'von Hand' manipulieren. Wenn wir anschließend zweimal die <enter> Taste drücken, gelangen wir wieder in den Interpretermodus von DEBUG zurück.

Um ein von DEBUG gekennzeichntes Wort wieder zu 'normalisieren' steht uns der Befehl UNBUG zur Verfügung. UNBUG erwartet keinerlei Parameter.

Ebenso häufig wie 'Stackfehler', sind Fehler, die auf Unklarheiten über Speicherinhalte beruhen. Um solche Fehler aufspüren zu können, brauchen wir ein Wort, mit dessen Hilfe wir praktisch in jede beliebige Speicherzelle unseres Computers 'hineinsehen' können. Dies leistet das Wort DUMP !

DUMP kippt uns den Inhalt von Speicherstellen gewissermaßen auf den Bildschirm aus. Selbstverständlich hält DUMP dabei ein für uns lesbares Format ein, und ebenso selbstverständlich tastet DUMP die Inhalte dabei nicht an. Wir wollen ja noch nichts verändern, sondern uns lediglich informieren.

Stellen wir uns einmal vor, wir hätten folgendes Array definiert:

```
CREATE feld 10 ALLOT
```

Über den Inhalt dieses Feldes aus 10 Bytes sind wir uns nicht im Klaren. Hier hilft die Anweisung

```
feld 10 DUMP
```

sofort weiter. Wir bekommen ungefähr folgende Ausgabe auf den Bildschirm: (siehe Listing 1)

Was macht DUMP nun genau ? Zunächst haben wir durch den Aufruf von <feld> die Startadresse des Array's auf den Stack gelegt. Die

Aufforderung 10 DUMP besagt, daß wir mindestens die ab <feld> folgenden nächsten 10 Byte angezeigt bekommen möchten. DUMP arbeitet aber generell mit einem Ausgabeformat, das immer 16 Bytes auf einmal anzeigt. Das erste Byte, in dem von uns geforderten Bereich ist durch das Zeichen 'V' gekennzeichnet. Unter diesem Zeichen lesen wir die Zahl 05. Dieser 1-Byte große Speicherplatz enthält das Bitmuster der Zahl 5. Das können wir noch 'direkt' auslesen. Bei dem folgenden Byte wird es etwas komplizierter. DUMP arbeit nämlich grundsätzlich mit hexadezimalen Zahlen. Die '41', die wir dort lesen, entspricht also dem Bitmuster der HEX-Zahl 41, oder der dezimalen Zahl '65' oder dem ASCII-Zeichen 'A'. Wir müssen bei der Arbeit mit DUMP daran denken, daß wir nur hexadezimale Zeichen angezeigt bekommen. Was diese Werte im Einzelnen bedeuten sollen, daß müssen wir selbst wissen. Die Bedeutung der Bitmuster sind ja ausschließlich von unserer Defintion anhängig, bzw. davon, ob wir einen solchen Speicherplatz z.B. von EMIT oder von '.' bearbeiten lassen.

Vor der Zeile mit den 'Bitmustern' sehen wir in diesem Beispiel die Zahl 8304 stehen. Das ist die HEXADEZIMALE (!) Startadresse des Array's. Dieser Wert wird bei jedem FORTH-System anders aussehen und ist von vielen unterschiedlichen Faktoren abhängig. Diese Angabe ist im Augenblick aber auch nicht von großer Bedeutung. Wichtiger ist die Ausgabe neben der 'Hex-Zeile'. Dort sehen wir 'richtigen' Text. DUMP versucht, die hexadezimalen Werte sofort in ASCII-Zeichen zu übersetzen. Wo das nicht gelingt, schreibt DUMP einfach einen Punkt, oder läßt ein Leerzeichen.

Mit DUMP können wir uns also, bei etwas Übung, sehr genau über die Inhalte jedes einzelnen Speicherplatzes informieren. Wenn wir allerdings größere, zusammenhängende Speicherplätze durchsehen möchten, ist die Eingabe von 'nn' DUMP etwas unbequem. Deshalb stellt ZF hierzu eine etwas komfortablere Version zur Verfügung.

feld DU

... legt die Adresse von FELD auf den Stack und gibt uns sofort drei der obigen Zeilen aus. Dabei zählt DU gleich 3*16 auf die Startadresse auf und behält diesen Wert auf dem TOS. Wenn wir nun die nächsten drei Zeilen sehen wollen, genügt ein erneuter Aufruf von DU !

Abgucken muß nicht unbedingt etwas mit plagiatieren zu haben und ist auch nicht unbedingt verwerflich. Im Gegenteil, ein intelligenter Mensch versucht eben nicht das Rad immer wieder neu zu erfinden. Gerade FORTH-Programmierer wissen dies und geben ihr Wissen und ihre Erfahrungen gerne weiter. Das findet seinen Niederschlag in dem Wort VIEW.

VIEW will uns die 'Quellen' einer beliebigen Definition zeigen. Der Aufruf

VIEW <name>

macht uns den Quelltext der Definition <name> sichtbar. Allerdings ist hierbei Voraussetzung, daß sich die Quelldatei, in der <name> definiert wurde, in einem Verzeichnis befindet, auf das ZF zugreifen kann. Ist dies nicht der Fall, meldet VIEW uns, daß es das entsprechende File im aktuellen Pfad nicht finden kann. Gerade VIEW ist ein besonders angenehmes Werkzeug bei der Arbeit mit FORTH. Wir können uns mit VIEW jede von Tom Zimmer oder anderen Experten geschriebene Definition im Detail ansehen und dabei erleben, 'wie die Experten das machen'.

Ebenso hilft VIEW aber auch dabei, den Aufbau einer Definition zu verstehen und damit zu erkennen, warum der Gebrauch eines Wortes in einer von unserern eigenen Definitionen vielleicht nicht genau das macht, was wir eigentlich wollten....

Dazu bietet ZF-FORTH aber noch ein weiteres Werkzeug an, den 'Diskompiler' SEE. SEE arbeit unabhängig davon, ob der Quelltext einer Definition zu Verfügung steht. Der Aufruf

SEE <name>

dekompiliert die Definition <name>. Das bedeutet, daß SEE den inneren Aufbau von <name> für uns auf dem Bildschirm sichtbar macht. Anders als bei VIEW bekommen wir hier natürlich keine Kommentare im Quelltext mehr angezeigt.

Den Umgang mit diesen Werkzeugen sollten wir ausgiebig üben. Wir können uns dazu zunächst einfache Definitionen schreiben, wie z.B.

: test 10 0 DO I 10 * . LOOP ;

und diese Definitionen unter DEBUG abarbeiten lassen. Mit SEE können wir uns die 'Struktur' von 'test' ansehen. Vor allem bei später komplizierteren Definitionen werden uns die Werkzeuge eine unentbehrliche Hilfe sein. Kaum eine Definition ist wirklich von Anfang an ohne Fehler. Fehler der verschiedensten Art zu suchen und zu beheben, macht einen Großteil der Programmierarbeit aus. Mit FORTH's Werkzeugen läßt diese Arbeit sich auf ein erträgliches Maß reduzieren.

Listing 1:

```
        V  5  6  7  8  9  A  B  C
D  E  F  0  1  2  3  456789ABC-
DEF0123
  8304  05 41 4C 4C 4F 54 20 20 4D
50  04  44  55  4D  50  20  |.ALLOT
MP.DUMP |
```

FORTH für Einsteiger

Kurs der FORTH-GRUPPE-MOERS

Leitung:
Friederich Prinz,
Hombergerstraße 335,
4130 Moers 1

Inhaltsverzeichnis

1. Kurstag

Vorstellung, allgemeine Einführung, Ausrichter des Kurses: FORTH-GRUPPE MOERS / Deutsche Forth-Gesellschaft/ DPWV, Lernziel, Einstieg in die Programmiersprache FORTH, Warum FORTH ?, Geschichtliches, Leistungsübersicht verschiedener PDS's, Voraussetzungen, Computer- im Unterricht und zu Hause, ZF-FORTH Compiler

2. Kurstag

Was ist FORTH ?, FORTH - eine interaktive Stackmaschine, FORTH - ein hochwertiger Compiler, FORTH - Assembler, ZF - Ein vollständiges Entwicklungssystem, Der Editor in ZF

3. Kurstag

UPN - 'Grammatik' in FORTH, Beispiele und Übungen

4. Kurstag

Der STACK, das zentrale Datenelement, Parameterstack, Returnstack, Stackmanipulatoren, Stackbilanzen, Stack-Reportlisten

5. Kurstag

Vergleiche und Entscheidungen, 'Flaggen' erzeugen, 'Flaggen' auswerten, Vergleichsworte, Entscheidungsstrukturen, Beispiele zu IF ELSE und THEN

6. Kurstag

Wiederholte Anweisungen, Zählschleifen mit DO .. LOOP, BEGIN .. AGAIN, BEGIN .. UNTIL, BEGIN .. WHILE .. REPEAT, Schleifen beenden, LEAVE, EXIT, ABORT, ABORT"

7. Kurstag

Konstanten, Variable und Felder, Was braucht man wann ?

8. Kurstag

Der Zahlenbereich in FORTH, Integerrechnung, 'Scalieren', 16-Bit Befehle, 32-Bit Befehle, 'gemischte' Befehle, Das Wort 'SQR'

9. Kurstag

Logische Operationen, AND, OR, XOR, NOT

10. Kurstag

Werkzeuge in ZF - FORTH, DEBUG, DUMP, VIEW, SEE, DIS

11. Kurstag

Programmiertechniken, Struktur ist alles, Modulares Programmieren

12. Kurstag

Das Vokabular für den Anfang, Arithmetik, Zeicheneingabe, Zeichenausgabe, Vergleiche, Speicherinhalte, Speicherzugriffe, Zahlen-Konvertierung, Weitere Worte

13. Kurstag

Das vollständige ZF - Vokabular

14. Kurstag

Programmbeispiele, Programme unter ZF

15. Kurstag

Eigenständige Programme

16. Kurstag

Übungen und Aufgaben

17. Kurstag

Nachbereitung und Ausblick

18. Kurstag

- zur freien Verfügung -

19. Kurstag

Abschluß

-- FORTH-Gruppen --

FORTH-Gruppen:

W-1000 Berlin
Claus Vogt
Tel. 030/2 16 89 38
Treffen nach Absprache

W-4130 Moers 1
Friederich Prinz
Tel. 02841/5 83 98
Treffen jeden Samstag
14 Uhr, Moerser Arbeits-
losenzentrum, Donaustr.1

Gruppe Rhein-Ruhr:
Jörg Plewe
Tel. 0208/42 35 14
W-4000 Düsseldorf
Gebäude des S-Bahnhof
Derendorf,
Münsterstraße 199
Treffen jeden ersten
Sonnabend im Monat

W-6800 Mannheim
Thomas Prinz
Tel. 06271/28 30
Ewald Rieger
Tel. 06239/86 32
Treffen jeden ersten
Mittwoch im Monat im
Vereinslokal desSegel-
vereins Mannheim e.V.
Flugplatz, Mannheim-
Neuostheim

W-7000 Stuttgart
Wolf-Helge Neumann
Tel. 0711/88 26 38
Treffen nach Abspache

O-Leipzig
FORTH-Gruppe Leipzig:
Michael Balig, Lützner
Plan 17
O-7033 Leipzig
Dr. Jürgen Hesse
Tel. 041/69 56 02
Liselotte-Herrman-Str. 40
O-7050 Leipzig

W-5100 Aachen
Arndt Klingelnberg
Tel. 02404-61648
Treffen jeden ersten
Montag als Gruppe des
Computer-Club
der RWTH, Seminarge-
bäude Raum 214

FORTH für Ratsuchende:

Jörg Staben
Tel. 02103/5 56 09
dienstags und freitags
von 20.00-22.00 Uhr

Frank Stüss
Tel. 06187/9 15 03

Karl Schroer
Tel. 02845/2 89 51

Andreas Findewirth
Tel. 05221/2 35 04

Andreas Jennen
W-1000 Berlin, UUCP

Jörg Plewe
Tel. 0208/42 35 14

FORTH Fachgruppen:

W-6800 Mannheim
FIS (FORTH Integriertes
System) - Datenbank,
Textverarbeitung,
Kalkulation
Postadresse:
Dr. med.
Elemer Teshmar
Danziger Baumgang 97
W-6800 Mannheim 31

FORTH Interessengebiete:

volksFORTH/ultraFORTH
Klaus Kohl
Tel. 08233/3 05 24
Klaus Schleisiek-Kern

Künstliche Intelligenz
Ulrich Hoffmann
Tel. 0431/80 12 14

NC4000 Novix Chip
Klaus Schleisiek-Kern
Tel. 040/2 20 25 39

Relationale Netze
HS/Forth
Künstliche Intelligenz
Realtime
Wigand Gawenda
Tel. 040/44 69 41

Gleitkomma-Arithmetik
Andreas Döring
Tel. 0721/59 39 35

32FORTH
Rainer Aumiller
Tel. 089/6 70 83 55

PostScript/FORTHscript
Christoph Krinninger
Tel. 089/ 7 25 93 82

FORTH im Unterricht
Rolf Kretschmar
Tel. 02401/43 90

Objekt-orientiertes FORTH
Christoph Krinninger
Tel. 089/7 25 93 82
Ulrich Hoffmann
Tel. 0431/67 88 50

F-PC Zimmer FORTH
ASYST (Meßtechnik)
embedded controller
(H8/5xx == TDS2020 fig)
(8051 ... e FORTH ...)
(6511-MiniBee RSCforth)
Arndt Klingelnberg
Tel. 02404/6 16 48

FORTH Fachgruppengründung:

Grafik, Arithmetik
W-7000 Stuttgart 80
Jörg Tomes
Tel. 07 11/7 80 22 93
nur am Wochenende

FORTH Gruppengründung:

W-3300 Braunschweig
Martin Holzapfel
Tel. 05 31/35 12 62
32-Bit Systeme

W-2000 Hamburg
Wigand Gawenda
Tel. 040/44 69 41
Treffen jeden ersten
Dienstag im Monat
Themen und Treffen
nach Absprache

FORTH-MAILBOX

SYSOP

Jens Wilke

Tel. 089/8 71 45 48
300-2400 baud
Parameter 8N1

(Leere Seite)