



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

In dieser Ausgabe:



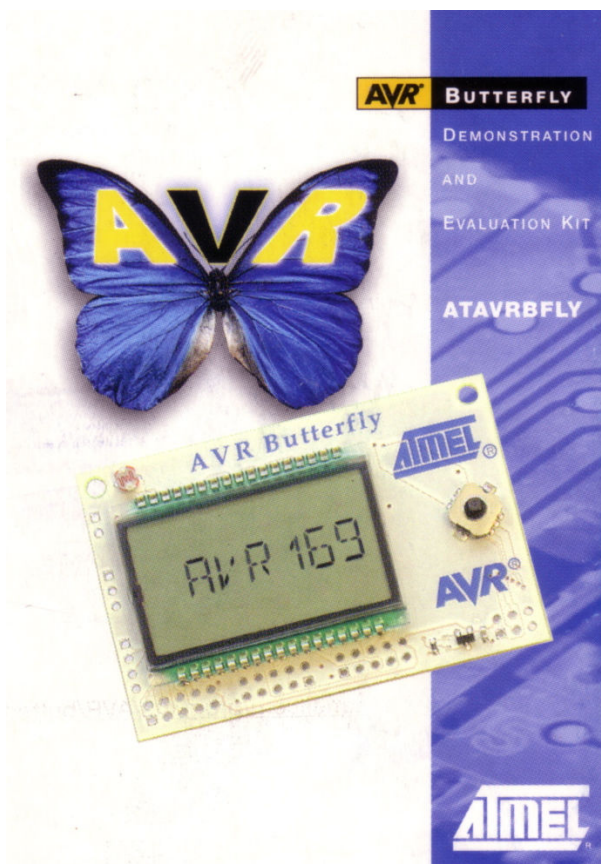
ByteForth für den AVR

USBasp-Programmierer

Forth von der Pike auf

amforth für Atmel AVR-ATmega

AVR-Butterfly Piezo-Summer



tematik GmbH Technische Informatik

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

Gegründet 1985 als Partnerinstitut der FH-Wedel beschäftigten wir uns in den letzten Jahren vorwiegend mit Industrieelektronik und Präzisionsmeßtechnik und bauen z. Z. eine eigene Produktpalette auf.

Know-How Schwerpunkte liegen in den Bereichen Industriewaagen SWA & SWW, Differential-Dosierwaagen, DMS-Messverstärker, 68000 und 68HC11 Prozessoren, Sigma-Delta A/D. Wir programmieren in Pascal, C und Forth auf SwiftX86k und seit kurzem mit Holon11 und MPE IRTC für Amtel AVR.

LEGO RCX-Verleih

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

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

Inhalt:

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

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

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

RetroForth

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

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

Hier könnte Ihre Anzeige stehen!

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

Secretary@forth-ev.de

KIMA Echtzeitsysteme GmbH

Tel.: 02461/690-380
Fax: 02461/690-387 oder -100
Karl-Heinz-Beckurts-Str. 13
52428 Jülich

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

FORTECH Software

Entwicklungsbüro Dr.-Ing. Egmont Woitzel

Budapester Straße 80 a D-18057 Rostock
Tel.: (0381) 46 13 99 10 Fax: (0381) 4 58 34 88

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

Hier könnte Ihre Anzeige stehen!

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

Secretary@forth-ev.de

Ingenieurbüro

Klaus Kohl-Schöpe

Tel.: 07044/908789
Buchenweg 11
D-71299 Wimsheim

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

Impressum	4
Editorial	4
ByteForth für den AVR	5
<i>Willem Ouwerkerk</i>		
USBasp-Programmierer	10
<i>Ulrich Hoffmann</i>		
Forth von der Pike auf	11
<i>Ron Minke</i>		
amforth für Atmel AVR-ATmega	28
<i>Matthias Trute</i>		
AVR-Butterfly Piezo-Summer	34
<i>Ulrich Hoffmann, Michael Kalus</i>		

Impressum

Name der Zeitschrift Vierte Dimension

Herausgeberin

Forth-Gesellschaft e. V.
Postfach 19 02 25
80602 München
Tel: (0 89) 1 23 47 84
E-Mail: Secretary@forth-ev.de
Direktorium@forth-ev.de
Bankverbindung: Postbank Hamburg
BLZ 200 100 20
Kto 563 211 208
IBAN: DE60 2001 0020 0563 2112 08
BIC: PBNKDEFF

Redaktion & Layout

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

Anzeigenverwaltung

Büro der Herausgeberin

Redaktionsschluss

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

Erscheinungsweise

1 Ausgabe / Quartal

Einzelpreis

4,00€ + Porto u. Verpackung

Manuskripte und Rechte

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

Liebe Leser,

Forth konnte seine Stärken schon immer in rauen Umfeldern zeigen — dort, wo die Ressourcen stark eingeschränkt sind, wo wenig Speicher zur Verfügung steht oder die Prozessoren langsam arbeiten, etwa weil höchstes Energiesparen gefordert ist.

Forth erreicht das durch seinen extrem kleinen aber bereits schon funktionalen Kern, seine geniale Einfachheit und indem es sich vollkommen für die jeweilige Aufgabe spezialisiert und dabei gegebenenfalls allen unnötigen Ballast über Bord wirft.

Das hat auch Nachteile. Ein geflügeltes Wort sagt *Wenn Du ein Forth gesehen hast, hast Du ein Forth gesehen*. Zwei verschiedene Forth-Systeme unterscheiden sich auf Grund der unterschiedlichen Anforderungen möglicherweise nur in ganz subtilen Kleinigkeiten, die aber letztlich verhindern, dass Programme des einen Systems sich ohne Weiteres im anderen System verwenden lassen.

Im vorliegenden AVR-Sonderheft legen wir unser besonderes Augenmerk auf die AVR-Mikrocontroller der Firma Atmel [1], die in jüngster Zeit größte Beliebtheit erreicht haben. Der interessierte Einsteiger kann für unter 50 Euro AVR-Experimentier-Hardware (wie etwa das auf der Titelseite abgebildete AVR-Butterfly-Board) erstehen, die sich auch schon für einfache Anwendungen eignet. Atmel stellt seine Windows-basierte Entwicklungssoftware kostenfrei zur Verfügung. Zahlreiche (teils Open-Source-)Projekte arbeiten zum einen an günstigen Programmiergeräten als auch an Entwicklungssoftware, die unter Windows, Linux oder Mac OS X eingesetzt werden kann. Aber auch auf den AVR-Prozessoren tut sich eine Menge. Besonders erwähnenswert ist sicher die Realisierung eines *USB-Controller für AVR-Micros* [2] rein in Software, die Minimalsysteme mit direkter Anschlussfähigkeit an moderne PCs ermöglicht.

AVR-Prozessoren sind einfach zu programmieren. In Assembler, in C oder eben in Forth. Je nach Anforderungen kommen dabei unterschiedliche Ansätze zum Zug: Byte-Forth von Willem Ouwerkerk verwendet nur 8Bit große Stack-Zellen und ist als Cross-Compiler konzipiert. amForth von Matthias Trute ist ein klassisches, interaktives 16Bit-Forth-System, das im Zielsystem definierte Worte on-the-fly in den AVR-Flash-Speicher programmiert. Welche Fragen sich ein Forth-Programmierer stellen muss, wenn er ein Forth-System für einen Mikrocontroller realisieren will, erläutert Ron Minke in seinem Artikel *Forth von der Pike auf*. Dieser Artikel ist ursprünglich als Serie im niederländischen Forth-Magazin *Vijgeblaadje* und dann auch in Deutsch, von Fred Behringer übersetzt, in der Vierten Dimension erschienen. Unsere Leser haben ihn dort in den Umfragen als besonders beliebten Artikel ausgewählt.

Und nun — viel Spaß beim Lesen des AVR-Sonderhefts. Möge es unsere Diskussionen um den Einsatz und die Realisierung von Forth beflügeln.

Ulrich Hoffmann

[1] <http://www.atmel.com>

[2] <http://www.obdev.at/products/avrusb/index.html>

Die Quelltexte in der VD müssen Sie nicht abtippen. Sie können sie auch von der Web-Seite des Vereins herunterladen.
<http://www.forth-ev.de/filemgmt/index.php>

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

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



ByteForth

Willem Ouwerkerk

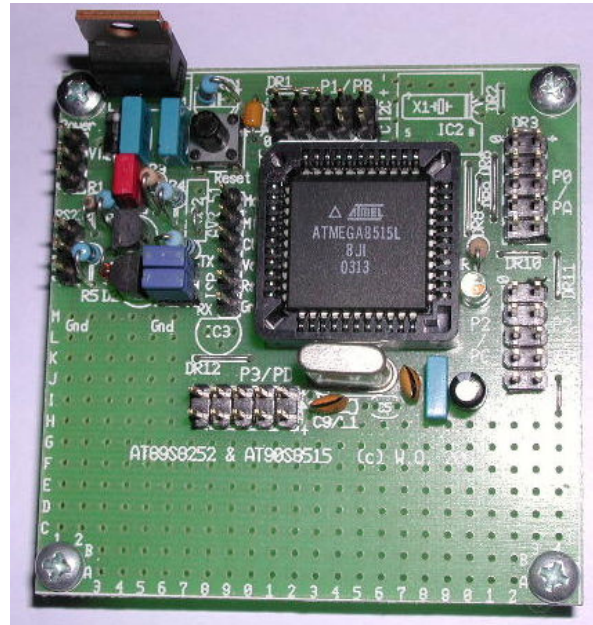
(Red.: Der Autor leitet seit vielen Jahren als Voorzitter die Geschiede der HCC-Forth-gebruikersgroep. Wir freuen uns über die stets gute Zusammenarbeit mit unseren Forth-Nachbarn im Westen. Die holländische Originalfassung wurde von Fred Behringer übersetzt.)

Was ist AVR ByteForth?

AVR-ByteForth ist ein 8-Bit-Forth-Makrocompiler unter MS-DOS, der Maschinencode für einen Mikro-Controller ohne externes ROM und RAM erzeugt. Es soll auch kleine Mikro-Controller, die zu wenig RAM für ein komplettes Forth-System haben, in die Lage versetzen, mit Forth betrieben zu werden. Das ByteForth-System liefert selbstständig arbeitenden Maschinencode ohne den bekannten Forth-Interpreter, es hat einen eingebauten Simulator und der erzeugte Code kann auf dem PC ausgetestet werden. Zum Testen der Hardware muss der Code natürlich auf dem Mikro-Controller laufen. ByteForth ist ein Alles-in-einem-Paket und enthält: einen optimisierenden Compiler, einen Assembler, einen Disassembler, einen Simulator, einen Tracer, einen Flash-ISP-Programmer und eine Online-Hilfefunktion. Vorhanden ist eine Bibliothek mit einer großen Zahl von getesteter Software, beispielsweise zur Ansteuerung von LEDs, numerische Routinen, I2C, RS232 usw. Außerdem sind viele vollständig ausgearbeitete Anwendungen vorhanden, so z.B. der Pincode-Türöffner, ein I2C-Thermometer, synchrone Servosteuerung usw. Im 'Egelwerkbuch der HCC-Forth-gg findet man eine ganze Reihe von Anwendungen für ByteForth. Die zugehörigen Dateien können auch aus dem Internet heruntergeladen werden.

Makrocompiler für ByteForth

- Basiert auf ANS-Forth
- Stackbreite: 8 Bit, doppeltgenau 16 Bit, 4-fachgenau 32-Bit.
- Arrays, sowohl für 8- als auch für 16-Bit-Zahlen.
- Einstellbare Speichereinteilung.
- Definierende Worte: CONSTANT 2CONSTANT VARIABLE 2VARIABLE VALUE : CODE CREATE DOES>.
- Nichtstandard-Erweiterungen: BIT-SFR FLAG SFR CONSTANTS VARIABLES 2VARIABLES REGISTER.
- Kontrollstrukturen: IF ELSE THEN DO LOOP BEGIN UNTIL FOR NEXT usw.
- Präfixe für Datenworte: FROM TO +TO INCR DECR CLEAR usw.
- Alle Interrupts werden unterstützt.
- Makros können als Unterprogramme eingeführt werden.
- Spezielle Compiler-Anweisungen für das Arbeiten mit Strings: ." S" [CHAR] und LITERAL.



Eine einfache Entwicklungs-Platine mit ATmega8515-Prozessor

Interaktives Testen (Software-Simulator)

Alle mit AVR-ByteForth geschriebenen Routinen können vom eingebauten Simulator durch Eintippen ihres Namens ausgeführt werden. Variablen können gelesen und beschrieben werden, (Teil)-Programme können ausgeführt werden. FLYER ist eine Routine, die die Ausführung auf dem PC zum AVR-Simulator leitet. Sie ist in der Routine VARIABLES enthalten und wird später vom internen Code aus AVR-ByteForth angepasst. Wenn neuer Code zu compilieren ist, beginne man mit EMPTY (entferne jeden eventuell noch im Programmpuffer enthaltenen Code) und den Namen des betreffenden Mikro-Controllers (z.B. TINY2313). Mit dem eingebauten Simulator kann viel Code getestet werden, die Hardware natürlich nicht.

Beispiel:

```
1 DUP .S      \ Teste die Bibliotheksroutine DUP
              \ mit der Zahl 1 auf dem Stack
( 1 1 )      \ Die Antwort von Forth
```

Das Wort NIP zu Forth hinzufügen:

```
: NIP ( x y -- y ) SWAP DROP ;
1 2 NIP .S      \ Teste NIP
( 2 )          \ Die Antwort von Forth
```

Die Routinen DUP und NIP funktionieren. Alle selbstgemachten Codeteile können auf die gleiche Art getestet werden.



Die 32 internen Register

Lade Konstante(n) in Extra-Akku	R0 oder/und R1 zur Verwendung mit dem Befehl LPM,
ByteForth-System	R1 oder R2 Highlevel-LOOP-Zähler
Bit-Variablen	Zusammen mit Register-Variablen in R2 oder R3 bis R15
Register-Variablen	Zusammen mit Bit-Variablen in R2 oder R3 bis R15
Akkus für Code-Definitionen	10 Bytes, R16 bis R25
Pointer Reg. X, Y, Z	Obere 6 Bytes R26 bis R31
	X = Daten-Stack-Pointer
	Y = Variablen-Basis-Pointer
	Z = Frei für Locals, SLITERAL
	DOES> INLINE\$ EXECUTE

I/O-Register

Wie vom Hersteller der SFR-0 bis SFR-63 oder SFR-255 definiert

Internes RAM

Reg. und I/O-Reg.	Auch auf den RAM-Adressen 0 bis 95 oder 255
Datenstack	Adresse 127 bis 96 oder 287 bis 256
Localstack	Adresse 128 oder 288 nach oben zu unter RSP
Returnstack	Mit Locals im selben RAM-Bereich von Adr. 255 oder 319 ab nach unten
Variablen	Oberhalb des Returnstacks bis RAMTOP. Maximal 64 an der Zahl.
Variablen	Arrays beginnen bei RAMTOP und wachsen nach unten auf Variablen zu
Zahlenumwandlung	Über Array im ByteForth-System

ROM

Interrupt-Vektoren	0 bis xxx (je nach AVR verschieden)
Programm	yyy bis ROMTOP

Tabelle 1: Wie ByteForth den Speicher aufteilt

Der ByteForth-Arbeitszyklus

1. Den Crosscompiler (wieder) starten.
2. Das Programm compilieren.
3. ISP-Kabel anschließen, soweit das noch nicht geschehen ist.
4. Den Flash-Speicher leeren: E.
5. Den Flash-Speicher programmieren: P.
6. Den Flash-Speicher verifizieren: V.
7. Das Programm absichern: Lock1 und Lock2.
8. und das Programm läuft.

Der interne Aufbau von ByteForth

Ich darf mit der Speichereinteilung nach dem Start von ByteForth beginnen. Davon hängen viele Entwurfsentscheidungen ab. Das Format von RAM und ROM richtet sich nach dem gewählten AVR. RAMTOP und ROMTOP bekommen daher je nach gewähltem AVR den einen oder anderen Wert. R16 und höher sind die Akkus für Codedefinitionen von ByteForth. Es sind die einzigen Register, in welche man Konstanten direkt einspeichern kann. Die Grundeinstellung wird mit Hilfe von internen Tabellen vorgenommen, welche Informationen über die Speicherausmaße des gewählten AVRs enthalten. Über die Worte MEMORY oder MAP kann diese Einteilung verschieden gewählt werden, so dass sich der AVR optimal einstellen

lässt. Die Tabelle 1 fasst zusammen, wie ByteForth die Register, RAM und ROM verwendet.

Beispiele von internem Code aus dem ByteForth-Compiler

Die Beschreibung aller unterstützten AVR-Typen ist in einer Anzahl von Tabellen festgelegt. Die erste davon enthält die Größe des Flash-Speichers. Der älteste und kleinste ist der AT90S2313, der jedoch schon wieder überholt ist. Der neueste in diesem Beispiel ist der ATmega649 mit 64 kByte Flash. ROMTOP verwendet diesen Wert.

```

$FD00 $8000 $FD00 $FD00 $FD00 $FD00 $8000
$2000 $1000 $0800 $2000 $1000 $0800 $A000
$4000 $4000 $2000 $1000 $0800 $0400 $2000
$4000 $4000 $8000 $2000 $0800 $FD00 $4000
$2000 $8000 $FD00 $4000 $4000 $2000 $2000
$0400 $0800 $0400 $2000 $2000 $1000 $1000
$1000 $0800 $0800 $0800 $0800 $0800 #AVR
AVRDATA iflash

```

Da nicht alle AVRs alle Maschinenbefehle unterstützen, habe ich die Befehlssätze in verschiedene Untergruppen aufgeteilt. Der Assembler und der Disassembler machen davon Gebrauch. Es gibt AVRs mit 90 Befehlen, die größeren Typen haben jedoch gut 131 Maschinenbefehle, worunter sich auch einige Varianten der Hardware-Multiplikation befinden.

IRTC : TST 3 I/O PORTA C! ;	ByteForth : TST 3 TO PORTA ;
Ergebnis IRTC ohne Optimierer (20 Byte Code exkl. C!) PUSHt, TOSL 3 LDI, TOSH 0 LDI, \verbPUSHT," TOSL PORTA LDI, TOSH 0 LDI, ' C! CALL, RET,	AVR-ByteForth ohne Optimierer (10 Byte Code) R16 3 LDI, -X R16 ST, R16 X+ LD, PORTA R16 OUT, RET,
Ergebnis IRTC mit Optimierer (8 Bytes) R16 3 LDI, PORTA R16 STS, RET,	AVR-ByteForth mit Optimierer (6 Bytes) R16 3 LDI, PORTA R16 OUT, RET,

Tabelle 2: IRTC und ByteForth generieren unterschiedlichen Maschinencode.

```
\ Instruction set size
#131 #131 #131 #131 #131 #131 #131 #120
#120 #120 #120 #120 #120 #131 #131 #131
#131 #131 #120 #120 #131 #131 #131 #131
#131 #118 #131 #131 #131 #131 #121 #131
#131 #128 #118 #090 #118 #090 #118 #118
#118 #118 #118 #118 #118 #118 #118 #AVR
AVRDATA iopcode
```

Nun einige Makros. Ein Makro ist ein vorassembliertes Stück Code, hauptsächlich für die wichtigsten Forth-Primärworte, wie z.B. SWAP und UM*, usw. Jedes Makro bekommt zunächst sein Stackverhalten mit. \$C1 bei DUP zeigt an, dass der Optimierer vorher Daten in R16 erwartet und danach R16 auf dem Stack ablegt. \$01 bei OVER bedeutet, dass vorher kein Standard-Optimierverhalten vorliegt und hinterher R16 auf den Stack gelegt wird. Und schließlich legt \$21 bei + fest, dass vorher R16 und R17 erwartet und nachher R16 wieder gepUSHT wird.

```
$C1 MACRO DUP ( x -- x x ) \ Makro "dupe"
  R16 X LD,
  -X R16 ST,
  RET,
END-CODE
```

```
$01 MACRO OVER ( x1 x2 -- x1 x2 x1 ) \ Makro
  XL 1 ADIW,
  R16 X LD,
  XL 1 SBIW,
  -X R16 ST,
  RET,
END-CODE
```

```
$21 MACRO + ( x1 x2 -- x3 ) \ Makro "plus"
  R16 X+ LD,
  R17 X+ LD,
  R16 R17 ADD,
```

```
-X R16 ST,
RET,
END-CODE
```

Um ByteForth noch etwas weiter zu optimieren, wurden zusätzlich zu den Standard-Optimierungsschritten noch einige Spezialfälle aufgenommen. Es wird ein bisschen Code von der/den vorausgegangenen Routine/n entfernt und dafür ein besser passendes Stück Code erzeugt. Im Folgenden wird ein Teil des Optimierers gezeigt, der die Plus-Operation weiter verbessert. Es treten hier vier Fälle auf, aber in Wirklichkeit sind das zehn Fälle. In allen Fällen wird der Code für das Literal (oder Konstante) entfernt und die zugehörigen Daten werden aufbewahrt. In zwei Fällen kann auch ein PUSH auf den Datenstack entfernt werden. Im letzten Fall werden beide Literals entfernt und das Aufzählen wird schon während des Compilierens vorgenommen, wonach dann ein neues Literal assembliert wird.

```
:SPECIAL +
CASE 2STATES
$05 OF REMOVE-LIT >R R16 X+ LD, R16 R> ADDI, -X R16 ST,
  1 RESET-OPT 0 ENDOF ( Stack + Literal )

$15 OF REMOVE-LIT&PUSH >R R16 R> ADDI, -X R16 ST,
  1 RESET-OPT 0 ENDOF ( R16 + Literal )

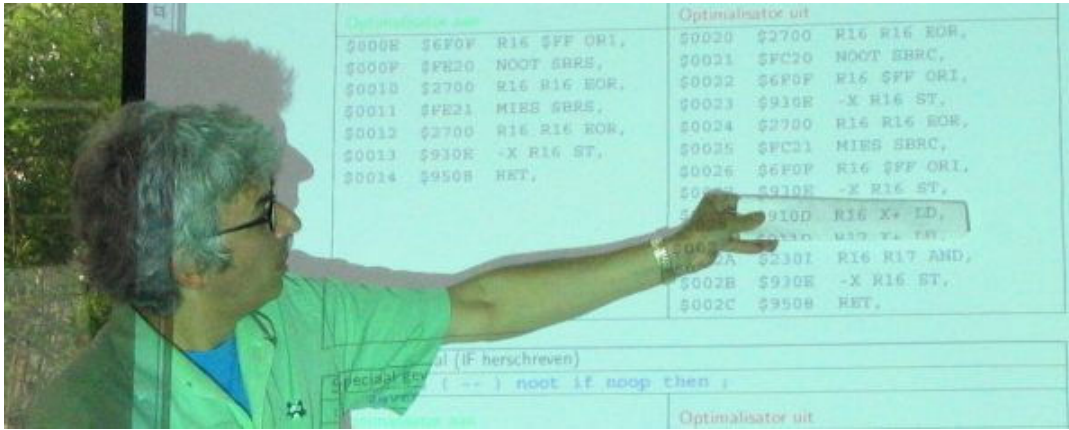
$25 OF REMOVE-LIT&PUSH >R R16 R> ADDI, -X R16 ST,
  1 RESET-OPT 0 ENDOF ( Literal + Literal )

$55 OF REMOVE-LIT >R REMOVE-LIT R> + 255 and
  >R R16 R> LDI, -X R16 ST,

OPT3 RESET-OPT 5 SET-OPT 0 ENDOF -1 SWAP
( Kein Spezialfall gefunden )
ENDCASE
;SPECIAL
```

Die definierenden Worte in ByteForth, die sich um die verschiedenen RAM-Ausmaße kümmern, werden





Willem Ouwerkerk erläutert, wie man einen AVR-Prozessoren programmiert.

schon während des Auswählens des AVR-Typs, der Variablen VA (Variablen-Adresse) und der Variablen VE (Variablen-Ende) gesetzt. Sie werden gebraucht, um zu kontrollieren, ob noch genügend RAM für VARIABLE vorhanden ist.

```
: VARIABLE ( "name" -- addr ) \ Cross
  VA VE > 20 ?ERROR \ Noch innerhalb des gueltigen RAMs?
  CREATE VA , 1 DUP , +TO VA IMMEDIATE
  DOES>
    @ POSTPONE LITERAL \ Variablen-Offset auf den Stack
    ; \ waehrend der Laufzeit! (wie ueblich).
```

Unterschiede zu anderen AVR-Compilern

Zunächst der Unterschied zwischen IRTC und AVR-ByteForth bei der Code-Erzeugung, danach die einfache Syntax des I/O-Bit-Befehls gegenüber dem IAR-C-Compiler.

Der MPE(IRTC)-Forth-Compiler verwendet überall die langen STS- und STD-Befehle. ByteForth verwendet sie nirgends. IRTC ist ein 16-Bit-Forth, ByteForth ist ein 8-Bit-Forth; Tabelle 2 zeigt die Gegenüberstellung.

Der IAR-C-Compiler benötigt einen viel umständlicheren Code, um ein einfaches SET, CLEAR oder FROM (lies) Bit zu verwirklichen. AVR-ByteForth kennt neben diesen Präfix-Operatoren noch ADR. ADR fischt die I/O-(Bit)Adresse für die Verwendung in Code-Definitionen heraus. Sodann gibt es noch TO, TOGGLE und PULSE. Man überlege sich deren Funktion selbst. Die Präfix-Operatoren funktionieren nicht nur mit BIT-SFR, sondern mit allen in AVR-ByteForth eingebauten Datentypen. Die unten stehenden Beispiele erzeugen beide einen SBI- oder CBI-Befehl. Es ist sogar möglich, Präfix-Operatoren einem neuen Datentyp zuzufügen, und zwar mit Hilfe von METHODS. Zunächst IAR-C:

```
// Macros for setting/clearing IO bits
// in the IAR C-compiler
// a=[sfr name] , b=[bit number]
#define set_sfrbit(a,b) ( (a) |= (1<<(b)) )
#define clr_sfrbit(a,b) ( (a) &= (~(1<<(b))) )
#define test_sfrbit(a,b) ( (a) & (1<<(b)) )
```

```
#define DIVVALVE_PORT PORTB
#define DIVVALVE_FOR 6 // Port B.6

void DivValveOnForward(void)
{
    set_sfrbit(DIVVALVE_PORT, DIVVALVE_FOR);
}
```

Nun AVR-ByteForth:

```
PORTB 6 BIT-SFR VALVE \ PB.6 controls valve

: VALVE-ON ( -- )
  SET VALVE
;
```

Programmbeispiel:

Zuerst wird der gewünschte Mikro-Controller ausgewählt und in diesem Zusammenhang werden auch alle Einstellungen, die zu diesem Controller gehören, vorgenommen, wie z.B. das Speichermodell, die einsetzbaren Opcodes und das ISP-Programmierprotokoll. Danach werden alle Labels und Interrupt-Vektoren, die zu diesem Chip gehören, geladen. Dann beginnt das eigentliche Programm. Mit Hilfe des Labels PORTB und des definierenden Wortes SFR wird ein I/O-Register zugänglich gemacht. Dann die erste Colon-Definition namens BLINKER, die die LEDs auf PORTB mal schnell aufblitzen lässt. Dann geht es wieder zum Hauptprogramm, wo zunächst die Forth-Stackmaschine initialisiert wird. Sodann wird das I/O-Register auf Ausgabe geschaltet und BLINKER wird kompiliert. Danach treten wir in eine Endlos-Schleife ein, in welcher fast alle Standard-Worte stehen, mit Ausnahme von TO LEDS. Auf das Wort ; folgt das Wort MAIN, das ähnlich wie IMMEDIATE auf das zuletzt definierte Wort rückwirkt und den Resetvektor mit der Adresse dieses Wortes füllt. Das war's.

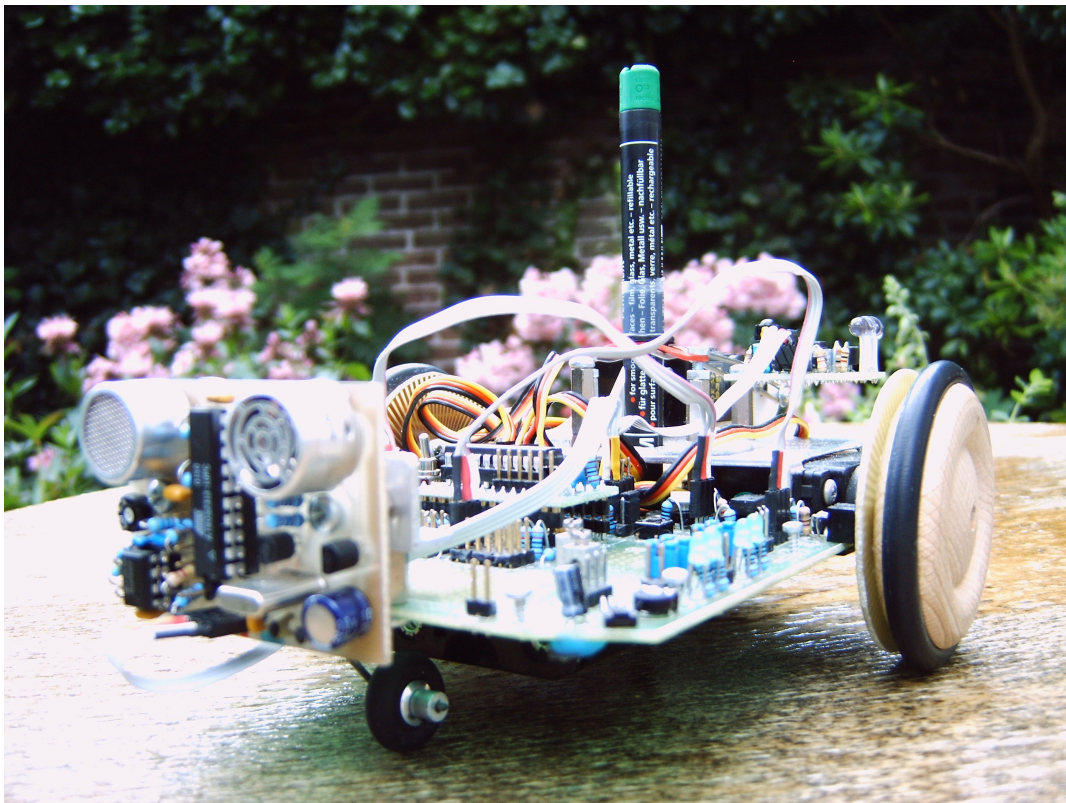
Links

ByteForth Homepage (niederländisch):
<http://www.forth.hccnet.nl/bytforth.htm>


```

1  \ Lauflicht mit ATTiny2313 auf Port B, Laenge: 162 Bytes, laenger
2  \ hauptsaechlich wegen des inline expandierten MS-Makros.
3  TINY2313          \ Compiliere fuer ATTiny2313
4  INCLUDE TARGET.FRT      \ Lade Labels & Vektoren
5  PORTB  SFR LEDS        \ Port B mit 8 LEDs
6
7  : BLINKER      ( -- )      \ Zeige Hochfahren an
8      0 TO LEDS  250 MS      \ Alle LEDs an
9      -1 TO LEDS 250 MS ;    \ Alle LEDs aus
10
11 : LAUFLICHT   ( -- )
12   SETUP-BYTEFORTH      \ Installiere Forth-Maschine
13   -1 SETDIR LEDS       \ Port B wird als Ausgang verwendet
14   BLINKER
15   BEGIN
16       8 0 DO           \ Durchlaufe Schleife achtmal
17           1 I LSHIFT    \ Mache Bitmuster
18           INVERT TO LEDS \ Kehre um und steuere die LEDs an
19           100 MS        \ Noch etwas warten
20   LOOP
21   AGAIN            \ Fange wieder von vorn an
22 ; MAIN            \ Setze das LAUFLICHT in den Resetvektor

```

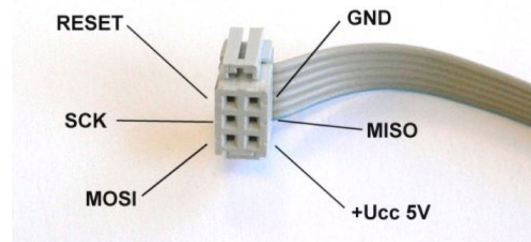


Ein *malender* Ushi-Roboter mit einer AVR-Steuerung und einem AVR-Sensormodul — programmiert in ByteForth

USBasp — ein Programmierer für AVR

Ulrich Hoffmann

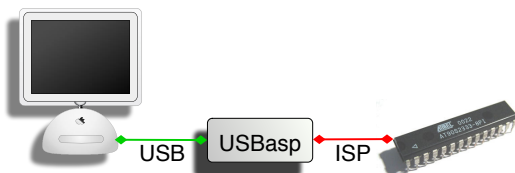
Die *Im-System-Programmier*-Schnittstelle (engl. in system programming, ISP) der Firma Atmel erlaubt Programmiergeräten, AVR-Prozessoren direkt in der Schaltung zu programmieren. Der von Thomas Fischl entwickelte Open-Source-Programmierer USBasp ist ein solches Gerät, das selbst von einem ATmega8-Prozessor gesteuert wird und seine USB-Schnittstelle rein durch Software realisiert.



Forth-Programme wollen ihren Weg in den Programmspeicher der AVR-Prozessoren finden.

Handelt es sich um ein interaktives Forth-System, so muss nur der Forth-Kern als eine Art Bootstrap-Loader in den Prozessor geschrieben werden. Forth-Programme werden dann interaktiv auf diesen Kern geladen und gegebenenfalls durch Autoprogrammierung der AVR-Chips in den Programmspeicher übernommen. Handelt es sich um einen Forth-Cross-Compiler, dann werden Forth-Anwendungen schon auf dem Entwicklungs-PC in AVR-Code übersetzt und gelangen dann auf direktem Weg in den Chip. In beiden Fällen muss der AVR-Programmspeicher von außen durch ein Programmiergerät beschrieben werden. Der von Thomas Fischl entwickelte kostengünstige Open-Source-AVR-Programmierer USBasp kann dazu verwendet werden.

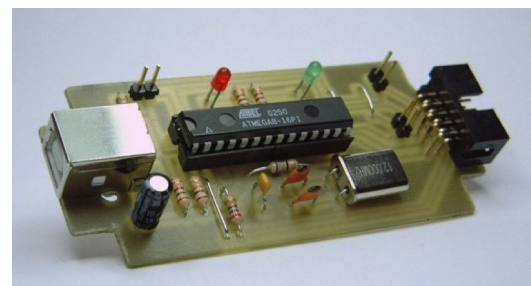
Er wird wie folgt zwischen Entwicklungs-PC und AVR-Zielsystem geschaltet:



Die Verbindung zum Entwicklungs-PC (oder Mac) wird dabei über eine USB-Schnittstelle hergestellt, deren Ansteuerung im USBasp durch die AVR-USB-Treiber-Firmware [2] von Objective Development rein durch Software realisiert ist. Die Verbindung zum Zielsystem erfolgt über die *Im-System-Programmier*-Schnittstelle ISP mittels des AVR-typischen 6Pin- oder 10Pin-Steckverbinders [7]:

Im USBasp werkelt selbst ein ATmega8-Prozessor, der neben dem AVR-USB-Treiber auch Thomas Fischls freie Firmware enthält, die die ISP-Schnittstelle bedient und die Programmieralgorithmen realisiert. Eine Steuersoftware auf dem Entwicklungs-PC kontrolliert wiederum den USBasp. Unter Windows, Linux, Mac OS X, ... kommt das Open-Source-Kommandozeilen-Programm AVRDUDE [3] zum Einsatz, das neben zahlreichen anderen AVR-Programmierern eben auch den USBasp unterstützt. Zum Ansprechen der USB-Schnittstelle (auf PC-Seite) stützt sich AVRDUDE dabei auf die Open-Source-Bibliothek LibUsb [4].

Der Programmierer lässt sich auch von Ungeübten ohne große Probleme selbst zusammenbauen. Neben dem ATmega8-Prozessor werden nur ein Quarz, zwei Hände voll Widerstände, ein paar Kondensatoren und zwei Leuchtdioden zur Statusanzeige benötigt. Verwendet man Thomas Fischls Platinentwurf, dann kann ein aufgebauter USBasp so aussehen:



Ein betriebsbereiter USBasp-Programmierer

Weitere Informationen, unter anderem der USBasp-Schaltplan, finden sich auf der USBasp-Homepage [1].

Links

- [1] USBasp-Hompage: <http://www.fischl.de/usbasp/>
- [2] AVR-USB-Treiber (rein in Software): <http://www.obdev.at/products/avrusb/>
- [3] AVRDUDE, AVR-Downloader/UploaDEr: <http://www.nongnu.org/avrdude/>
- [4] LibUsb für Unix: <http://libusb.sourceforge.net/>
- [5] LibUsb für Windows: <http://libusb-win32.sourceforge.net/>
- [6] Atmel-AVR bei Wikipedia: http://de.wikipedia.org/wiki/Atmel_AVR
- [7] ISP-Steckverbinder: http://www.hardwarebook.info/AVR_ISP

Forth von der Pike auf

Ron Minke

Die hier mit freundlicher Genehmigung der HCC-Forth-gebruikersgroep wiederzugebende achteilige Artikelserie erschien in den Jahren 2004 und 2005 in der Zeitschrift "Vijgeblaadje" unserer niederländischen Forth-Freunde. Übersetzung: Fred Behringer.

Wir haben im vorliegenden Sonderheft die Übersetzungen aller acht Teile zu einem einzigen Ganzen zusammengefügt. Wir hoffen, dass es uns gelungen ist, Unstimmigkeiten, die einzig und allein auf unser Konto gehen, zu vermeiden (die Redaktion). Und hier der Text des Autors:

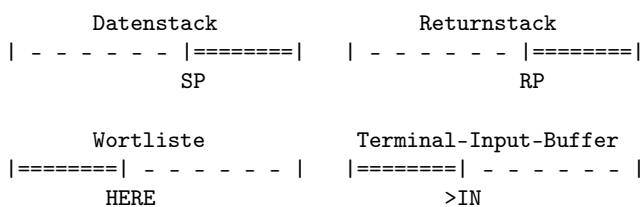
Die folgenden Zeilen stellen den Versuch dar, ein Forth-System auf die Beine zu stellen, dessen Voraussetzung "überhaupt nix", oder auf gut Deutsch "from scratch", lautet.

Zunächst ein paar Worte über die Vorgeschichte: 1997 bringt die Firma Atmel aus ihrer AVR-Serie den Mikroprozessor AT90S8515 auf den Markt. Laut Atmel ist die AVR-Serie eine Serie von revolutionären Mikroprozessoren, die zwar die Vorteile von ähnlichen Prozessoren auf diesem Marktsegment von 8-Bitern haben, aber nicht deren Nachteile. Überdies haben Chipbäcker und Software-Spezialisten (sprich C-Compilerbauer) beim Entwickeln des Prozessors von Anfang an zusammengearbeitet. Weitere Informationen findet man auf der Atmel-Website.

Ein neuer Prozessor! Für den echten Forth-Enthusiasten ist das natürlich eine Herausforderung: Da muss Forth drauf laufen! Und damit begann es ...

Erst flugs den Befehlssatz des neuen Prozessors durchstudieren. Und was hat er denn an Timern, Ports, seriellen I/O-Anschlüssen und so weiter schon an Bord? Und wie sieht denn ein Forth-System auf unterster Ebene noch gleich aus? All die verschiedenen Pointer ...

Wie baut sich Forth auf? Der virtuelle Forth-Computer ist ein Programm, das im Arbeitsspeicher eines echten Computers, im vorliegenden Fall unseres AVR-Mikroprozessors, läuft. Der vorhandene Speicher ist in Bereiche für verschiedene Aufgaben aufgeteilt und das Ganze sorgt dafür, dass der echte Computer den Forth-Kommandostrom verarbeiten kann.



Das ist eine schematische Wiedergabe der Funktionsteile des einfachsten Falles eines virtuellen Forth-Computers. Er besteht aus einer Wortliste (dem Dictionary), zwei Stacks, einem Terminal-Input-Buffer und eventuellem Disk-IO (Input-Output-Anschlüsse).

Die virtuelle Forth-Maschine verwendet einen Satz von Registern, um die hauptsächlichsten Informationen über den Verlauf des Programms zu steuern. An Registern treten auf:

SP	Datenstack-Pointer
RP	Returnstack-Pointer
IP	Interpreter-Pointer (wo wird gerade gearbeitet)
W	Word-Pointer (zum laufenden Wort)
PC	Program-Counter (Maschinenprogramm-Zähler)

Sodann beginnt die "echte Arbeit": Wie wollen wir die Interna unseres Mikroprozessors (Register und dergleichen) zur Implementation der virtuellen Forth-Maschine einsetzen? Um diese Frage gut beantworten zu können, müssen wir erst ausfindig machen, wie die virtuelle Forth-Maschine ihre Register verwendet. Zur Verdeutlichung haben wir einen Pseudobefehlssatz für einen Pseudo-Assembler definiert.

Dieser Befehlssatz enthält nur drei Befehle:

MOV dest,src	Kopiere das Datenregister src ins Register dest
INC dest	Vergrößere den Inhalt des Registers dest um 1
DEC dest	Verkleinere den Inhalt des Registers dest um 1

Es existiert auch eine indirekte Form: Das Setzen von dest oder src in Klammern signalisiert, dass es hier "um den Inhalt von" geht, und nicht um das Register selbst. Die Klammern deuten eine Indirektionsebene an.

MOV dest,(src)	Kopiere Daten vom INHALT des src-Registers, d.h., wohin src zeigt, ins Register dest.
-----------------------	---

Die Wortliste von Forth ist eine verkettete Liste von aneinandergereihten Wortdefinitionen. Man erkennt diverse eigenständige Teile (Felder):

Namensfeld	Hier steht der Name des Wortes
Linkfeld	Ankopplung an das vorherige Wort
Codefeld	Zeiger auf ausführbaren Maschinencode
Parameterfeld	Was muss dieses Wort tun?

Die oben stehenden Pseudobefehle brauchen wir, um uns in den aneinandergeschalteten Feldern dieser Wortdefinitionen umherbewegen zu können. Das Namensfeld und das Linkfeld sorgen dafür, dass Definitionen zu einer linearen Liste zusammengekoppelt werden können, die dann vom Textinterpreter durchsucht werden kann. Wie das genau geschieht und wie das eine oder andere in den



Speicher gesetzt wird, bewahren wir uns für später auf. Das Codefeld enthält die Adresse des Code-Interpreters für diese Definition und das Parameterfeld enthält alle Informationen, die diese Definition benötigt, um ihre Aufgabe auszuführen.

Die Suche nach der besten Kombination von Prozessorregistern, um Forth laufen lassen zu können, beginnt beim Wort EXECUTE im Text-Interpreter. Das Wort EXECUTE ruft ein Stückchen Maschinencode auf, um das betreffende Wort auszuführen. Hierbei müssen wir berücksichtigen, dass bei jedem Aufruf von EXECUTE die Adresse des Codefeldes (die CFA, Code Field Address) desjenigen Wortes, das ausgeführt werden soll, vom Text-Interpreter auf den Stack gelegt wird.

Es folgt der Code in Pseudo-Assembler-Notation:

EXECUTE:	(CFA -) Codefeld-Adresse steht auf dem Datenstack
MOV W, (SP)	Kopiere den obersten Datenstack-Eintrag, CFA, ins W-Reg.
INC SP	Verwerfe den Datenstack-Wert, mache einen Schritt vorbei
MOV PC, (W)	Kopiere die Adresse des Code-Interpreters in der Code Field Address in den Program Counter: Indirekter Sprung dahin

EXECUTE kopiert die CFA in das W-Register und springt – indirekt – über den Inhalt eben dieses W-Registers zum Code-Interpreter. Der Maschinencode des Code-Interpreters wird nun ausgeführt. Da das W-Register selbst weiterhin auf die CFA des Forth-Wortes zeigt, das gerade ausgeführt wird, kann der Code-Interpreter Informationen darüber einholen, wie das Forth-Wort weiter zu behandeln ist. In unserer Forth-Implementation (FIG-Forth, siehe betreffende Literatur) liegt das Parameterfeld direkt hinter dem Codefeld. Wenn die Parameterinformation benötigt wird, holen wir diese von hier aus über den betreffenden Code-Interpreter ein.

Alle Code-Interpreter müssen ihren ausführenden Teil mit einem Stück Code beenden, der “NEXT” genannt wird. Der gibt die Kontrolle an den darüberliegenden Text-Interpreter zurück. Wird der Code-Interpreter von einem Hi-Level-Wort aufgerufen, so wird die Kontrolle an dieses Hi-Level-Wort zurückgegeben. NEXT setzt voraus, dass die Adresse des als nächstes auszuführenden Wortes im IP-Register aufbewahrt wird (der “Wo-bin-ich-geblieben”-Pointer). So kann der Text-Interpreter die Liste der im Parameterfeld einer Hi-Level-Definition gelagerten Adressen scannen.

Das Codestück für NEXT sieht in unserem Pseudo-Assembler so aus:

NEXT:	IP zeigt auf das als nächstes auszuführende Wort
MOV W, (IP)	Kopiere den Inhalt von IP, die CFA des als nächstes auszuführenden Wortes, ins W-Register.
INC IP	Lass IP auf das Wort NACH dem momentanen zeigen, um da schnurstracks weiter zu gehen.
MOV PC, (W)	Führe den Code-Interpreter aus, dessen Adresse jetzt im W-Register sitzt. Dieser Wert kommt aus dem Codefeld des gerade auszuführenden Wortes (indirekter Sprung).

Alle Worte in der Wortliste können durch EXECUTE ausgeführt werden, wenn ihre CFA auf dem Datenstack steht, oder durch NEXT, wenn die CFA, die sich in der Wortliste befindet, durch IP angezeigt wird. Anzumerken bleibt noch, dass es bei der Ausführung eines Forth-Wortes der Maschinencode des Code-Interpreters ist, der vom “Host-Computer” ausgeführt wird. NEXT und EXECUTE kriegen die Adresse dieses Code-Interpreters aus dem Codefeld der Definition des betreffenden Wortes zugewiesen.

Unten werden wir weiter auf die Interna eingehen, und zwar auf das tatsächliche Abbilden der Register des AVR-Prozessors auf die Register der virtuellen Forth-Maschine, und natürlich auch auf den zugehörigen Maschinencode.

16-Bit-Register auf dem AVR

Jetzt werden wir versuchen, die Register des AVR-Prozessors auf die Register der virtuellen Forth-Maschine abzubilden. Zuerst müssen wir uns klar machen, dass wir drauf und dran sind, ein 16-Bit-Forth auf einem 8-Bit-Prozessor hochzuziehen. Wir werden also alle Register mit Register-Paaren des AVR-Prozessors koppeln müssen (die zum Glück in genügender Zahl vorhanden sind).

Zunächst noch schnell rekapitulieren, welche Register wir nötig haben:

PC	Maschinenprogrammzähler	Der Motor des Mikro-Controllers.
SP	Datenstack-Pointer	Zeigt auf das zur Zeit oberste Element des Datenstacks.
RP	Returnstack-Pointer	Zeigt auf das zur Zeit oberste Element des Returnstacks.



IP Interpreter-Pointer Zeigt auf die nächste Anweisung (Forth-Definition), die ausgeführt werden soll; überwacht die Reihenfolge der Ausführung.

W Wort-Pointer Zeigt auf die Definition, die gerade ausgeführt wird; nötig, um den Parameterteil dieser Definition anzuspringen.

Die Wahl des Registers PC ist einfach: Der PC! Der Programmzähler eines Mikro-Controllers ist das einzige "Register", das das "Runnen" des eigentlichen Programms, des Low-Level-Programms, steuert.

Der AVR-Prozessor hat 32 frei verfügbare 8-Bit-Register an Bord: R0 bis R31. Zum Glück haben die Prozessor-Entwickler gut nachgedacht: 8 Register können zu 4 Registerpaaren zusammengekoppelt werden. Das sind:

- R24 - R25 W,
- R26 - R27 X,
- R28 - R29 Y,
- R30 - R31 Z.

Um die richtige Wahl treffen zu können, müssen wir uns die verschiedenen Eigenschaften der Registerpaare ansehen.

Der Datenstack

Wir beginnen unsere Zuordnungssuche mit SP, dem Datenstack-Pointer. Wir wollen Daten auf den Datenstack legen. Hier begegnen uns schon vier Wahlmöglichkeiten:

Pseudo-Code	Was der tut
1. INC SP MOV (SP),data	Datenstack wächst nach oben, erst Pointer anpassen, danach dann Daten ablegen.
2. MOV (SP),data INC SP	Datenstack wächst nach oben, erst Daten ablegen, danach dann Pointer anpassen.
3. DEC SP MOV (SP),data	Datenstack wächst nach unten, erst Pointer anpassen, danach dann Daten ablegen.
4. MOV (SP),data DEC SP	Datenstack wächst nach unten, erst Daten ablegen, danach dann Pointer anpassen.

Die Entscheidung darüber, ob der Datenstack nach oben oder nach unten wachsen soll, stellen wir noch etwas zurück. Allerdings wäre es besonders schön, wenn wir keine separaten Befehle INC oder DEC benötigen würden. Am liebsten hätten wir einen Befehl, der automatisch

auch gleich noch inkrementiert bzw. dekrementiert. Beim AVR-Prozessor steht diese Möglichkeit für die Registerpaare X, Y und Z tatsächlich zur Verfügung. Mit dem Befehl ST (store)

ST X,R5

setzen wir den Inhalt des Registers R5 an die Stelle, wohin der Inhalt des X-Registerpaares zeigt. (Ausführliche Informationen finden sich auf der ATMEL-Webseite.)

Der Befehl

ST X+,R5 (Auswahlmöglichkeit 2)

macht dasselbe, aber danach wird in einem einzigen Zug auch noch der Inhalt des X-Registerpaares um 1 erhöht (post increment). Und gratis ist das obendrein: Die Post-Increment-Aktion kostet *keinen* Extra-Maschinentakt.

Der Befehl

ST -X,R5 (Auswahlmöglichkeit 3)

arbeitet genauso, jedoch wird erst der Inhalt des X-Registerpaares um 1 erniedrigt, (pre decrement), bevor R5 auf dem dann angewiesenen Platz aufbewahrt wird.

Nun müssen wir uns noch damit beschäftigen, was der SP-Pointer macht: SP zeigt auf das oberste (oder unterste, je nach Richtung) Element des Datenstacks. Wollen wir auf dem Stack neue Daten ablegen, dann muss *erst* Platz gemacht werden, bevor wir die Daten abspeichern können. Von den oben genannten vier Möglichkeiten bleibt also nur die mit Nummer 3 übrig. Damit haben wir uns für ein Nach-unten-Wachsen des Datenstacks entschieden.

Unsere virtuelle Forth-Machine ist 16 Bit breit, so dass wir nun für den 8-Bit-AVR-Prozessor zum folgenden Code kommen: (Daten stehen in R4 und R5 bereit)

ST -X,R4
ST -X,R5

Man beachte, dass wir damit bereits den Maschinencode für das Forth-Wort "!" (store) gemacht haben.

Aber gemacht! Wir wollen unsere Wahl der AVR-Register-Zuweisung bequem gestalten: Sowohl das X- wie auch das Y- und das Z-Register haben die oben genannte Autodekrement-Eigenschaft. Die endgültige Wahl für SP muss also noch etwas zurückstehen.

Wir bekommen es jetzt noch mit etwas anderem zu tun. Es muss noch beschlossen werden, was zuerst aufzubewahren ist: Das obere Byte oder das untere Byte der 16-Bit-Zahlenwerte. Oder auch: Wie wollen wir die Daten auf dem Stack gelagert sehen? Dem Forth-System selbst ist diese Frage egal. Wir müssen uns also nach einem anderen Kriterium umsehen. Die Entscheidung darüber, welche Wahl wir treffen, bleibt noch einen Augenblick lang offen (na ja ... so dringend ist das noch nicht).



Der Returnstack

Erst noch ein weiteres Forth-Register: Der RP. Der Returnstack heißt so, weil die virtuelle Forth-Maschine ihn dazu verwendet, die Rückkehr-Adressen (return = zurück) aufzubewahren, diejenigen Adressen, wo besagte virtuelle Maschine weiterarbeiten soll, *nachdem* ein High-Level-Wort vollständig ausgeführt worden ist. Wenn ein High-Level-Forth-Wort ein schon früher definiertes anderes Forth-Wort aufruft, wird die Adresse des *nächsten* Wortes in die Wortliste auf dem Returnstack eingereiht. Diese Adresse wird wiederhergestellt, sobald das gerade eben aufgerufene Wort vollständig abgearbeitet ist. Das Programm kann dann vom Verzweigungspunkt aus weitergehen.

Der erste Gedanke, der aufkommt, ist der, ob man den RP nicht an den Maschinen-Stack-Pointer SP (nun, beim AVR heißt der nun einmal so) koppeln sollte. Dieses AVR-SP-Register hat genau dieselbe Funktion wie die, die wir bei der virtuellen Forth-Maschine haben wollen: Es zeigt auf die aufbewahrten Adressen, wo der Programm-Counter weiterarbeiten soll, wenn ein Unterprogrammaufruf beendet ist. Auto-Inkrement, bzw. Auto-Dekrement sind auch eingebaut. Beim Aufrufen eines Maschinensprach-Unterprogramms wird gleichzeitig die unmittelbar folgende Adresse auf den Returnstack gelegt. Wenn das Unterprogramm fertig ist, wird diese Adresse wieder in den Programm-Counter PC zurückgeschrieben, so dass das Programm weiterlaufen kann, als ob da nichts geschehen wäre. Also genau das, was wir brauchen.

Was passiert da nun genau? Angenommen, ein willkürlich herausgesuchtes Stück AVR-Maschinencode ruft ein Unterprogramm auf (CALL). Verfolgt man den sich ergebenden CALL-Code (siehe ATMEL-AVR-Befehlssatz auf deren Website), so bekommt man die Sequenz (in Pseudo-Code, in Bytes):

```
MOV (AVR-SP),unteres-Byte-momentaner-PC + 1
DEC AVR-SP
MOV (AVR-SP),oberes-Byte-momentaner-PC + 1
DEC AVR-SP
MOV PC, (momentaner-PC)
```

Wir sehen hier, dass der Returnstack nach *unten* wächst und dass *erst* die Daten abgespeichert werden, bevor Platz gemacht wird. Der AVR-SP zeigt also offenbar auf den ersten freien Platz auf dem Stack. So haben sich die Entwickler bei ATMEL das jedenfalls vorgestellt.

Festlegungen

Nun wissen wir zumindest einiges. Beim Codieren der virtuellen Forth-Maschine wollen wir es uns so einfach wie möglich machen. Wir halten uns an die oben genannte Arbeitsweise.

Wir treffen drei Entscheidungen:

1. Die Forth-Stacks SP und RP wachsen *nach unten*.

2. Das untere Byte eines 16-Bit-Wortes wird zuerst auf den Stack gelegt, darunter dann das obere Byte.
3. Der Forth-Pointer zeigt auf das *obere* Byte eines 16-Bit-Wortes (und also — in Abweichung von der Arbeitsweise des AVR-SPs — nicht auf den leeren Platz unter dem Wort). Das entspricht der Wahlmöglichkeit 3 bei der obigen Besprechung der Datenstack-Zuweisung.

Nun denn... jetzt kommt allmählich etwas Struktur in die virtuelle Forth-Maschine.

Nach den drei bereits getroffenen Entscheidungen wollen wir uns jetzt dafür interessieren, wie wir Daten auf den Returnstack legen können. Für Returnstack-Operationen gibt es im AVR-Prozessor eigens zwei Maschinencode-Befehle:

PUSH	legt Daten auf den Returnstack
POP	holt Daten vom Returnstack

Diese Befehle arbeiten so, als wenn in einem Unterprogrammaufruf CALL (in Pseudocode) stünde:

```
PUSH:
MOV (AVR_SP), Rn   lege Daten von Register Rn ab
DEC AVR_SP         zeige auf die neue leere Stelle

POP:
INC AVR_SP         erhöhe Pointer, um an die Daten zu kommen
MOV Rn, (AVR_SP)  kopiere Daten, lass alten Wert stehen
```

Wie wir sehen, zeigt der Pointer auf einen freien Platz. Wir hatten uns aber vorgenommen (Entscheidung 3), genau das in unserer virtuellen Forth-Maschine **nicht** zu tun. Doch noch ist nicht alles verloren: Mit den oben stehenden PUSH- und POP-Befehlen können wir alle Returnstack-Operationen verwirklichen. Innerhalb der virtuellen Forth-Maschine ist es weniger von Belang, *wo* der Pointer genau hinzeigt, solange wir nur an die Daten kommen. Wollen wir tatsächlich wissen, wo der Pointer hinzeigt, dann müssen wir den Offset zwischen dem angezeigten freien Platz und dem Platz der eigentlichen Daten in Rechnung ziehen. Zum Glück beträgt dieser Offset nur 1 Byte. Unser Vorgehen, für den Forth-RP den AVR-SP zu wählen, stellt sich also als gangbar heraus.

Entscheidung 4: Das Forth-RP wird dem AVR-SP-Register zugewiesen.

Der Interpretier-Pointer

Das nächste Register der virtuellen Forth-Maschine, für das wir eine AVR-Lösung suchen wollen, ist der Interpretier-Pointer. Der IP zeigt auf den nächsten Befehl (Forth-Definition, Wort) der in unserem Forth-Programm ausgeführt werden soll. Der IP steuert die Reihenfolge der Ausführung auf dieselbe Weise, wie der Maschinenprogramm-Counter PC die Reihenfolge

in einem Assembler-Programm bestimmt. Das wichtigste Codestück, das den IP verwendet und steuert, ist das Stück Code für NEXT (lesen Sie sich die eingangs gebrachten Erklärungen noch einmal durch). Sodann wird IP unter anderem in den Codeteilen verwendet, die Sprunganweisungen ausführen (BRANCH, LOOP u. dgl.). Der Vollständigkeit halber wiederholen wir das Stück Pseudo-Assembler-Code für NEXT:

NEXT:	IP zeigt auf das als nächstes auszuführende Wort
MOV W, (IP)	kopiere den Inhalt von IP, die CFA des als nächstes auszuführenden Wortes, ins W-Reg.
INC IP	Lass IP auf das Wort <i>nach</i> dem momentanen Wort zeigen, um dann dort ohne Umwege fortzufahren.
MOV PC, (W)	Führe den Code-Interpreter, dessen Adresse nun im W-Register sitzt, aus. Dieser Wert kommt aus dem Codefeld des momentan auszuführenden Wortes (indirekter Sprung).

Welches der AVR-Registerpaare W, X, Y und Z können wir hier nun am vorteilhaftesten einsetzen? Die Wahl wird eigentlich durch die letzte Zeile im Pseudoassemblercode bestimmt:

```
MOV PC, (W)
```

Hier wird indirekt auf eine Adresse gesprungen, die im Virtuellen-Forth-Register **W** steht. Das einzige AVR-Registerpaar, das indirekte Sprünge zulässt, ist das Z-Registerpaar R30-R31 (Genaueres findet man im Befehlssatz auf der Atmel-Website). Wir benötigen das Z-Registerpaar als Zwischenschritt, um indirekt springen zu können. Leider verfügt der AVR nicht über Maschinenbefehle, die ein Registerpaar in einem einzigen Zug laden können. Also müssen wir das in zwei Schritten tun. Hierzu definieren wir **ZL** (= R30) als den unteren Teil (low) des Z-Registerpaares und **ZH** (= R31) als den oberen Teil (high). Wir brauchen auch ein W-Registerpaar zur Zwischenlagerung. Dafür wollen wir vorläufig das W-Registerpaar des AVR wählen. Ob diese Entscheidung richtig war, werden wir später sehen (dass sich im AVR-Chip ein Registerpaar befindet, das auch W heißt, ist Zufall). Das W-Registerpaar spalten wir in zwei Teile auf, **WL** (= R24) und **WH** (= R25).

Wenn wir den gesamten Pseudocode für NEXT hinschreiben, bekommen wir:

Pseudocode	Assemblercode	NEXT-Routine
MOV W,(IP)	(1) Ld WH,Rn	indirekt hereinholen
INC IP	(2) Inc Rn	
	(3) Ld WL,Rn	indirekt hereinholen
	(4) Inc Rn	
	(5) Mov Raaa,WH	Pointer setzen
	(6) Mov Rbbb,WL	
MOV PC,(W)	(7) Ld ZH,Raaa+	indirekt, auto incr
	(8) Ld ZL,Raaa+	
	(9) IJmp	indirekter Sprung

Uff, das ist ein schönes Stück Code! Um es in den Griff zu bekommen, nochmal alles der Reihe nach. Die Assembler-Befehle in den Zeilen 1-4 holen indirekt die Stelle herein, wo die *nächste* Befehlsdefinition zu finden ist. Gleichzeitig wird der Pointer so erhöht, dass er auf die darauffolgende Definition zeigt. Die Zeilen 5 und 6 kopieren den hereingeholten Wert in ein Registerpaar Raaabbb. Das kann das X-, das Y- oder das Z-Registerpaar sein. Auf welches die Wahl fällt, werden wir gleich sehen. Da unser Forth eine indirekt gefädelte Version (das klassische Modell) ist, müssen wir abermals einen indirekten Wert hereinholen, wenn wir erfahren wollen, wo der eigentliche Maschinencode für diese Definition steht.

Die Zeilen 7 und 8 holen diesen indirekten Wert herein und legen ihn in das Z-Register. Wissen Sie es noch (Entscheidung 2): Erst das obere Byte hereinholen, dann das untere. Man beachte, dass dabei von der Auto-Inkrement-Funktion Gebrauch gemacht wird, so dass wir das Registerpaar Raaabbb nicht selbst zu erhöhen brauchen. Und schließlich wird über den Befehl IJmp in Zeile 9 der indirekte Sprung vollzogen. Der zum gerade auszuführenden Forth-Wort gehörende tatsächliche Maschinencode macht sich nun an seine Arbeit.

Wir müssen noch festlegen, welches der in Frage kommenden Registerpaare X, Y und Z wir für das eben verwendete Registerpaar Raaabbb einsetzen wollen. Dabei müssen wir auch daran denken, dass die Register IP und SP der virtuellen Forth-Maschine noch endgültig festgelegt werden müssen. Große Auswahl haben wir eigentlich nicht, das Z-Registerpaar haben wir bereits für indirekte Sprünge verwendet. Bleiben für die Zuweisung an IP und SP das X- und das Y-Registerpaar übrig (welches zu welchem, werden wir sogleich sehen). Zur Darstellung von Raaabbb bleibt also nur noch das Z-Registerpaar übrig. ??? Aber das Z-Registerpaar hatten wir ja gerade verwendet...??? Wir befreien uns aus dieser Situation, indem wir *beides* tun!

Entscheidung 5: Wir verwenden das Z-Registerpaar für Raaabbb *und* auch für indirekte Sprünge.

Durch wohlüberlegten Umgang mit dieser Kombination entsteht der folgende AVR-Maschinencode (wir benötigen dabei allerdings zwei Zwischenregister, für welche wir R0 und R1 nehmen).



Pseudocode	Assemblercode	Next-Routine
MOV W,(IP)	(1) Ld WH,Rn	indirekt hereinholen
INC IP	(2) Inc Rn (3) Ld WL,Rn	indirekt hereinholen
	(4) Inc Rn (5) Mov ZH,WH (6) Mov ZL,WL	Pointer setzen
MOV PC,(W)	(7) Ld R0,Z+	ind, oberes Byte, auto incr
	(8) Ld R1,Z (9) Mov ZL,R1	ind, unteres Byte kopiere (Z) zurück nach Z
	(10) Mov ZH,R0 (11) IJump	indirekter Sprung

Wie sich herausstellt, gibt es im Befehlssatz der MEGA-AVR-Prozessorserie einen Befehl, der ein Registerpaar in einem einzigen Zug kopieren kann, so dass die Zeilen 5 und 6, bzw. 9 und 10 zu einem einzigen Mov zusammengefasst werden können. Für die kleinen AVR-Prozessoren trifft das jedoch nicht zu.

Wir können aber auch von den Zwischenregistern noch etwas abknapsen. Bedenkt man, dass ZL auch ein gewöhnliches Register ist (und zwar R30) und dass man es auch als solches verwenden kann, können wir eines der Zwischenregister einsparen. Der Code:

MOV PC,(W)	(7) Ld R0,Z+	ind, oberes Byte, auto incr
	(8) Ld ZL,Z	ind, unteres Byte
	(10) Mov ZH,R0	kopiere nur das obere Byte
	(11) IJump	indirekter Sprung

ist erfreulicherweise einen Befehl kürzer! Und angesichts der Tatsache, dass NEXT das am häufigsten gebrauchte Stückchen Forth-Code ist, nehmen wir das auch noch gern mit. Das Register R0 dient nach wie vor als Zwischenregister.

Wir hatten versucht, den Pointern IP und SP je ein AVR-Registerpaar zuzuordnen. Beim Untersuchen der Möglichkeiten dazu hatten wir die Verwendung des AVR-Z-Registers festgelegt. Wir verwenden es als Notizblock, als einen Platz zum schnellen Zwischenspeichern, mit dem eigentlichen Ziel, einen indirekten Sprung auszuführen.

Zuordnung der Forth-Register SP und IP

Dann wird es nun also Zeit, uns zu überlegen, welche Register wir den Pointern IP und SP zuordnen können. Wir haben noch die AVR-Registerpaare W, X und Y übrig. Für den Zugriff auf die Worte (à 16 Bits) im gesamten Forth-System wäre eine Auto-Inkrement-Funktion bequem. Das AVR-System ist 8 Bits breit, so dass wir die Daten so oder so in zwei Schritten hereinholen müssen. Das Register W hat keine Auto-Inkrement-Funktion, fällt also weg. Sodann würde es uns sehr zupass kommen, wenn wir bei Zugriffen auf den Datenstack nicht nur das oberste Element (eigentlich ja das unterste, der Stack steht Kopf) erreichen könnten, sondern auch die Daten von den Elementen weiter oben auf dem Datenstack. Der AVR-Befehlssatz hat dafür vorgesorgt: Das Y- und das Z-Registerpaar können (in beschränktem Umfang) auch

Daten mit einem Extra-Offset hereinholen. Und das ohne Rechenleistung. Der Offset sitzt ganz *normal* im Opcode. Das Z-Registerpaar haben wir bereits vergeben. Bleibt uns also das Y-Registerpaar. Es folgt ein Beispiel, um das noch etwas deutlicher zu machen:

Stackposition	Wert
5	Wort 3 unteres Byte
4	Wort 3 oberes Byte
3	Wort 2 unteres Byte
2	Wort 2 oberes Byte
1	Wort 1 unteres Byte
SP → 0	Wort 1 oberes Byte

Der Datenstack-Pointer SP zeigt auf einen Platz im RAM-Speicher. Wie vereinbart, steht dort das obere Byte eines Wortes. Wir greifen etwas vor und setzen Y auf den Wert von SP. Dieser Wert wird für die momentanen Erklärungen als Basiswert festgehalten. Mit dem Maschinenbefehl

```
Ld R4,Y
```

holen wir uns das obere Byte von Wort 1 und legen es ins Register R4. Und jetzt, ohne Extraberechnung: Mit dem Maschinenbefehl

```
Ldd R5,Y+3
```

laden wir auf einen Schlag das untere Byte von Wort 2 ins Register R5. Der hier verwendete Offset von 3 wird im Opcode automatisch verarbeitet. Es liegt nun also sehr nahe, dem Forth-Datenstack-Pointer SP das Registerpaar Y zuzuordnen... Das ist am Ende jener Platz, mit welchem das gesamte Forth-System arbeitet: Das System ist stack-orientiert.

Entscheidung 6: Der Forth-Datenstack-Pointer SP wird dem AVR-Registerpaar Y zugeordnet.

Nun haben wir nur noch den Interpreter-Pointer IP übrig. Und es bleiben nicht mehr viel AVR-Registerpaare zu verteilen...

Wir hatten bereits gesehen, dass eine Auto-Inkrement-Funktion für einen Pointer außerordentlich bequem ist. Für das Registerpaar, das wir für IP verwenden wollen, wäre diese Funktion auch sehr willkommen. AVR-Registerpaare mit Auto-Inkrement-Funktion sind X, Y und Z. Davon haben wir das Y- und das Z-Paar bereits vergeben. Es bleibt uns also keine Wahl mehr!

Entscheidung 7: Der Forth-Interpreter-Pointer IP wird dem AVR-Registerpaar X zugeordnet.

Wir können nun den (beinahe) endgültigen Code für NEXT zusammenstellen. Die Zeilennummerierung wurde unmittelbar vom Code oben übernommen. Jene Zeilennummern, die hier nicht mehr vorkommen, wurden dadurch eingespart, dass wir unsere Entscheidungen anpassten und pfiffige Code-Lösungen verwendeten.

Pseudocode	Assemblercode	NEXT–Routine
MOV W,(IP)	(1) Ld WH,X+	indirekt, auto– increment
INC IP	(2) Ld WL,X+	indirekt, auto– increment
MOV PC,(W)	(5) Movw ZL,WL	kopiere Pointer
	(7) Ld RO,Z+	ind, oberes Byte, auto incr
	(8) Ld ZL,Z	ind, unteres Byte
	(10) Mov ZH,RO	kopiere nur das obere Byte
	(11) IJump	indirekter Sprung

Das Einzige, was wir uns noch überlegen müssen, ist die Frage, ob unsere Wahl des AVR–Registerpaares W für das Forth–Register W die richtige Wahl war. Um das beurteilen zu können, müssen wir im Forth–Prozess noch einen Schritt weitergehen, nämlich zur Behandlung von High–Level–Worten.

Die Behandlung von High–Level–Worten

In einem High–Level–Wort, das aus einer :-Definition besteht, enthält das Parameterfeld der Definition eine Liste mit Adressen (mit CFAs, wie eingangs bereits erklärt) von anderen Worten, die ausgeführt werden sollen. Die Verarbeitungs–Routine dieser High–Level–Forth–Worte muss diese Adressenliste in der richtigen Reihenfolge abarbeiten. Das geschieht im Adressen–Interpreter DOCOL. DOCOL verwendet den Interpreter–Pointer IP auf dieselbe Weise, wie der AVR–Programmzähler PC die Maschinenbefehle verarbeitet. Anders gesagt, der IP läuft durch die Adressenliste im Parameterfeld so, wie der Programmzähler durch die Folge von Maschinenbefehlen läuft. Wenn die CFA auf eine andere High–Level–Definition zeigt, muss IP verwendet werden, um durch die neue Liste von Adressen zu laufen. Den alten Wert von IP bewahren wir auf dem Returnstack (daher der Name) auf, um später wieder zurückkehren zu können. Damit wird IP wieder zur Verarbeitung der neuen Liste frei. Auf diese Weise bildet der Returnstack eine Erweiterung von IP, so dass es möglich wird, ein weiteres High–Level–Wort aus einem anderen heraus aufzurufen. Die maximale Anzahl von Worten, die sich eines aus dem anderen heraus aufrufen können (die Nesteltiefe) hängt ausschließlich vom Platz auf dem Speicher ab, den der virtuelle Forth–Computer dem Returnstack zur Verfügung stellt.

Am Ende einer :-Definition muss die Kontrolle wieder an das aufrufende Wort zurückgegeben werden. Das besorgt der EXIT–Code. Die Rückkehradresse hatten wir auf dem Returnstack (dessen Bezeichnung jetzt klar wird) aufbewahrt. Wir können den gesamten Sachverlauf in Pseudocode fassen:

DOCOL:	Das W–Register zeigt auf die CFA des momentan gerade ausgeführten Wortes
DEC RP	Schaffe Platz auf dem Returnstack.
MOV (RP),IP	Setz die Adresse des als nächstes auszuführenden Wortes auf den Returnstack; wir benötigen IP zum Durchlaufen der neuen CFA–Liste.
INC W	Lass W auf die PFA des laufenden Wortes zeigen, auf die erste Adresse in der Liste mit CFAs.
MOV IP,W	Kopiere diese Adresse des ersten Wortes aus der neuen CFA–Liste nach IP, bereit zur Verwendung durch NEXT.
NEXT	Führe den Code für NEXT aus (siehe oben), um dieses neue Wort auszuführen.

Man beachte, dass wir zwei dieser Befehle durch einen einzigen ersetzen können:

```
DEC RP
MOV (RP),IP → PUSH IP
```

Der PUSH–Befehl erledigt auf einen Schlag beide Dinge zugleich.

Wenn wir auf diese Weise die gesamte Liste von High–Level–Worten durchgearbeitet haben, müssen wir wieder dorthin zurückkehren, wo wir hergekommen sind. Dafür sorgt der EXIT–Code oder das ; am Ende unserer Forth–Definition.

EXIT:	Die Rückkehradresse liegt auf dem Returnstack.
MOV IP,(RP)	Stelle vom Returnstack aus die Adresse des als nächstes auszuführenden Wortes wieder her.
INC RP	Gib den frei gewordenen Platz auf dem Returnstack wieder zurück.
NEXT	Führe NEXT aus, um dort fortzufahren, wo wir nach Ausführung dieses Wortes verblieben waren.

Auch hier können wir zwei dieser Befehle durch einen einzigen ersetzen:

```
MOV IP,(RP) → POP IP
INC RP
```

Der POP–Befehl erledigt auf einen Schlag beide Dinge zugleich.

Nun übersetzen wir den oben stehenden Pseudocode in AVR–Maschinenbefehle. Ein funktionierendes, im klassischen Sinne aufgebautes AVR–Forth rückt näher!



Wir hatten IP und SP je zu je den AVR-Registerpaaren X und Y zugeordnet. Wir gehen jetzt auf den verwendeten Pseudocode zur Umsetzung von High-Level-Worten nach Befehlen in AVR-Maschinensprache ein. Werfen wir noch mal einen Blick auf das bisher Gesagte.

Legen wir gleich mit dem Code los:

DOCOL-Pseudo		DOCOL-Assembler	
PUSH IP	(1)	Push XL	Bewahre IP auf dem Returnstack auf
	(2)	Push XH	
INC IP	(3)	Adiw WL,2	Zeig auf das nächste Wort
MOV IP,W	(4)	Mov XL,WL	Kopiere Pointer
	(5)	Mov XH,WH	
NEXT			Führe den Code für NEXT aus; Fahre mit dem nächsten Wort fort

Zur Aufbewahrung von IP auf dem Returnstack benötigen wir zwei Befehle: Unser Prozessor ist ja nur acht Bit breit. Aus demselben Grund erhöhen wir W um zwei, damit es auf das nächste Wort (= 2 Bytes weiter oben) zeigt.

EXIT-Pseudo		EXIT-Assembler	
POP IP	(1)	Pop XH	Stelle IP vom Returnstack aus wieder her
	(2)	Pop XL	
NEXT			Führe den Code für NEXT aus; Weiter dort, wo wir verblieben waren

Auch hier zwei Befehle, um IP wiederherzustellen. Man beachte, dass es nicht nötig ist, W wiederherzustellen. W bekommt in NEXT einen neuen Wert.

Die richtige Wahl getroffen?

Wir müssen uns noch eine Antwort auf die Frage verschaffen, ob die Wahl des AVR-Registerpaares W für das Forth-Register W die **richtige** Wahl gewesen ist. Das W-Register wird zur Markierung des Speicherplatzes für das nächste Wort verwendet.

In diesem AVR-Forth verwenden wir W ausschließlich in High-Level-Definitionen zur Anzeige des nächsten Wortes. Dazu muss W innerhalb des Codes für NEXT (siehe dort) einen Wert bekommen und dieser Wert muss beim Verarbeiten von DOCOL zur Verfügung stehen. Zur Verfügung stehen muss er, wie wir später sehen werden, auch in DOCOL-artigen Konstruktionen wie DOCON und DOVARIABLE. Außerhalb dieser in Maschinensprache gehaltenen Teile wird weder das Forth-Register W noch das AVR-Registerpaar W benötigt. Wir können also das Registerpaar W als ein ganz *gewöhnliches* Registerpaar verwenden. In den Maschinensprachteilen machen wir von der Möglichkeit Gebrauch, dass wir bei Registerpaaren auf einen Schlag gleich noch einen konstanten Wert (hier 2, die Anzahl von Bytes in einem Wort) hinzuaddieren können, um das nächste Wort zu erreichen. Das brauchen wir also nicht in zwei getrennten Byte-Additionen zu tun. Der verwendete Befehl, **Adiw**, ist auf die Registerpaare W, X, Y und Z und nur auf diese anwendbar.

Da wir die Registerpaare X, Y und Z bereits zugeordnet haben, ist das W-Registerpaar das letzte Paar, bei welchem dieser **Adiw**-Befehl möglich ist. (Befehlssatz: Siehe Atmel-Website.)

Die Wahl des AVR-Registerpaares W für das Forth-Register W ist sicher eine gute Wahl.

Entscheidung 8: Das Forth-Register W wird dem AVR-Registerpaar W zugeordnet.

So, die Basis ist damit gelegt. Wir zählen die getroffenen Entscheidungen noch einmal auf:

Forth-Register	AVR-Register
RP	SP
W	WL und WH (= R24 und R25)
IP	XL und XH (= R26 und R27)
SP	YL und YH (= R28 und R29)
Hilfsregister für indirekte Sprünge	ZL und ZH (= R30 und R31)

Struktur hineinbringen

Um dem Ganzen etwas Struktur zu verleihen, treffen wir folgende Vereinbarung über die Verwendung von Registern innerhalb des Datenstacks. Alle Datenstack-Aktionen ordnen wir ab R23 (gleich unter dem Registerpaar W) nach unten zu an, wobei R23 das obere Byte eines Wortes enthält und R22 das untere und so weiter. Benötigt ein Wort beispielsweise zwei Stack-Einträge, dann kommen wir zu folgendem Bild (wir nehmen das Wort **AND** als Beispiel):

AND (n2 n1 -- n3)

Input	Output	AVR-Register
0	0	R19
1 n2 unteres Byte	1 n3 unteres Byte	R20
2 n2 oberes Byte	SP → 2 n3 oberes Byte	R21
3 n1 unteres Byte	3	R22
SP → 4 n1 oberes Byte	4	R23
5	5	

Der Maschinencode wird nun:

Code_Aand:

Ld R23,Y+	Hole oberen n1-Teil herein
Ld R22,Y+	Hole unteren n1-Teil herein
Ld R21,Y+	Hole oberen n2-Teil herein
Ld R20,Y+	Hole unteren n2-Teil herein

And R21,R23	AND oberer Teil
And R20,R22	AND unterer Teil

St -Y,R20	Speichere unteren n3-Teil ab
St -Y,R21	Speichere oberen n3-Teil ab

Next

Benötigen wir einen Vorgang, bei dem Daten hereingeholt werden müssen (indirekt, also über einen Pointer), dann verwenden wir dafür das Registerpaar Z. Eine Zuordnung dieses einen Stackeintrags zu einem Registerpaar aus der Reihe R23 ... entfällt dann.

Nehmen wir als Beispiel den Code für das Wort @

```
@ ( adresse -- wert )
```

Code_At:

```
Ld ZH,Y+   Hole oberen Adressteil herein
Ld ZL,Y+   Hole unteren Adressteil herein
```

```
Ld R21,Z+  Hole oberen Teil des Wertes
            von dieser Adresse herein
Ld R20,Z+  Hole unteren Teil des Wertes
            von dieser Adresse herein
```

```
St -Y,R20  Speichere unteren Teil des
            Wertes ab
```

```
St -Y,R21  Speichere oberen Teil des Wer-
            tes ab
```

Next

Wir reservieren vorläufig R23 ... R12 für sechs Stack–Einträge. Im Übrigen sind Forth–Worte, die mehr als sechs Stack–Einträge benötigen, nicht mehr einfach zu nennen!

Inzwischen sind wir einem guten Teil von AVR–Maschinencode begegnet. Bevor wir jedoch zu einem funktionierenden Forth–System kommen, müssen wir auch die Hardware–Seite noch unter die Lupe nehmen.

Hardware–Umsetzung

Der AVR–Prozessor von Atmel ist ein Prozessor, der mit der Harvard–Speichereinteilung arbeitet. Das bedeutet, dass der Speicherplatz für das Programm vom Speicherplatz für die Daten ganz und gar getrennt ist (man vergleiche die Datenblätter der verschiedenen Prozessoren auf der Website von Atmel). Das stellt an den Entwurf unseres Forth–Systems spezielle Anforderungen. Die Codierungsmöglichkeiten für Forth sind:

- Indirekte Fädelung (klassisches Modell)
- Direkte Fädelung
- Unterprogramm–Fädelung

Die **einzig** mögliche Art der Implementation auf dem AVR–Prozessor ist das klassische Forth–Modell, die indirekt gefädelte (indirect threaded) Version.

Warum ist das so? Die indirekt gefädelte Version geht von einer Wortliste aus, die ausschließlich Zeiger (Pointer) enthält, welche auf einen Platz (die CFA) verweisen, der wiederum einen Zeiger auf den Maschinencode (im internen Flash–Speicher) enthält. Neue Worte, die an die Wortliste angefügt werden, bestehen ausschließlich aus Zeigern. Nirgends wird direkt auf Maschinencode verwiesen. Sehen Sie sich den schon besprochenen Codeteil für NEXT daraufhin noch einmal an. Die Werte, welche im Datenspeicher landen, sind **echte** Daten (die Pointer). Das Ablegen von Daten und das anschließende Ausführen dieser Daten so, als wäre es Maschinencode,

ist **nicht** möglich; Maschinencode kann einzig und allein vom internen Flash–Speicher aus ausgeführt werden.

Im folgenden wird auf die Pointer–Struktur der drei Fädelungstypen näher eingegangen. Bei einem Prozessor der Serie 8051, das werden Sie danach begreifen, sind alle drei Typen möglich, indem man nämlich die beiden Speicherbereiche gewissermaßen *aufeinander* legt. Bei der AVR–Prozessorserie geht das ganz bestimmt nicht und wir bleiben an das indirekte Verdrahtungsmodell gebunden. (Das finden wir aber auch nicht sonderlich schlimm. Das ist das wahre *Basismodell*.)

Wir sehen uns nun die Hardware–Ausstattung des Systems an. Das System hat eine Standard–Ausführung und besteht aus einem AVR–Prozessor, einem Adress–Latch und einem RAM–Speicher. Man beachte, dass das übliche EPROM fehlt. Das sitzt als Flash–Speicher im Prozessor selbst. Und in diesem Flash–Speicher sitzt nun wiederum unser Vorhaben, das AVR–Forth. Wir gehen davon aus, dass ein serieller Anschluss in Form eines im Prozessor–Chip vorhandenen UARTs zur Verfügung steht. Auch müssen wir natürlich einen AVR–Prozessortyp wählen, der externes RAM ansteuern kann (das können sie durchaus nicht alle).

Indirekt gefädelte, direkt gefädelte, unterprogrammgefädelte???

Bevor wir die verschiedenen Fädelungsarten (threads) besprechen, brauchen wir noch eine Methode, :-Definitionen in Forth ins RAM zu setzen. Wie wir das tun, ist im Augenblick nicht so wichtig. Wir gehen davon aus, dass es ganz *normal* möglich ist. Ausgangspunkt ist die Definition

```
:   QUADRAT   DUP   *   ;
```

Irgendwo in unserem Forth–System verwenden wir das Wort QUADRAT. Um das zu Sagende besser ins Bild zu bringen, nehmen wir ein paar fiktive Speicheradressen an, wo die verwendeten Worte abgelegt sind. Den **Aufruf** von QUADRAT finden wir beispielsweise an der Hex–Adresse 9812. Die **Definition** von QUADRAT finden wir an Adresse 834F. Das Wort DUP ist ein Low–Level–Codewort an Adresse 0426 im Flash–Speicher und * ist eine High–Level– (eine :-) Definition an Adresse 1278 im Flash–Speicher. Für ein traditionelles indirekt gefädeltes Forth können wir jetzt den folgenden Speicherausgang zeichnen. Abbildung 1 zeigt ein Beispiel, bei dem je zwei Bytes eine Zelle bilden.

Wir nehmen an, dass der IP jetzt auf die Stelle 9812 zeigt, so dass also bei Aufruf von NEXT die Definition QUADRAT ausgeführt wird.

Was geschieht nun genau? Wir verwenden den einfachen Befehlssatz, den wir in ganz zu Anfang eingeführt hatten (man achte auf die Zahlenbeispiele):



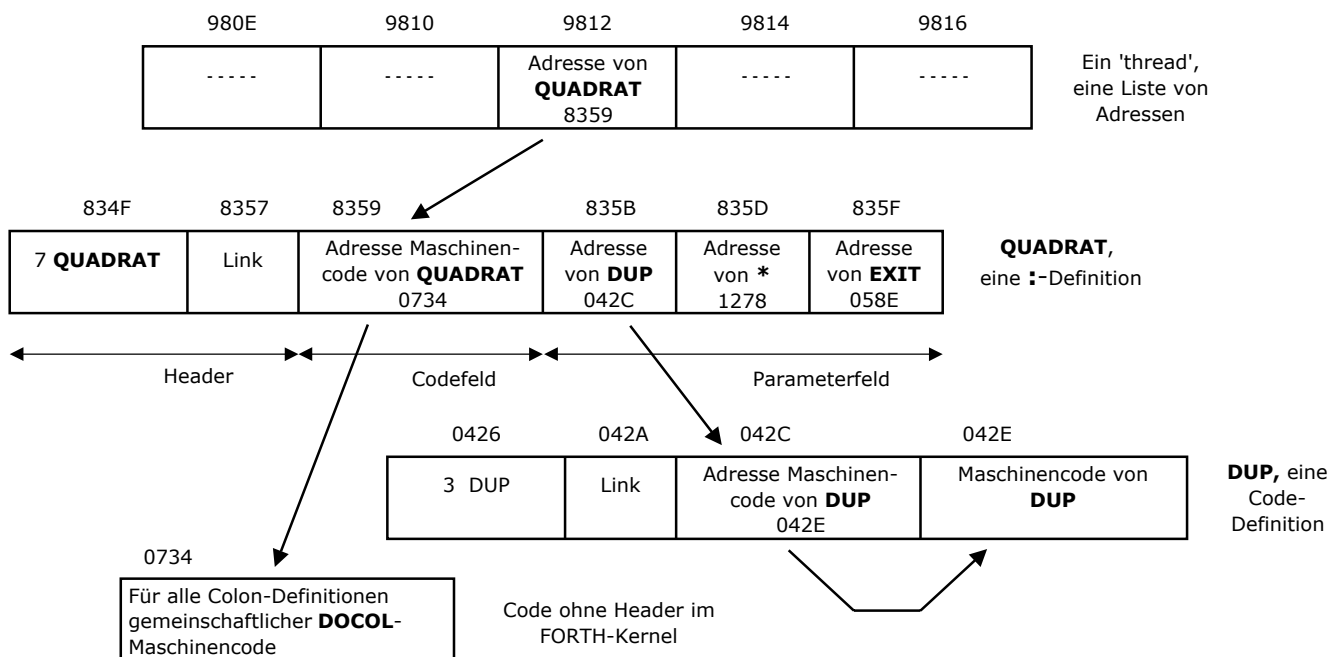


Abbildung 1: Indirekte Fädelung

NEXT: IP zeigt auf das als nächstes auszuführende Wort an Adresse 9812

MOV W, (IP) Kopiere den Inhalt von IP (=8359), die CFA des als nächstes auszuführenden Wortes, ins Register W.

INC IP Setze den IP so, dass er auf das Wort NACH dem momentan bearbeiteten zeigt, um da unmittelbar weiterzuarbeiten (=9814).

MOV Z, (W) Führe den Maschinencode aus, dessen Adresse jetzt im W-Register (=0734) sitzt. Verwende Z als Zwischenregister.

JMP (Z) Gehe über einen indirekten Sprung in den Maschinencode.

Wäre **QUADRAT** eine Maschinencode-Definition gewesen, wären wir nun fertig. Das Stückchen Maschinencode wird ausgeführt und wir springen auf das nächste **NEXT** zurück, das uns zu Platz 9814 leitet. **QUADRAT** ist jedoch ein High-Level-Wort. Es enthält keinen Maschinencode, sondern einen thread, eine Liste von Adressen. Um diese Definition ausführen zu können, muss der Interpreter an Adresse 835B, dem Parameterfeld von **QUADRAT**, aufs Neue gestartet werden. Wir müssen aber auch den alten Wert von IP sichern, um da dann fortfahren zu können, wenn das Wort **QUADRAT** abgearbeitet ist. Da wir gerade einen indirekten Sprung ausgeführt haben, müssen wir hierzu auf ein Stückchen Maschinencode stoßen. Das ist der Code für **DOCOL** (lesen Sie dort noch einmal

kurz nach). Dieses Stückchen Maschinencode ist für jede High-Level-Definition dasselbe. Wir wiederholen den Pseudocode:

DOCOL:

PUSH IP Sichere den momentanen IP auf dem Returnstack (=9814)

INC W Zeige auf das nächste Wort (=835B)

MOV IP, W Kopiere diesen Pointer als neuen IP

NEXT Fahre beim nächsten Wort fort

Wir sind nun bei der Ausführung der Definition **QUADRAT** eine Ebene tiefer gerutscht. Die Situation, in der wir uns jetzt befinden, ist eigentlich dieselbe wie an unserem Ausgangspunkt, das Ausführen einer Reihe von Definitionen (Worten). Wir sind auf indirekte Weise hierher gelangt, über Zeiger (Pointer). Diese Vorgehensweise ist die traditionelle altmodische Art von Forth. Das ist die indirekt gefädelte Methode.

Es ist klar, dass wir auch noch zurück müssen. Haben wir das Wort ***** auf dieselbe Weise ausgeführt, so stoßen wir auf das Wort **EXIT**. Dieses Wort ist der Returnbefehl, der in den Speicher gesetzt wird, wenn der Forth-Compiler auf das Wort **;** (Ende der Definition) trifft. **EXIT** macht das Umgekehrte von **DOCOL**. In Pseudocode sieht das so aus:

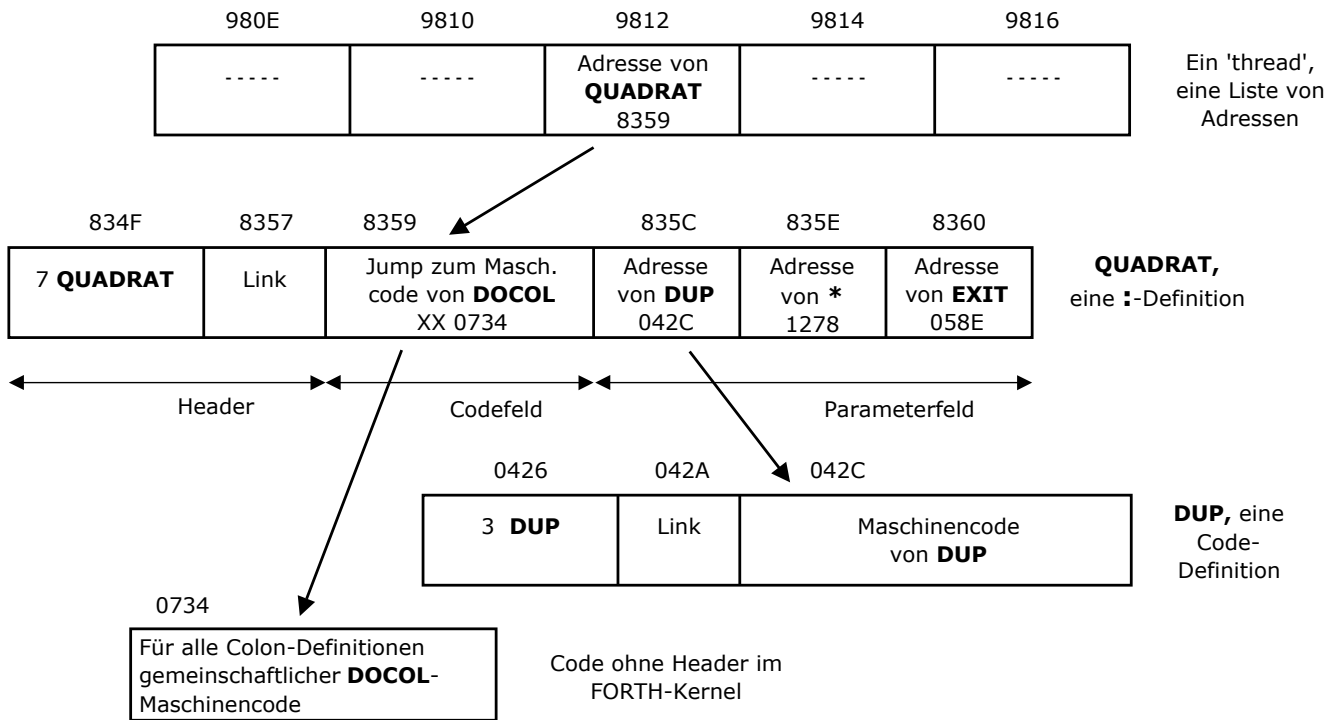


Abbildung 2: Direkte Fädung

EXIT: Die Rückkehradresse steht auf dem Returnstack

POP IP Stell die Adresse des als nächstes auszuführenden Wortes wieder her (=9814)

NEXT Fahre dort fort, wo wir nach dem Ausführen des Wortes **QUADRAT** verblieben waren

NEXT: IP zeigt auf das als nächstes auszuführende Wort auf Adresse 9812

MOV W, (IP) Kopiere den Inhalt von IP (=8359), die CFA des als nächstes auszuführenden Wortes, ins W-Reg.

INC IP Setze den IP so, dass er auf das Wort nach dem momentan bearbeiteten zeigt (=9814), um da unmittelbar weiterarbeiten zu können.

JMP (W) Springe direkt zum Maschinencode (=0734)

Die charakteristischen Merkmale eines indirekt gefädelten Forths: Jedes Forth-Wort hat ein Codefeld von genau einer Zelle (hier 2 Bytes). High-Level-(:-) Definitionen compilieren für jedes Wort genau eine Zelle in die Definition. Der Forth-Interpreter muss doppelt indirekt arbeiten, um die Adresse des schließlich auszuführenden Maschinencodes zu finden, erst über IP, danach dann über W.

Direkt gefädelter Code

Der Unterschied ist klein: Beim direkt gefädelten Code steht im Codefeld keine Adresse, sondern Maschinencode. Oft hat der Code die Form von **JMP** adresse oder **JSR** adresse, aber es kann natürlich auch eine vollständig ausgeschriebene Routine sein, die auf **NEXT** endet (siehe Abbildung 2).

Da **NEXT** nun eine Indirektionsstufe weniger auszuführen braucht, wird es ein Stückchen einfacher:

Der letzte Befehl in diesem Beispiel ist ein **JMP**. Und hier liegt zugleich das Problem unseres AVR-Forth-Systems: Der Prozessor wurde so entworfen, dass im RAM-Speicher nichts anderes als Daten liegen können. Der auszuführende Maschinencode findet sich einzig und allein im FLASH-Programmspeicherbereich des Prozessors. Und sollten Sie die Maschinencodebefehle ins RAM setzen, dann könnten diese auf keine einzige Art ausgeführt werden. Der Trick des Aufeinanderlegens der beiden Speicherplatzarten (Programmbereich und Datenbereich), so wie er bei der Prozessorserie 8052 üblich ist, greift hier nicht. Schade, mit dem AVR-Prozessor ist keine direkt gefädelte Forth-Version möglich!

Unterprogrammgefädelter Code

Eigentlich können wir es nun schon erraten: Ein unterprogrammgefädelt Forth besteht aus einer Anzahl



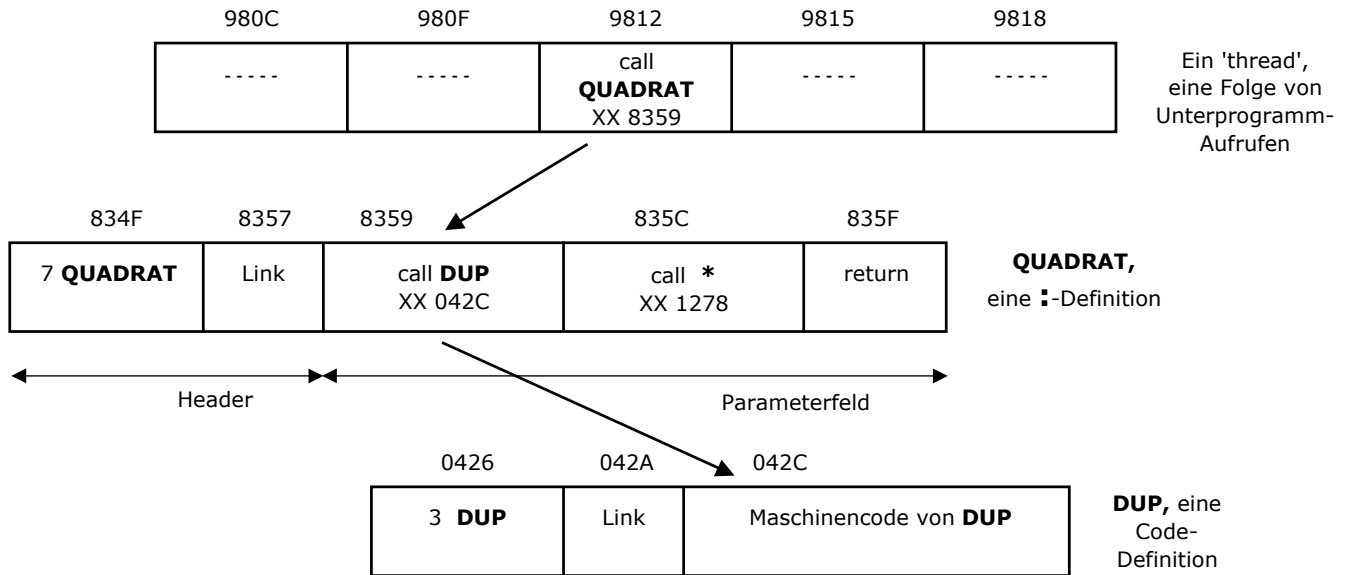


Abbildung 3: Unterprogramm-Fädelung

von hintereinander platzierten Unterprogrammaufrufen (Calls) der verwendeten Worte.

Aber wenn wir die Calls im RAM unterbringen, können wir auch hier leider keinen Maschinencode ausführen. Abbildung 3 zeigt der Deutlichkeit halber einen Speicherauszug.

In der vorigen Folge haben wir gesehen, dass bei Verwendung eines AVR-Prozessors die klassische, indirekt gefädelte Methode für unser Forth-System die einzige in Frage kommende Möglichkeit ist. In der vorliegenden Folge überlegen wir uns, wie das eine oder andere zu implementieren ist. Wir gehen von einer Standard-Ausführung aus. Die besteht aus einem AVR-Prozessor, einem Adress-Latch und einem RAM-Speicher. Das übliche EPROM fehlt; das sitzt bereits als FLASH-Speicher im Prozessor selbst. Wir gehen davon aus, dass im Prozessorchip ein serieller Anschluss in Form eines UARTs zur Verfügung steht. Natürlich müssen wir auch einen AVR-Prozessortyp wählen, der überhaupt externes RAM ansteuern kann (das können sie längst nicht alle). Das im Prozessor eingebaute RAM verwenden wir auch (dadurch wird ein kleiner Teil des externen RAMs außer Kraft gesetzt). In unserem Beispiel wählen wir einen ATmega162-Prozessor in Verbindung mit einem RAM von 32 Kilobyte (Schaltbild siehe Abbildung 4).

Stackgröße

Zuerst müssen wir ein paar Annahmen machen. Wie groß sollen wir den Datenstack wählen, und wie groß den Returnstack? Wenn wir sie alle beide im internen RAM des Prozessors unterbringen können, dann haben wir auf jeden Fall die allerschnellste Konfiguration. Der Zugriff auf das interne Prozessor-RAM kostet nur 1 Maschinenzklus, während ein Zugriff auf das externe RAM 2 Zyklen

Man beachte, dass sich durch die Verwendung von Unterprogrammaufrufen die Adressen, an denen die Calls stehen, verändert haben! Der CALL-Befehl selbst nimmt einen gewissen Platz ein.

Unsere Wahl ist einfach:

Entscheidung 9: Wir wählen ein indirekt gefädelt AVR-Forth!

verlangt. Hier bietet sich der erste Zeitgewinn an. Für ein Standard-Forth für Experimentierzwecke reichen uns fürs Erste ein Datenstack von 32 Worten und ein Returnstack von ebenfalls 32 Worten. Der ATmega162 hat ein internes RAM von 1024 Bytes, also genügend Platz, um tatsächlich beide Stacks intern aufzunehmen.

Entscheidung 10: Der Datenstack und der Returnstack sind je 32 Worte lang und befinden sich im internen RAM des Prozessors.

Über den Forth-Code

Dieses experimentelle Forth ist nach dem FIG-Modell (aus dem Jahre 1982!) aufgebaut. Das deshalb, weil das Modell an zahlreichen Literaturstellen beschrieben wurde. Ein erschöpfendes Buch ist *The Forth Encyclopedia* von Mitch Derick und Linda Baker. Das FIG-Modell geht jedoch von einem Forth aus, das vollständig im RAM läuft. An vielen Stellen im System werden da Pointer angepasst und Code-Worte hinzugefügt. In Folge 6 unserer Serie haben wir gesehen, dass das in der AVR-Umgebung nun gerade nicht geht. Mit einigem Erfindungsgeist lässt sich hierfür aber eine Lösung finden. Wenn wir uns eine Methode ausdenken, bei der alle Low-Level-Codeworte ins Flash wandern und alle High-Level-Worte ins RAM, dann können wir uns dem Forth in normaler High-Level-Art nähern. Das Selbermachen von Code-Worten ist dann jedoch nicht möglich. Es geht

also darum, einen wohldurchdachten Kernel zu entwickeln, der bereits alles enthält, was wir haben wollen!

Der Low-Level-Kernel

Woraus besteht der Kernel-Code des Forth-Systems nun eigentlich? Richtig: Aus CODE. Wir müssen uns gut klarmachen, dass ausschließlich Code (Maschinenbefehle) ausgeführt werden kann. Dass die Forth-Codeworte auch einen Namen haben, ist nebensächlich, aber angenehm. Und wenn wir jetzt alle Maschinenbefehle auf einen Haufen ins Flash werfen, und alle Namen ins RAM bugsieren? Dann entsprechen wir auf jeden Fall dem AVR-Prozessor-Modell. Aber mit Code-Worten ohne Namen haben wir noch kein Forth-System. Auf die eine oder andere Art müssen wir doch die Namen verarbeiten können. Die Auflösung: Wir stellen eine vorgefertigte Liste auf, die nur die Namen der Code-Worte enthält, setzen die in den Flash-Speicher und kopieren beim Hochfahren des gesamten Systems sämtliche Codewort-Namen ins RAM. Die Forth-Worte können dann in gewohnter Weise dadurch angesprochen werden, dass man ihre Namen im RAM-Speicher aufsuchen lässt und anschließend den zugehörigen Maschinencode im Flash-Speicher zur Ausführung bringt. Der zugehörige Maschinencode wird über einen Pointer, die CFA, erreicht. Zur Verdeutlichung die Darstellung des Wortes DUP (siehe Abbildung 5).

Der Header und das Codefeld befinden sich im RAM-Speicher, wobei das Codefeld auf ein Stückchen Maschinencode auf Adresse 0530 im Flash-Speicher zeigt. Wir sehen hier auch, dass neben dem Namen des Forth-Wortes auch die Linkfeld-Adresse des vorhergehenden Wortes und natürlich die CFA im RAM-Speicher liegen müssen. Der Maschinencode des Parameterfeldes liegt im Flash-Speicher. Damit haben wir ein Standard-Forth erzeugt, mit der Besonderheit, dass die Bytes nicht allesamt hintereinander im RAM-Speicher liegen, sondern sich auf zwei Bereiche verteilen, die ihren Platz in zwei verschiedenen, voneinander getrennten Speicherteilen haben. Das geht problemlos, solange nur alle Zeiger auf den richtigen Platz verweisen. Es sollte klar sein, dass die Suchfunktion (ein Low-Level-Codewort im Flash), die die Forth-Worte in der Wortliste aufsucht, ihre Suchaktion vollständig im RAM-Speicher (dem Datenaufbewahrungsort) ausführt. Bevor das alles richtig ineinanderpasst, ist noch einiges an Knobelarbeit zu verrichten.

Der High-Level-Kernel

Neben den Low-Level-Codeworten besteht der Forth-Kernel auch aus High-Level-Worten. Zunächst einmal müssen wir uns vergegenwärtigen, dass ein High-Level-Wort ein echtes Forth-Wort ist, das also aus Daten besteht. Für Maschinencode gibt es hier nichts zu tun. Die Bytes, die da stehen, sind ganz und gar reine Daten. Es braucht hier kein Maschinencode ausgeführt zu werden.

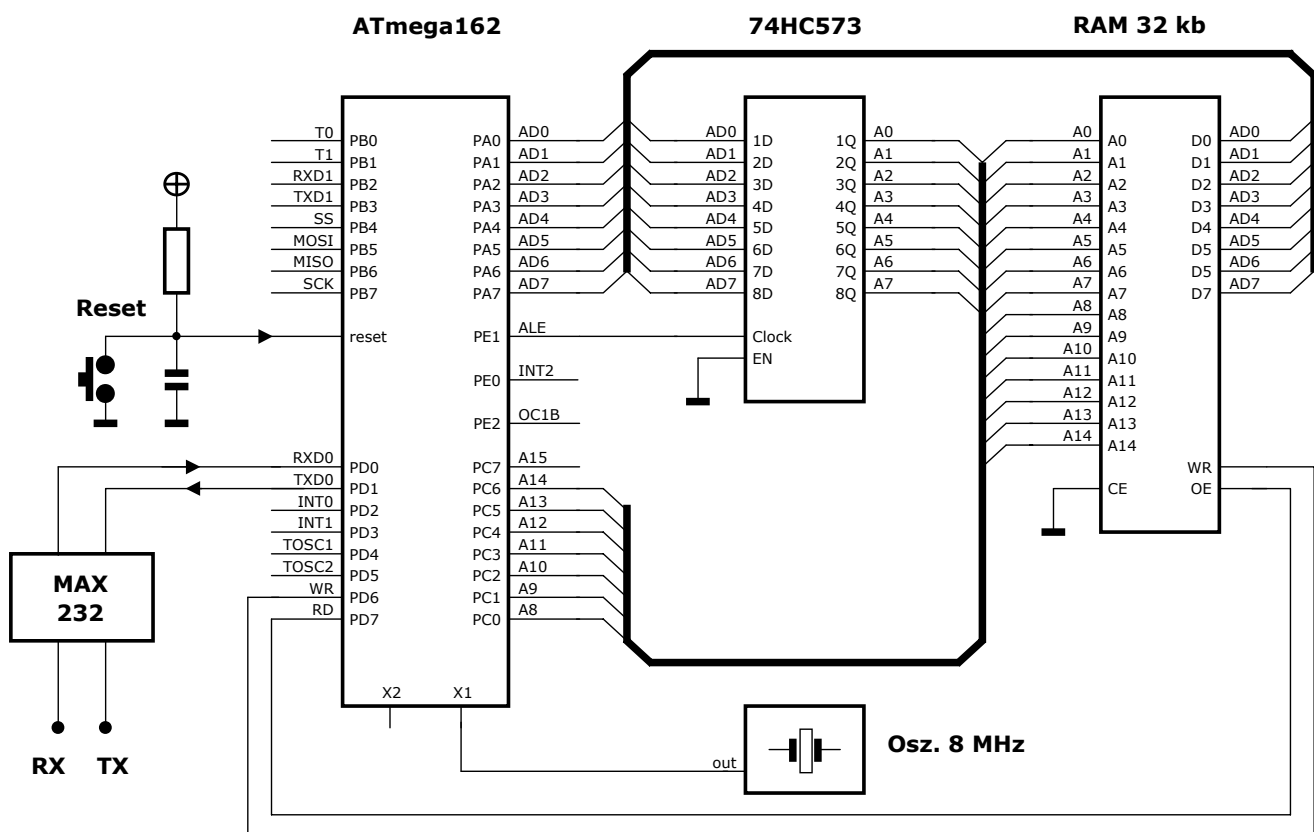


Abbildung 4: Schaltbild des Versuchsaufbaus



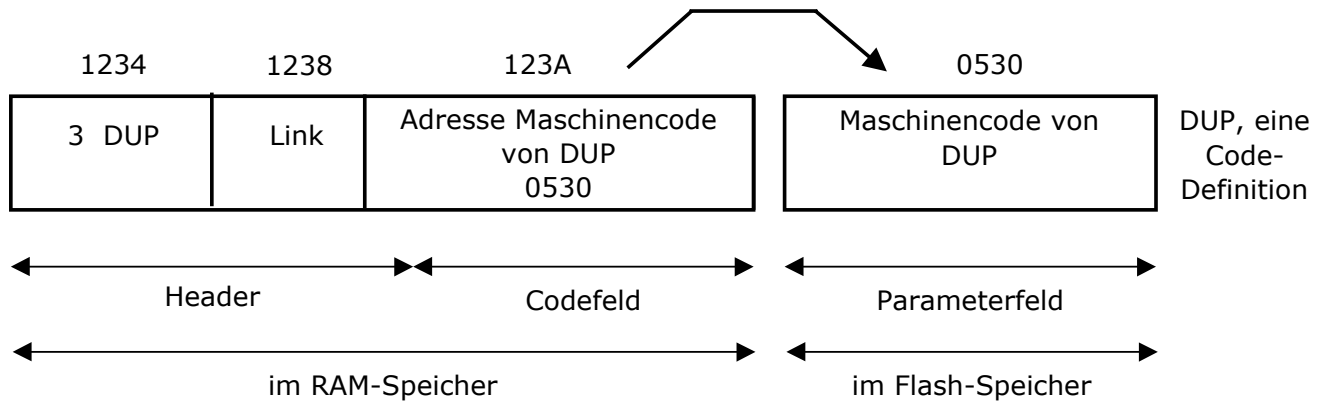


Abbildung 5: Das Wort DUP, teils im RAM, teils im Flash

Die Forth-Maschine ist ja eine virtuelle Maschine. Die eigentliche Arbeit wird von einem ganz kleinen Stück speziellem Maschinencode geleistet, von NEXT. Wir können in den Kernel sehr wohl eine Anzahl von High-Level-Worten aufnehmen, die dann aber als vorgefertigte Daten vorliegen müssen, welche beim Hochfahren erst ins RAM zu kopieren sind. Wir müssen also in den vorgefertigten Worten alle Kopplungen (Links) und Verweise auf andere High-Level-Worte von Anfang an sorgfältig setzen. Das Ganze bekommt erst dann seinen vollen Wert, wenn es ins RAM kopiert ist. Auf dem ursprünglichen Platz im Flash lässt sich nichts damit anfangen. Hier liegt eine auserlesene Aufgabe für einen Forth-META-Compiler vor (ein META-Forth ist ein Forth, mit welchem man ein anderes Forth aufbauen kann). Eine Version, die mit einem eigenständigen AVR-Assembler (beispielsweise den von Atmel, dem Hersteller des Atmega162) erzeugt wurde, ist natürlich auch möglich, aber das kostet viel mehr Mühe. Allerdings weiß man bei Verwendung eines Assemblers bis auf den letzten Maschinenbefehl, wie das aufzubauende Forth-System aussieht.

Abbildung 6 gibt eine Vorstellung davon, wie die Dinge nach der Kopieraktion beim Hochfahren im Speicher eingeteilt sind.

In Abbildung 6 haben wir gesehen, wie wir mit einer Kopieraktion das vorgefertigte Forth vom Flash ins RAM verfrachten können. Das liefert uns ein funktionierendes Forth-System. Mit solch einem System(chen) kann man bereits viele Dinge auf einem selbstentwickelten Hardware-Experimentieraufbau ausprobieren. Unangenehm am Quelltext einer solchen Forth-Version ist der Umstand, dass man schon beim Entwurf darauf achten muss, dass die vorgefertigten Worte im RAM am richtigen Platz zu liegen kommen. Eigentlich wäre hier ein gehörig großer RAM-Bereich eine schöne Sache: Es wäre dann Platz genug da, um den vorgefertigten Anteil an High-Level-Worten ins RAM zu kopieren. Später angefügte Definitionen fänden dann ihren Platz oberhalb des so kopierten Teils. Aber was ist, wenn ein hinreichend großer RAM-Bereich gar nicht vorhanden ist?

Können wir unser Forth-System dann immer noch verwenden? Das widerspricht doch eigentlich unserem Ausgangspunkt, nämlich dem Entwurf eines Forth-Systems ganz aus dem Nichts.

Das Unmögliche wurde doch möglich

Es ist natürlich eine Extraherausforderung, ein Forth auf einem nackten Prozessor ohne externes RAM zum Laufen zu bringen. Im ersten Moment denkt man: Das klappt nie, das funktioniert nicht. Wenn Sie jedoch das hier Geschriebene lesen, werden Sie begreifen, dass es sehr wohl gelungen ist. Aber wie können wir denn den vorgefertigten Block an High-Level-Worten auf einem nur recht kleinen Bereich an internem RAM unterbringen? Die Antwort lautet: Das tun wir nicht. Wir lassen den vorgefertigten Block einfach da liegen, wo er liegt, nämlich im Flash! ABER ????? Forth benötigt doch einen gewissen RAM-Bereich, um seine Verwaltungsaufgaben erledigen zu können?? Wo es seine Wortliste absetzen kann?? Das ist absolut richtig, jedoch die Art und Weise wie, können wir unseren eigenen Vorstellungen anpassen.

Die Wortliste intern oder extern?

Basis unseres Forth-Systems ist die Wortliste. Wenn wir mit einer Colon-Definition ein neues Wort erzeugen, müssen, wie wir wissen, die Verweise auf das, was dieses neue Wort tun soll, im RAM Platz finden. Und zwar ganz oben im schon bestehenden Teil. Und wenn wir nun dafür sorgen könnten, dass sich in der Wortliste, die im RAM liegt, nur ein einziges Wort befindet und dass alle anderen vorgefertigten Worte im Flash abgelegt sind? Wir müssen dann dafür sorgen, dass der Verweis auf die internen Worte richtig arbeitet. Und dass wir die internen Worte mit der Suchfunktion FIND finden können. Hier drängt sich wieder der Unterschied zu einem System auf, das mit einem 8052-Prozessor arbeitet: ROM (beim AVR der Flash-Speicher) und RAM können wir dort im Gesamtspeicher aufeinanderlegen. Für die Suchfunktion besteht dann kein Unterschied, ROM- und RAM-Bereich sind dann praktisch dasselbe. Schön und gut, aber wir haben einen AVR-Prozessor, und der

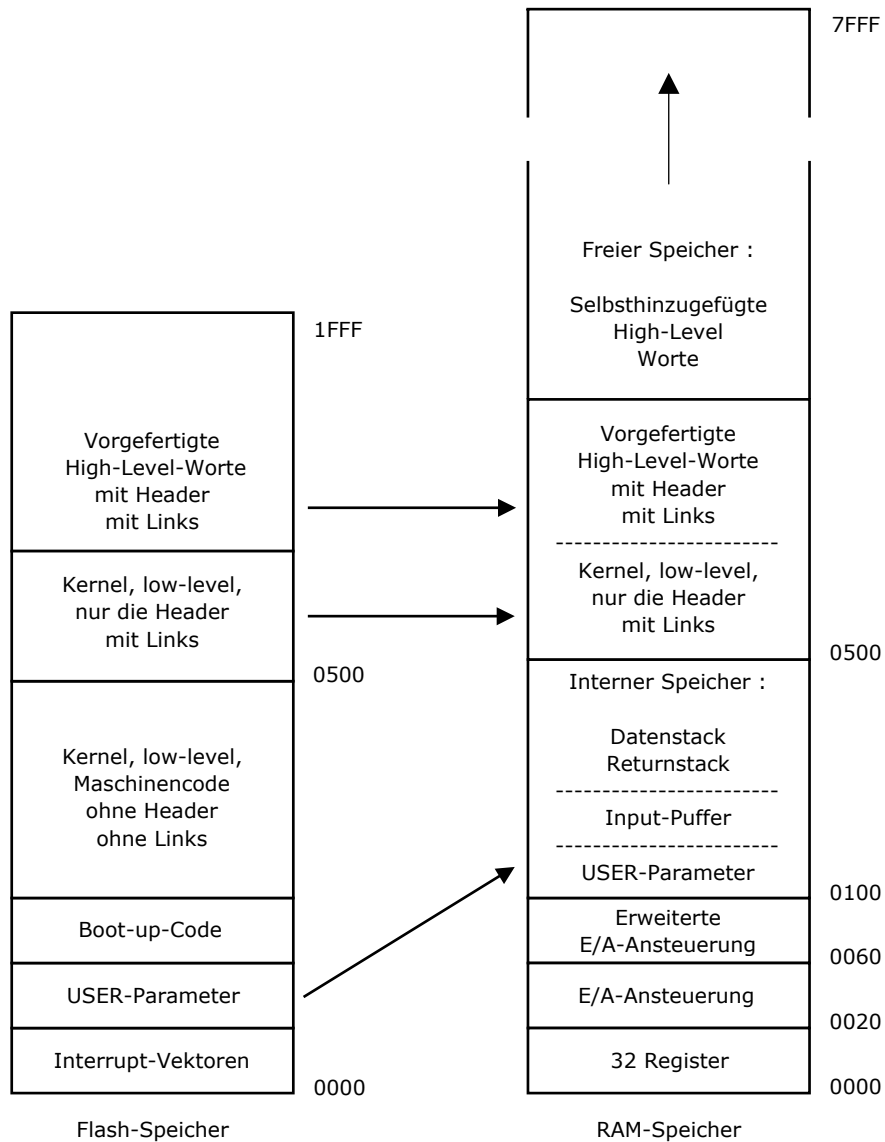


Abbildung 6: Speicheraufteilung nach der Kopieraktion

kann das nun einmal nicht. Was er aber sehr wohl kann, ist die Behandlung der beiden Speicherarten auf eine andere Weise. Im Befehlssatz des AVR's gibt es für das Einholen von Daten zwei verschiedene Maschinenbefehle (siehe Befehlssatz in der AVR-Dokumentation auf der Atmel-Website).

Das Anpassen von FIND

Wir müssen uns für die Suchfunktion FIND etwas überlegen, das deutlich macht, wo FIND suchen soll. Man denkt zunächst einmal an ein Software-Flag, das anzeigt, ob die Suche intern, im Flash des Prozessors, vonstattengehen soll, oder extern, im RAM. Alle neu zu machenden Definitionen koppeln wir mit dem Flagstand extern aneinander, und bei allen vorgefertigten Definitionen steht das Flag auf intern. Die Suchfunktion FIND muss aus zwei Teilen bestehen; der erste Teil sucht ausschließlich im

RAM, der zweite Teil ausschließlich im Flash. Als Trennung zwischen den beiden Teilen könnten wir ein Null-Link gebrauchen, das auf das Ende der Wortliste verweist. Sobald das FIND zum ersten Mal auf das Null-Link trifft, muss es auf interne Suche umschalten. Beim zweiten Mal ist das Ende der Wortliste schon erreicht. Auf diese Weise muss eine Forth-Definition gefunden werden können. Nach einigen Software-Experimenten schien das tatsächlich zu funktionieren. Wir begegnen dabei jedoch einem anderen Problem. Das Auffinden eines Wortes ist nur die eine Hälfte des Problems, dessen Ausführung ist die andere Hälfte. Auf die eine oder andere Weise muss man an der Liste von PFAs erkennen können, woraus eine Definition aufgebaut ist, wo sich das Wort befindet. Intern oder extern? Der Schlüssel dazu liegt beim Adresswert.



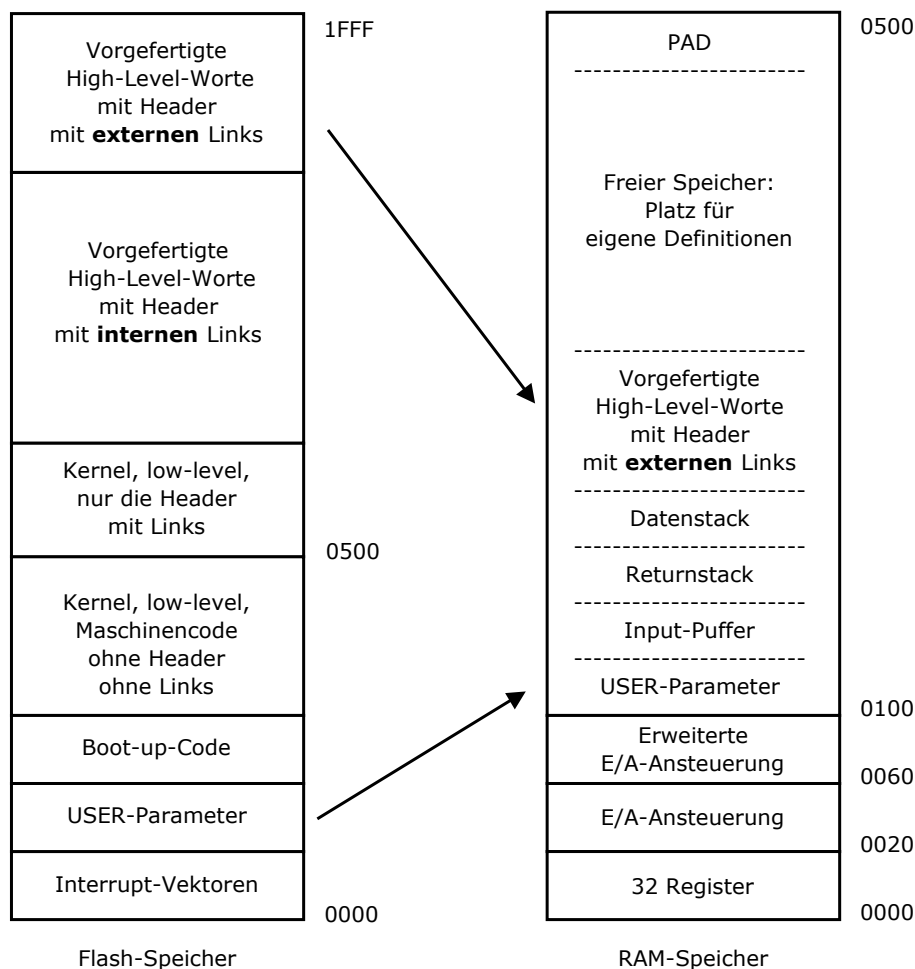


Abbildung 7: Speicherplan

Es musste noch experimentiert werden

Zunächst dachten wir daran, die internen Worte irgendwie zu markieren, beispielsweise durch ein Hochsetzen von Adressbit 15. Bei den externen Worten sollte das Bit zurückgesetzt bleiben. Alle Code-Worte, die mit dem direkten Einholen von Daten aus dem Speicher zu tun haben, müssen dann dieses Bit untersuchen und die richtige Arbeitsweise wählen. Der Gedanke war einfach, die Ausführung nicht ganz so einfach. Beim Ausarbeiten dieser Idee ergab sich jedoch eine andere Lösung: Eine Trennung in Adressbereiche.

Wenn wir uns den Speicherplan (siehe Abbildung 7) genauer betrachten, bemerken wir zwei Dinge:

Internes RAM beim MEGA162: Von 0100 bis 0500
Flash: Die Header im Kernel beginnen bei (ungefähr) 0500.

Was können wir damit anfangen? Nun ja, hier liegt die Basis einer Implementation der Innerhalb/außerhalb-Idee.

Die Implementation

Wenn wir nun unterscheiden würden zwischen Bereichen unterhalb der Adresse 0500 und Bereichen oberhalb 0500? Unterhalb der Adresse 0500 entscheiden wir uns für den Zugang zum internen RAM (mit dem Befehl `Ld`) und oberhalb der Adresse 0500 verwenden wir den speziellen Befehl `Lpm`, um uns den Zugang zum Flash zu sichern. Erst müssen wir jedoch herausfinden, welche Forth-Worte nun eigentlich mit dem Speicher direkt zu tun haben. Worte, die Daten auf dem Stack bearbeiten, fallen nicht hierunter. Die betreffenden CODE-Worte sind: `LIT` `BRANCH` `NEXT` `EXECUTE` `@` `C@` `CMOVE` (`FIND`) `COUNT` und die internen Codeteile der Worte `CONSTANT` und `USER`. Alle diese Worte holen aus der Wortliste etwas herein und stellen dort dann etwas damit an. In allen diesen Worten wurde ein Test eingebaut, der prüft, ob die angebotene Adresse kleiner als 0500 ist. Wenn ja, dann ist es eine externe Adresse im internen RAM. Wenn nein, dann ist es eine Adresse aus dem vorgefertigten Teil, der nur die Header und Links enthält. Das Ergebnis des Adresstests zeigt an, mit welchem Befehl die entsprechenden Daten hereingeholt werden sollen. Und siehe da: ES FUNKTIONIERT. Das experimentelle Forth läuft auf einem nackten Prozessor!

amforth

Matthias Trute

Überblick

amforth ist ein interaktives Forth-System für die ATmega-Mikrocontrollerbaureihe der Firma Atmel. Es ist geeignet, sowohl *im* als auch interaktiv *mit* dem Zielsystem zu arbeiten. In diesem Artikel werden die Entstehung und das Design von *amforth* beschrieben.

amforth klingt seltsam, aber *avrforth* war schon vergeben: *amforth* steht einfach für **AtMegaFORTH**.

Warum ein eigenes Forth-System aufbauen, wo es doch offensichtlich schon eines gibt¹? Warum überhaupt ein Forth neu schreiben?

Nun, *avrforth* von Daniel Kruszyna (krue.net/avrforth) ist nach seinen eigenen Worten ein colorless colorforth. Nach einigen Mails wurde klar, dass er das nicht ändern will und ich das nicht haben will. Da mir zudem sein Programmcode zu kompliziert ist, habe ich „einfach“ mein eigenes System begonnen. Wer Daniels Code betrachtet, wird rasch feststellen, dass *amforth* einige seiner Ideen übernommen hat. Die wichtigste ist vielleicht, dass *amforth* direkt in den Flash-Speicher des Controllers compiliert.

Der zweite wichtige Ratgeber war Ron Minke mit seiner Artikelserie „Forth von der Pike auf“. An dieser Stelle der Dank auch an die Redaktion der Vierten Dimension, die Vereinszeitschrift elektronisch verfügbar gemacht zu haben. Als dritte Quelle hat Stephen Pelcs Buch „Programming Forth“ meine Entscheidung, dem ANS-Forth-Standard nachzueifern, maßgeblich beeinflusst.

Nachdem ich bereits gute Erfahrungen mit dem Entwicklerportal sourceforge.net sammeln konnte, habe ich ebendort das Projekt *amforth* angemeldet und auch problemlos einrichten können. Damit standen Webspace für die Homepage des Projekts (amforth.sourceforge.net), ein Softwareverwaltungssystem (ich habe mich für Subversion entschieden), Mailinglisten und Trackingmodule bereit. Nicht zu vergessen die Downloadangebote und natürlich nette Statistiken. Alles kostenlos und gut gemacht, vor allem aber auch für den Einzelnen ohne Mühe verwendbar.

Die Entscheidung, von vornherein auf Englisch zu setzen, hat sich bewährt, wie ich an den E-Mails erkennen kann, die mich aus verschiedenen Ländern erreichen. Auch ist auf der Homepage des Projekts eine kleine Weltkarte zu sehen, auf der die Besucher der Einstiegsseite einen kleinen (oder wenn sie öfter kommen auch größeren) roten Punkt hinterlassen. Sehr interessant, selbst wenn man unterstellt, dass da nur Google-Roboter drauf zugreifen. . .

¹ Es gibt sogar viele Forths für AVR–ATmega. Sie sind in C programmiert oder kosten Geld oder haben sonst einen Makel ;=)

amforth ist unter der Gnu Public License GPL veröffentlicht. Damit stehen die Quelltexte frei zur Verfügung. Jeder kann damit machen, was er will, Änderungen müssen jedoch unter GPL wieder veröffentlicht werden.

Historie

Die ersten Schritte waren mühsam, da die Hardware keinen Mucks von sich gab. Hier half der Simulator des AVR-Studios, die kleineren und größeren Fehler zu entdecken. Dann war es soweit: Das Terminal gab den Prompt aus. Danach ging es rasch vorwärts. Jetzt wurden die Stackdiagramme und Beschreibungen der Dut-zenden an Forthworten wieder und wieder gewälzt, von Hand in die Folge der Execution Tokens oder in Assemblercode umgesetzt und auf den Controller gebrannt. Irgendwann war der Interpreter fertig: Worte konnten eingegeben werden (**accept**), wurden mit **find** gefunden und per **execute** ausgeführt.

Der nächste Meilenstein war das Wort **i!**, das eine Zelle im Flash neu beschreibt. Jetzt konnten die Compilerworte wie **if** und **again** implementiert werden. Hier half Google. Viele dieser Worte wurden bereits irgendwann von irgendwem mit Quellcode beschrieben, so dass es an einigen Stellen eigentlich nur Abschreiben war.

Eine Hürde war das Wort **does>**. Es zu implementieren, erschien mir jedoch wichtig. Der Grund ist in einem Paper von Julian V. Noble im Journal of Forth Application and Research zu finden: Finite State Machines. Diese werden in der Robotik gerne eingesetzt und seine Implementierung basiert auf diesem Wort. **does>** ist das einzige Wort, in dem Forthcode und Assemblercode gemischt vorkommen. Alle anderen sind entweder reine Colon- oder reine Assemblerworte. Zudem haben mit **does>** definierte Worte eine weitere Unschönheit: Das Flash durchläuft bei Worten, die es benutzen, mindestens einen Löschyklus. Mehr dazu weiter unten.

Die Version 1.0 von *amforth* war ein kleiner Meilenstein, denn das System hatte für mich seinen ersten Abschluss erreicht. Zudem ist die „magische“ Versionsnummer geeignet, größere Aufmerksamkeit zu wecken.

Umgebung

amforth wird unter Linux (Kubuntu 6.10) entwickelt und gelegentlich unter Windows im Simulator ausgeführt. Als Assembler nutze ich **avra** (avra.sourceforge.net und cdk4avr.sourceforge.net) zusammen mit dem aus der höheren Programmierung bekannten **make**. Der Upload auf den Controller erfolgt mit dem Programm **avrdude** (www.nongnu.org/avrdude) mit ISP durch einen kleinen STK200-kompatiblen Dongle am Parallelport oder

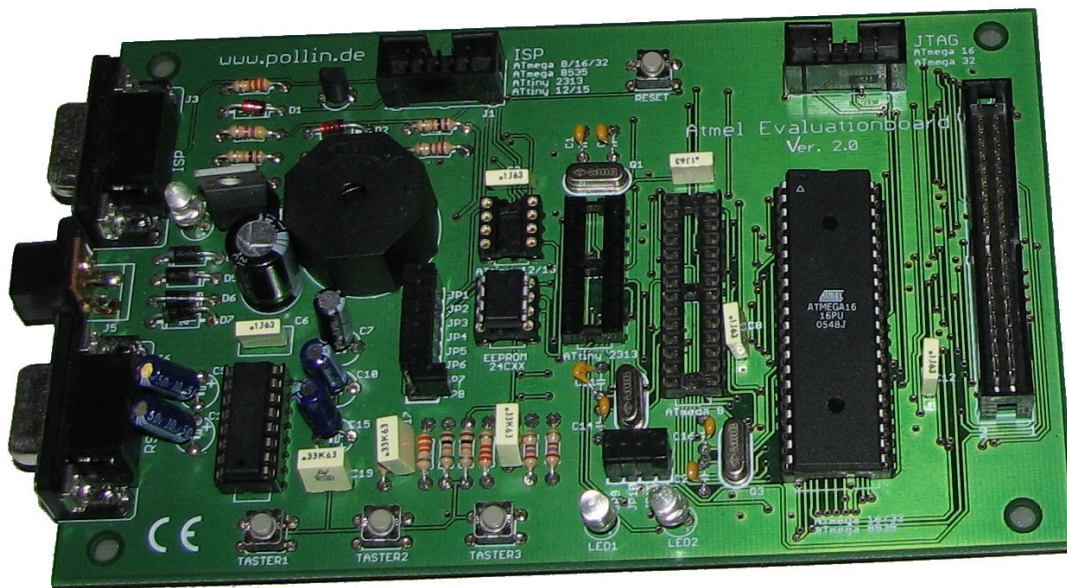


Abbildung 1: amforth wurde auf einem Pollin-AVR-Board entwickelt.

den mysmartUSB-Dongle von www.myavr.de. Letzterer hat zudem den Vorteil, die Entwicklungsplatine auch mit Strom aus dem USB versorgen zu können. Als Terminal nutze ich meist `minicom`.

Unter Windows kommt natürlich das AVR-Studio zum Einsatz. Die Unterschiede in der Syntax der Assemblerquelltexte zwischen den beiden Welten sind glücklicherweise gering. Von Anwendern weiß ich, dass das Hyperterminal von Windows eingesetzt werden kann, selbst nutze ich es jedoch nicht.

Hardware

Die ursprünglich geplante Laufzeitumgebung für *amforth* sind einfache Roboter wie der Asuro (www.arexx.com) und (schon anspruchsvoller) der `c't-bot` des `c't`-Computermagazins aus dem Heise-Verlag. Daneben sollte *amforth* auf einer noch zu schaffenden Modellbahnsteuerung (Vergleichbar mit www.opendcc.de) Einsatz finden. Zusätzlich habe ich noch einige Entwicklungsboards der Fa. Pollin im Einsatz, die den Prozessorwechsel unkompliziert ermöglichen. Nun hat mich die Forth-Gesellschaft überredet, auch den Atmel-Butterfly mit in die Liste aufzunehmen. . .

Wer die Hardware der Zielsysteme betrachtet, wird feststellen, dass es weder externes RAM oder EEPROM noch sonstige Anbauten gibt (der Butterfly kam erst später). Das Gesamtsystem muss mit 8KB Code und wenigen Bytes RAM zurechtkommen. Wobei natürlich nicht nur *amforth* laufen sollte, sondern auch für die Umgebung nützlicher Code. Dieses Ziel hat *amforth* erreicht.

Auf die Mikrocontroller kommt das Hexfile² mit *amforth* über die üblichen Verdächtigen: ISP oder JTAG. *amforth* und Bootloader sind einander nicht fremd, ihr Verhältnis ist dergestalt, dass *amforth* sein eigener Bootloader

ist, der keinem Standard oder den Atmel-Appnotes folgt. Die Gründe hierfür werden weiter unten noch deutlich.

Fuses

Atmel hat die AVR-Controller mit einigen Konfigurations-Bits (Fuses) versehen, die wesentliche Eigenschaften festlegen. Diese Bits sind sehr sensibel und die Standardwarnung ist, sie nur nach sorgfältiger Prüfung zu ändern. Im laufenden Betrieb sind in der Regel weder les- noch änderbar.

amforth hat auf den bislang getesteten Controllern mit deren Lieferzustand gut funktioniert. Darüber hinausgehend habe ich lediglich die Fuses für die Taktquelle angepasst, denn der Lieferzustand lässt die Controller mit nur 1MHz aus dem internen Oszillator laufen, was natürlich wenig ist, wenn extern ein 16MHz-Quarz bereitsteht.

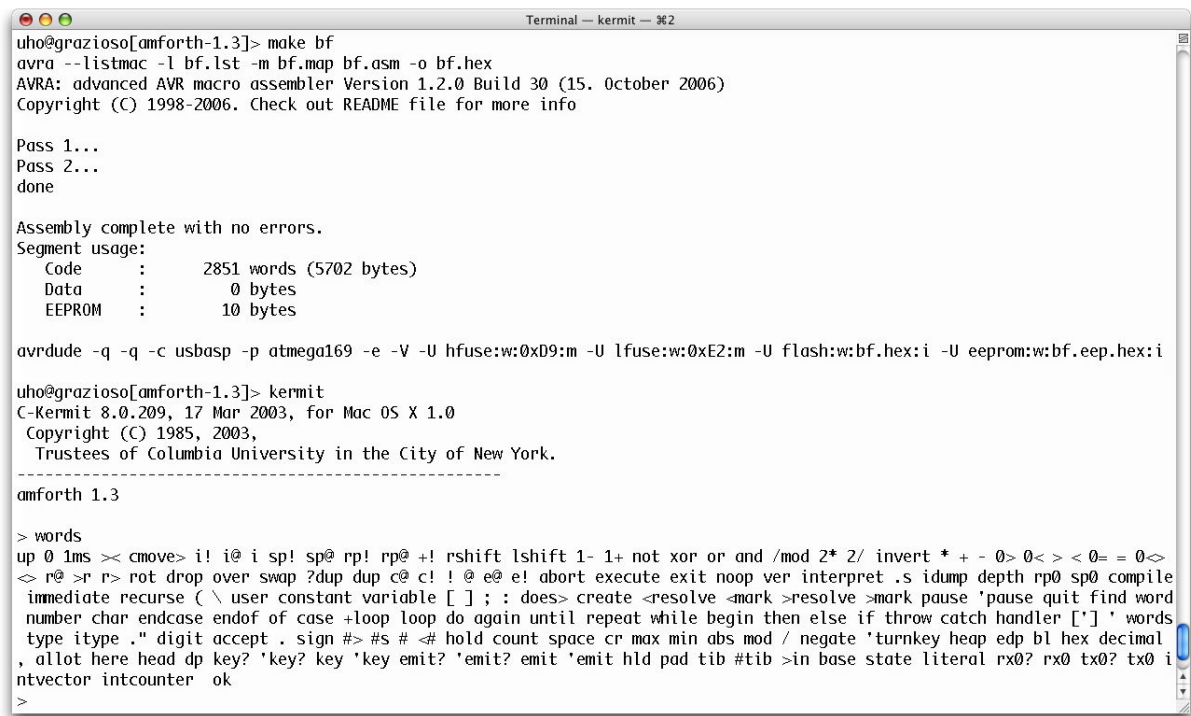
Evaluationboards

Die Boards der Fa. Pollin müssen selbst zusammengelötet werden. Dies gelingt auch einem mäßig talentierten Bastler (wie ich es einer bin) ohne größere Blessuren. Auf den Boards werden die Controller in Fassungen aufgenommen und können unkompliziert ausgetauscht werden. Die Schnittstellen umfassen ISP, JTAG, serielles Programmierinterface und einen RS232-Anschluss. Zusätzlich dabei sind (abschaltbare) LED, Summer und Taster. Als Prozessoren steht die ganze Bandbreite der im DIL-Format produzierten ATmegas zur Verfügung. Bei mir sind es der ATmega32 und der ATmega8, andere nur gelegentlich.

AREXX-Asuro

Der Asuro ist eigentlich nur eine Platine mit zwei Motoren. Der Prozessor ist ein ATmega8 mit 8MHz. Der

² eigentlich sind es zwei Dateien, eine für den Flash-Speicher und eine für das EEPROM



```
Terminal — kermit — #2
uho@grazioso[amforth-1.3]> make bf
avra --listmac -l bf.lst -m bf.map bf.asm -o bf.hex
AVRA: advanced AVR macro assembler Version 1.2.0 Build 30 (15. October 2006)
Copyright (C) 1998-2006. Check out README file for more info

Pass 1...
Pass 2...
done

Assembly complete with no errors.
Segment usage:
  Code      :    2851 words (5702 bytes)
  Data      :         0 bytes
  EEPROM    :         0 bytes

avrdude -q -q -c usbasp -p atmega169 -e -V -U hfuse:w:0xD9:m -U lfuse:w:0xE2:m -U flash:w:bf.hex:i -U eeprom:w:bf.eep.hex:i

uho@grazioso[amforth-1.3]> kermit
C-Kermit 8.0.209, 17 Mar 2003, for Mac OS X 1.0
Copyright (C) 1985, 2003,
Trustees of Columbia University in the City of New York.
-----
amforth 1.3

> words
up 0 1ms >> cmove> ! i@ i sp! sp@ rp! rp@ +! rshift lshift 1- 1+ not xor or and /mod 2* 2/ invert * + - 0> 0< > < 0= = 0<>
<> r@ >r r> rot drop over swap ?dup dup c@ c! ! @ e@ e! abort execute exit noop ver interpret .s idump depth rp0 sp0 compile
immediate recurse ( \ user constant variable [ ] ; ; does> create <resolve <mark >resolve >mark pause 'pause quit find word
number char endcase endof of case +loop loop do again until repeat while begin then else if throw catch handler [' ] ' words
type itype ." digit accept . sign #> #s # <# hold count space cr max min abs mod / negate 'turnkey heap edp bl hex decimal
, allot here head dp key? 'key? key 'key emit? 'emit? emit 'emit hld pad tib #tib >in base state literal rx0? rx0 tx0? tx0 i
ntvector intcounter ok

>
```

Abbildung 2: amforth 1.3 für AVR-Butterfly auf dem Macintosh assemblieren, flashen und booten

Kontakt zur Außenwelt kommt über Odometrie an den Antriebsachsen (Lichtreflexschranken zur Ermittlung der Drehzahl), einer Reihe von Tastern vorn und seitlich, einem Linienfolgemodul und einer Infrarotschnittstelle zum PC zustande. Letztere soll auch als Terminal für *amforth* dienen und ist bislang ein hartnäckiges Hindernis für die erfolgreiche Kommunikation.

c't-Bot

Der Roboter der c't wurde im Laufe des Jahres 2006 vorgestellt. Im Prinzip ist er dem Asuro ähnlich, aber deutlich leistungsfähiger. So hat er einen ATmega32 mit 16MHz. Zusätzlich besteht die Möglichkeit, ein Display anzuschließen oder über WLAN mit dem Roboter zu kommunizieren. Für *amforth* ist die Sache zunächst einfacher, da im Gegensatz zum Asuro die Infrarotschnittstelle nicht als serielles Terminal genutzt wird und die für selbiges notwendigen Pins separat bereitstehen.

AVR Butterfly

Michael Kalus hat mich geradezu genötigt, der kleinen Platine mehr Aufmerksamkeit zu widmen. Nachdem selbst so wichtige Argumente wie *Termine werden nicht eingehalten* und *Funktionsumfang kann auch null sein* bei ihm verpufften, hat er mir freundlicherweise zwei Exemplare dieser kleinen Systeme zur Verfügung gestellt.

Die erste *amforth*-Portierung erfolgte buchstäblich binnen weniger Minuten. Das Schwierigste war, die Taktfrequenz des ATmega169 herauszufinden. Die im Internet verfügbaren Unterlagen schwiegen sich weitgehend aus, resp. meinten, es wäre offensichtlich oder widersprachen

sich. Die Ursache für diese Verwirrung wurde aber rasch klar und der Prozessor arbeitet unter *amforth* derzeit mit 8MHz und ohne Stromsparfunktionen.

Als sich *amforth* am seriellen Terminal mit dem Prompt meldete, war das Thema Portierung beendet. Nun hat der Butterfly nicht nur den Prozessor, sondern auch recht viel Peripherie wie ein LCD, ein über SPI angeschlossenes sehr großes Flash, einen Summer, einen kleinen Joystick, einen Licht- und einen Temperatursensor. Nicht zu vergessen: Eine Li-Ion Knopfzelle.

Damit ergeben sich faszinierende Möglichkeiten, Fehler in *amforth* zu finden und auszumerzen. Und natürlich auch Einsatzgebiete.

Und die Modellbahn?

Da gibt es schlicht noch nichts zu vermelden. Die Hardware hat Wolfgang Kufer unter www.opendcc.de beschrieben, sie tut jedoch erst mal ihren Zweck wie es vom Erschaffer vorgesehen ist. Wer aber mit Schlagworten zufrieden ist, hier kommen ein paar: SRCP und embeddedloconet.

Andere Systeme

Es bleibt nicht aus, dass *amforth* auch auf anderen Systemen eingesetzt wird. So liegt eine kleine Platine mit einem ATmega88 und Ethernet-Anschluss vor mir. Dann gibt es einige Prozessoren der ATtiny-Reihe, die über hinreichend viel Flash verfügen.

Umsetzung

amforth ist ein 16Bit-Forth in der indirect-threaded-Ausführung für die AVR-ATmega-Mikrocontroller. Leider hat Atmel die ATmegas nicht vollständig softwarekompatibel gestaltet, so dass es unumgänglich ist, für jeden Prozessortyp einige Einstellungen vorzunehmen und ein eigenes Hexfile für *amforth* zu erstellen. Diese Einstellungen umfassen alle nicht zur Laufzeit änderbaren Parameter, wie z. B. die Taktfrequenz oder Adressbereiche und einige Startwerte, um überhaupt Kontakt aufnehmen zu können.

amforth ist als Stand-alone-Forth-System konzipiert. Es wird einmalig auf dem PC aus den Quellen übersetzt und auf den Mikrocontroller übertragen. Anschließend arbeitet es autonom. Kommandos werden über das serielle Terminal entgegengenommen bzw. über einen turnkey getauften Mechanismus bei Programmstart automatisch ausgeführt.

amforth hat einen Compiler. Damit definierte Worte erweitern das Dictionary im Flash direkt und ohne weitere Aktionen im laufenden Betrieb. Der Compiler arbeitet klassisch. Er kennt keine Optimierung oder Codeanalysen, wie sie von modernen Compilern (auch für Forth) angeboten und eingesetzt werden.

amforth lehnt sich an den vielfach im Internet vorhandenen Text Dpans94-draft6 an. Dabei folgt es dem Standard nicht sklavisch. Die Stackeffekte wie auch die grundsätzliche Funktion stimmen überein, aber in Details ergeben sich Abweichungen. So arbeitet `find` case-sensitiv und die Worte im Dictionary sind alle in Kleinbuchstaben notiert. Mir erschien es aber nicht hinreichend wichtig, vollständige Konformität herzustellen, zumal einige Aspekte auf dem Mikrocontroller meiner Meinung nach keinen Sinn ergeben.

Der Zugriff auf die verschiedenen Speichertypen erfolgt über die Standardworte ggf. ergänzt mit Präfixen. Der Einsatz dieser Präfixe ist reine Konvention. Das Präfix `e` steht für EEPROM, das Präfix `i` für Flash (== Instruction) Memory. Die Worte `@` und `!` (also ohne Präfix) operieren mit den RAM-Adressen. Nur für das RAM existieren auch byte-weise Zugriffsoperationen: `c@` und `c!`. Diese sind insbesondere bei den I/O-Registern unverzichtbar. Dafür fehlt `c,`, da auf das Flash ohnehin nur wortweise zugegriffen werden kann. Einen byte-weisen Zugriff auf das EEPROM hielt ich bislang für unnötig, da bis auf den Sonderfall mit den I/O-Adressen ohnehin immer 16Bit große Worte verarbeitet werden.

Aus dem CORE-word-set fehlen Worte wie `environment`, `evaluate` (und auch `s"`), da ich für diese (noch) keinen Nutzen gefunden habe. Eine Strukturierung des Dictionarys in einzelne Vocabularys wird ebensowenig realisiert.

Einige Eigenschaften sind dem angestrebten Minimalsystem (ATmega8 ohne Zusatzspeicher) geschuldet. So ist

die Fehlertoleranz eher gering ausgeprägt. Wer z.B. Worte mit mehr als 31 Zeichen definiert, wird daran nicht gehindert. Es wird nur nicht funktionieren. Außerdem ist *amforth* sehr schweigsam, auch im Fehlerfall. Es gibt keine kleinen „Romane“, die erläutern, was wie warum falsch lief und was man alles tun könnte. Einzig die Position in der aktuellen Eingabezeile und ein Errorcode werden ausgegeben.

Keine der Eigenschaften ist in Stein gemeißelt. Beispielsweise kann man sich vorstellen, das Dictionary teilweise in andere Speicherbereiche auszulagern und dabei verschiedene Vocabularys zu benutzen³. Auch kann eine Blockverwaltung sinnvoll werden, wenn beispielsweise SD-Cards einbezogen werden.

Architektur

Die Architektur von *amforth* lehnt sich stark an die an, die von Ron Minke in seiner Artikelfolge „Forth von der Pike auf“ beschrieben wird. Zum Glück hat Ron erst im letzten Artikel seine Hardware enthüllt, *amforth* wäre anderenfalls vielleicht nicht entstanden. Die von ihm genutzte Hardware ist deutlich komplexer und seine Forth-Implementierung viel aufwändiger als *amforth*.

Die Verwendung der CPU-Register der ATmega-Controller ist wie bei ihm beschrieben. Hinzu kommen die Register `r16` bis `r27` für temporäre Zwischenspeicher. Bei der Multiplikation kommt zudem das Register `r0` zum Einsatz. Weiter werden die Register `r2` bis `r5` für interne Zwecke genutzt. Die restlichen Register sind noch ungenutzt, im Ausblick am Ende dieses Artikels werden auch sie eventuell noch Nutzen finden.

Interrupts

amforth kann Interrupts durch Forth-Worte ausführen. Hierfür kommt ein zweistufiges Verfahren zum Einsatz, das den Interrupt auf Maschinenebene in einen Interrupt auf Forth-Ebene umsetzt. Hierfür wird das sonst ungenutzte `t`-Bit im Statusregister genutzt. Wenn es gesetzt ist, verzweigt der innere Interpreter in die Interruptverarbeitung. Damit ist die Latenz bis zur Verarbeitung des Interrupts recht hoch. Im Gegenzug hat man alle Freiheiten eines Forth-Interpreters. Interrupts wirken nur beim Einsprung in den inneren Interpreter. Das bedeutet, dass alle Assemblerworte als atomare Operationen wirken und sich der Forth-Interpreter in einem wohlbekanntem Zustand befindet.

Forth-Worte, die als Interrupt aufgerufen werden, sollten natürlich keine Seiteneffekte haben. Das eigentliche Programm wird es danken...

Gegenwärtig stehen nicht alle ATmega-Interrupts auch als Forth-Interrupts zur Verfügung. Die Anzahl und auch die Auswahl werden über die prozessor- und plattformspezifischen Steuerdateien festgelegt.

³ Eine vielleicht etwas eigenwillige Interpretation des Forth-Standards.

Multitasking

Multitasking wird über das Wort `pause` organisiert. *amforth* selbst definiert `pause` vektorisiert (deferred) als `noop` (No Operation), es macht also nichts. `pause` wird bei den I/O–Routinen bereits intern aufgerufen. Wer das Abenteuer sucht, definiert `pause` über einen Timerinterrupt...

Der Kontakt zur Außenwelt wird über 4 Worte hergestellt: `emit`, `emit?`, `key` und `key?`. Diese sind über Uservariablen (die Namen derselben beginnen mit einem `'`) vektorisiert. Default ist die Nutzung des ersten seriellen Ports (`usart0`). Denkbar ist aber ebenso eine Nutzung von TWI (I2C) oder SPI oder (bei passenden ATmegas) des CAN–Busses zur Entgegennahme von Befehlen. Ebenso lässt sich `emit` auf ein LCD (z. B. des Butterflies oder des `c't`–Roboters) oder TV umsetzen — Die ATmegas sind in der Lage, ein FBAS–Fernsehsignal zu generieren, Schaltungen im Internet.

Flash

Der Flash–Speicher wird in *amforth* in mehrere Teile untergliedert. Eine wichtige Gliederung wird von Atmel vorgegeben: NRW– und RWW–Speicher. Der NRW(non read while write)–Speicher liegt im Bootsektorbereich am oberen Ende des Speichers. Programmcode, der sich hier befindet, kann auf das Flash im RWW–Bereich schreibend zugreifen, sofern nicht Fuses oder Lockbits dies verhindern. Die hierfür genutzte Instruktion `spm` ist im RWW–Bereich wirkungslos. Dieser von Atmel in verschiedenen Appnotes beschriebene Vorgang wird z. B. von Bootloadern genutzt, die ein Firmwareupdate ermöglichen. *amforth* hat in diesem Bereich seinen gesamten in Assembler codierten Sprachkern und einige Colonworte, um das Wort `i!` ausführen zu können. Jetzt wird sicher auch verständlich, warum *amforth* keinen Bootloader mag (er wird überschrieben): Kein Platz mehr frei.

Aus einer anderen Perspektive ist *amforth* selbst ein Bootloader: Über das serielle Terminal lassen sich Befehle eingeben, die das Dictionary verändern und so die Funktion des Controllers verändern. Die Kommandosprache ist nur nicht kompatibel zu den in den Appnotes beschriebenen Befehlen. Aber vielleicht findet sich ja jemand, der die Syntax der Bootloader–Kommandos in *amforth* einbaut.

Damit sind wir im unteren, mit RWW bezeichneten Bereich. Hier sind die Interruptvektoren (ab Adresse 0), die für die Interrupts zuständigen Maschinencodbefehle, weitere Routinen, die für die konkrete Laufzeitumgebung wichtig sind, und nicht als Forth–Worte codiert werden, und der zweite, größere Teil des Dictionarys. Neue Worte werden an diesen, unteren Teil des Flash angehängt. Hier ist auch der Grund für das große Hexfile: Die „Lücke“ im Flash kann recht groß sein (*amforth* ist derzeit (Version 1.3) ca. 5,5 KB groß, trotzdem wird immer der gesamte Flash neu programmiert).

Flash–Speicher hat beim Schreibzugriff einige Besonderheiten. Die erste ist, dass es keinen Zugriff auf einzelne Zellen (Eine Zelle ist so groß wie ein Maschinenwort für den Prozessor: 16Bit), sondern nur auf eine Page gibt, die „am Stück“ geschrieben wird. Wenn man eine einzelne Zelle ändern möchte, muss man also zunächst den aktuellen Inhalt der betreffenden Page in einen eigens vorhandenen Puffer einlesen, die gewünschte Zelle richtig eintragen (Der Puffer kann nur einmal beschrieben werden), das Flash löschen, und kann dann die Page neu schreiben. Um die Anzahl der Löschvorgänge zu minimieren, prüft *amforth*, ob überhaupt ein Löschvorgang erforderlich ist. Dies ist der Fall, wenn ein Bit von 0 nach 1 geändert wird. Von 1 nach 0 geht auch ohne Löschen und damit flash–schonend. Solange immer nur neue Worte eingetragen werden ist *amforth* flash–schonend, die Ausnahme bei Worten, die `does>` benutzen wurde erwähnt: `does>` ändert das von `create` angelegte und bereits geschriebene Execution–Token von `DO_VARIABLE` nach `DO_COLON`.

Bevor jetzt Befürchtungen aufkommen: *amforth* wird immer noch auf dem ersten Mikrocontroller entwickelt.

RAM

Die Hardware der ATmegas bietet ungewöhnliche Zugriffsmethoden. So sind die CPU–Register und die I/O–Ports über eine RAM–Speicheradresse les– und beschreibbar. Register `r0` hat die Speicheradresse 0, I/O–Port 0 hat die Speicheradresse 32. Das „wirkliche“ RAM (RAM–Zellen sind im Unterschied zu Flash–Zellen 8Bit groß) hat eine prozessorabhängige Startadresse. Der ATmega32 z. B. fängt bei Adresse 96 (hex 60) mit RAM an; der ATmega169 erst bei 256 (hex 100).

Der Vorteil dieser Adressierung ist, dass man mit einem Wort (`c@` resp. `c!`) sowohl auf den Hauptspeicher als auch auf I/O–Ports zugreifen kann. CPU–Register sollte man nur nach reiflicher Überlegung nutzen.

amforth benötigt RAM, um die verschiedenen Forth–Datenbereiche abzubilden (`h1d`, `pad` etc.). Hier zeigt sich eine weitere Besonderheit des *amforth*: `PAD` wandert nicht mit dem Dictionary mit, sondern ist ortsfest. Hinzu kommen einige Bytes, um dem seriellen Terminal Platz für Ein– und Ausgabepuffer bereitzustellen und die Interruptverwaltung zu erledigen. Ebenso wird eine Userarea initialisiert. Dafür werden ca. 200 Bytes benötigt. Der Rest wird über Forth–Worte wie `heap` und `variable` (das `heap` intern nutzt) verwaltet.

EEPROM

Das in die Prozessoren eingebaute EEPROM wird auf den ersten Blick fast gar nicht genutzt. Das zugehörige Hexfile umfasst nur wenige Bytes, so dass man geneigt sein könnte, es wegzulassen. Nur: Ohne diese Bytes arbeitet *amforth* nicht.

Im EEPROM werden Variablen geführt, die die Verwaltung der zentralen Datenstrukturen realisieren. Sie können nicht im RAM liegen, da sie einen Stromausfall überstehen müssen, können aber andererseits nicht in das Flash geschrieben werden, da sie oft geändert werden und dies das Flash sehr schnell ruinieren würde.

Im EEPROM sind die beiden Einstiegspunkte in das Dictionary (`dp` und `head`) gespeichert und auch die nächste freie RAM-Adresse für z.B. `variable`. Ebenso ist der erwähnte `turnkey`-Mechanismus im Grunde nur eine EEPROM-Variable, die das Execution-Token eines Forthwortes enthält, das durch das Wort `quit` ausgeführt wird, bevor der `accept/interpret`-Zyklus startet.

Assemblerumsetzung

Jedes Wort besitzt seine eigene Quelldatei mit der Endung `.asm`, auch wenn es als Folge von Execution-Tokens definiert wird. Bei diesen Worten ist zudem Vorsicht geboten: Es kann sein, dass sie nicht in korrekten Forth-Code zurückübersetzt werden können.

Es werden außerdem Worte verwendet, die keinen Forth-Header haben. Dies betrifft vor allem Worte, die von immediate-Worten in das Dictionary compiliert werden, um zur Laufzeit aktiv zu werden. In anderen Forth-Systemen werden z. B. runde Klammern genutzt, um diese Worte zu markieren. *amforth* verbirgt sie hier für eine Optimierung des Platzbedarfs. Es ist nicht vorgesehen, dass diese „hidden words“ von anderen als den zugehörigen `immediate`-words genutzt werden. Also spart sich *amforth* den Speicherplatz für den Dictionary-Eintrag und compiliert nur das Execution-Token (XT). Damit werden Worte wie `see` natürlich problematisch. Aber die sind ohnehin unnötig, da der Quelltext zur Verfügung steht.

In der Regel wird der relative Sprungbefehl des ATmega-Prozessors verwendet. Dieser ist schneller und benötigt weniger Platz. Der Nachteil ist, dass nicht der gesamte Flash-Speicher erreicht werden kann. Dies ist ein weiterer Grund, warum *alle* in Assembler codierten Worte im NRWW-Bereich des Flash platziert sind.

Updates

Ein Update des *amforth* im Mikrocontroller ist ein aufwändigeres Verfahren. Zuerst muss das Basissystem mit ISP oder JTAG neu geladen werden. Anschließend muss die eigentliche Anwendung wieder aus den Quellen oder per Hand im seriellen Terminal eingespielt werden.

Es ist sehr empfehlenswert, die interaktiv definierten Worte anderweitig abzulegen, um sie erneut einspielen zu können. Von *amforth* ist hier keine Unterstützung zu

erwarten, da es als mikrocontroller-basiertes System keine Ahnung von Filesystemen hat und somit einen Befehl wie `include <filename>` nicht bereitstellen *kann*.

Libraries

Es wurde bereits angedeutet, wie Libraries bei *amforth* funktionieren: Gar nicht. Es ist Aufgabe des Nutzers, die Definitionsfiles in der richtigen Reihenfolge auf den Controller zu übertragen.

Ausblick

amforth hat trotz seines geringen Alters eine gewisse Aufmerksamkeit geweckt. Die Mails der Nutzer umfassen sowohl Erfolgs- wie auch Fehlermeldungen bzw. Unklarheiten des Systems betreffend. Die Einarbeitung dieser Informationen hat *amforth* auch schon ein gutes Stück vorangebracht.

Für die Zukunft ist eine Sammlung von Routinen wünschenswert, die Standardaufgaben abdeckt. Auch wenn es kein Library-Konzept seitens *amforth* gibt, können Routinen für beispielsweise den Zugriff auf den SPI- und TWI/I2C-Bus bereitgestellt werden. Darauf können dann Worte aufsetzen, die spezielle I2C-Bausteine ansprechen. Idealerweise kann diese Bibliothek mit anderen Hardwareplattformen kombiniert werden, z. B. mit den Artikeln von Erich Wälde „Adventures in Forth“.

Weitere zukünftige Änderungen betreffen die Organisation des obersten Stackelements TOS. Anton Ertl hat herausgefunden, dass dies Geschwindigkeitsvorteile bringen kann. Auf jeden Fall wird durch entfallene `push/pop`-Operationen der Code kleiner, was auch einen Wert an sich darstellt.

Literatur

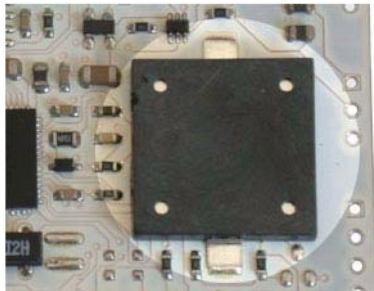
- [1] *amforth* Homepage, <http://amforth.sourceforge.net/>
- [2] Ron Minke „Forth von der Pike auf“ in: Vierte Dimension 3/4-2005 bis 1-2007
- [3] Stephen Pelc „Programming Forth“ MPE Limited, 2005, <http://www.mpeltd.demon.co.uk/arena/ProgramForth.pdf>
- [4] Julian V. Noble „Finite State Machines in Forth“ in: THE JOURNAL OF FORTH APPLICATION AND RESEARCH, <http://dec.bournemouth.ac.uk/forth/jfar/vol17/paper1/paper.html>
- [5] Erich Wälde „Adventures in Forth“ in: Vierte Dimension 3-2006



Der Piezo–Summer des AVR–Butterfly

Ulrich Hoffmann, Michael Kalus

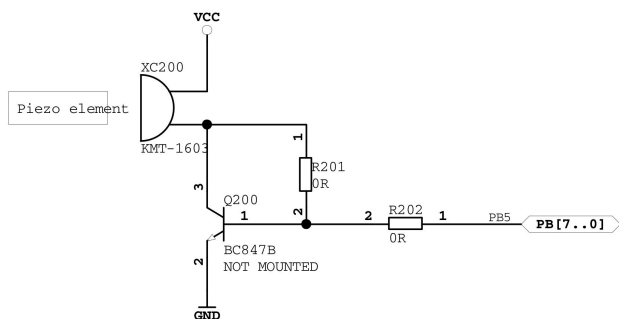
Für die Ausgabe von Tönen hat der AVR–Butterfly einen Piezosummer. Dieser ist an PORTB5 angeschlossen. Benutzt man PWM (Pulsweiten–Modulation), lassen sich verschiedene Frequenzen generieren, um Melodien zu spielen.



Das Piezo–Element auf dem Butterfly–Board

Dieser Port–Pin 5 dient neben der normalen I/O–Funktion auch als Output für den PWM–Timer. Er ist daher zur Ausgabe von Frequenzen besonders gut geeignet. Deswegen haben ihn die Atmel–Ingenieure wohl auch als Anschluss für den Piezo–Summer auf dem AVR–Butterfly–Board verwendet.

Aber auch durch einfaches Umschalten des Pins lassen sich dem Summer Töne entlocken: Siehe Forth–Beispiel–Code im Listing auf Seite 35



Ansteuerungsschaltung des Piezo–Elements

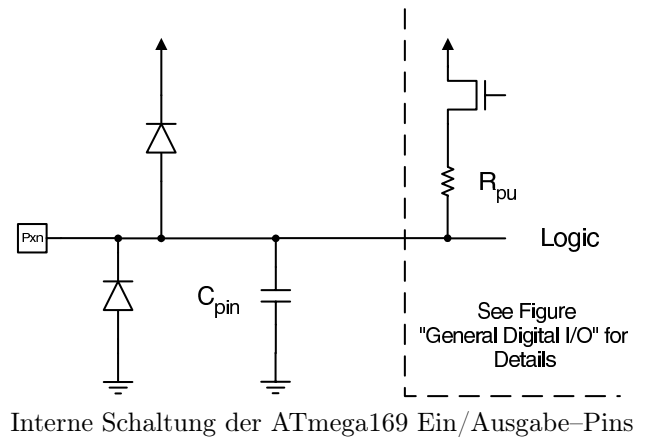
Funktion der Port–Pins

Die Output–Pins auf dem ATmega169 können alle einzeln gesteuert werden. Direkt mittels der SBI– und CBI–Befehle oder durch Beschreiben/Lesen des zugehörigen Data–Registers. Jeder Pin verfügt über hi–sink & hi–source–Eigenschaften und ist daher in der Lage, eine LED zu treiben. Auch hat jeder Pin einen eigenen pull–up–Widerstand mit Vcc–unabhängigem Wert sowie Schutzdioden gegen Vcc und Gnd.

Links

- AVR–Butterfly Manual: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3146
- Piezo: http://www.forth-ev.de/wiki/doku.php/projects:avr:pizto_speaker
- Datenblatt ATmega169: <http://www.atmel.com/> Suche: ATmega169

Schaltungen und Foto entstammen den Atmel–Datenblättern. Sie sind hier nur zur Illustration wiedergegeben.



Interne Schaltung der ATmega169 Ein/Ausgabe–Pins

Nomenklatur:

Registername, Portbuchstabe *x*, Pin Nummer *n*

DRB5	Port B data register pin 5;	w/r
DDRB5	Port B data direction register pin 5;	w/r
pinB5	Port B pin register B pin 5;	read only

Jeder Port — und damit jeder Pin einzeln auch — besteht aus 3 Registern. Dem Data–Register DRx, dem Data–Direction–Register DDRx und dem Port–Input–Register PINx. Die Pins können einzeln Eingang oder Ausgang sein. Das PINx spielt eine besondere Rolle, es kann nur eingelesen werden. Das geschieht aber über die gleichen Pins wie beim DRx.

Eine den ganzen Portx betreffende Kontrolle besteht über das MCUCR. Wird dort das pull–up–disable–bit (PUDbit) gesetzt, sind alle Pins des Ports ohne pull–up–Widerstand.

Jeder Pin eines Ports ist mehrfach belegt. Unser PORTB5 hat neben dem I/O noch die Funktionen OC1A/PCINT13, Bit 5.

OC1A, Output–Compare–Match–A–output: Der PB5–Pin dient zur Ausgabe für den Timer/Counter1 Output–Compare–A. Dazu muss der Pin als Output (DDB5 set (one)) gesetzt sein. Dann ist der OC1A der Ausgabepin für die PWM–mode–timer–function. Und schließlich dient er noch als PCINT13, Pin–Change–Interrupt–Source 13: Der PB5–Pin ist dann Eingang für eine externe Interrupt Quelle.

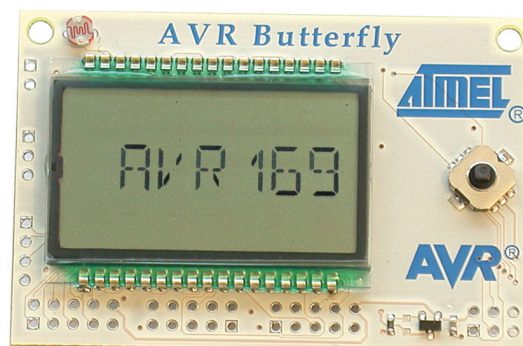
Über einen Multiplexer können diese alternativen Funktionen zugewiesen werden. Siehe Datenblatt des ATmega169.



```

1  \ Butterfly Piezo
2
3  25 constant portB    \ used with c@ c!  20 higher than I/O-port
4  24 constant ddrB    \ see page 22 I/O Memory in ATmega169 manual
5  23 constant pinB
6  20 constant piezoMask \ uses pins PB5
7
8  : piezo-init ( -- )
9      ddrB c@ piezoMask or ddrB c! ;
10
11 piezo-init
12
13 : click ( -- ) portB c@ piezoMask or portB c! ;
14 : clack ( -- ) portB c@ piezoMask not and portB c! ;
15
16 variable fudge 5 fudge !
17
18 : delay ( n -- )
19     begin ?dup while fudge @
20         begin ?dup while 1- repeat
21             1-
22         repeat ;
23
24 : tone ( dur 1/pitch -- )
25     dup >r / r> \ quotient * pitch = duration
26     begin
27         over
28         while
29             click dup delay
30             clack dup delay
31             swap 1- swap
32         repeat
33     drop drop ;
34
35 \ name your tones
36
37 : beep ( -- ) 100 3 tone ;

```



amforth 1.3 Referenz

Stack Manipulation

```
drop      ( n -- )
dup       ( n -- n n )
swap     ( n1 n2 -- n2 n1 )
over     ( n1 n2 -- n1 n2 n1 )
rot      ( n1 n2 n3 -- n2 n3 n1 )
?dup     ( n1 -- [ n1 n1 ] | 0 )
depth    ( -- n )
>r       ( n -- )
r>       ( -- n )
r@       ( -- n )
```

Vergleiche

```
>        ( n1 n2 -- flag )
<        ( n1 n2 -- flag )
=        ( n1 n2 -- flag )
<>       ( n1 n2 -- flag )
0>       ( n1 -- flag )
0<       ( n1 -- flag )
0=       ( n -- flag )
0<>      ( n -- flag )
```

Arithmetisch und logisch

```
*        ( n1 n2 -- n3 )
+        ( n1 n2 -- n3 )
-        ( n1 n2 -- n3 )
/        ( u1 u2 -- u3 )
/mod     ( u1 u2 -- u3 u4 )
0        ( -- 0 )
1+       ( n1 -- n2 )
1-       ( n1 -- n2 )
2*       ( n1 -- n2 )
2/       ( n1 -- n2 )
abs      ( n1 -- u1 )
invert   ( n1 -- n2 )
lshift   ( n1 n2 -- n3 )
max      ( n1 n2 -- u1 )
min      ( n1 n2 -- u1 )
mod      ( n1 n2 -- n3 )
negate   ( n1 -- n2 )
or       ( n1 n2 -- n3 )
rshift   ( n1 n2 -- n3 )
xor      ( n1 n2 -- n3 )
and      ( n1 n2 -- n3 )
not      ( flag1 -- flag2 )
><       ( n1 -- n2 )
```

Speicher

```
!        ( n addr -- )
@        ( addr -- n )
+!       ( n addr -- )
c!       ( c addr -- )
c@       ( addr - c1 )
cmove>   ( addr-from addr-to n -- )
count    ( addr -- addr+1 n )
e!       ( n addr -- )
e@       ( addr -- n )
i!       ( n addr -- )
i@       ( addr -- n1 )
```

Kontrollstrukturen

```
case     ( -- 0 )
do       ( -- addr )
else     ( addr1 -- addr2 )
endcase  ( flag -- )
endof    ( addr1 -- addr2 )
if       ( x -- )
loop     ( addr -- )
of       ( -- )
repeat   ( -- )
then     ( -- )
until    ( flag -- )
while    ( flag -- )
again    ( -- )
begin    ( -- )
execute  ( xt -- )
exit     ( -- )
i        ( -- n )
recurse  ( i*x - j*x )
```

Zeichen Ein/Ausgabe

```
cr       ( -- )
emit     ( c -- )
emit?    ( -- flag )
key      ( -- c )
key?     ( -- flag )
itype    ( addr1 -- addr2 )
space    ( -- )
type     ( addr n -- )
#tib     ( -- addr )
tib      ( -- addr )
accept   ( addr n1 -- n2 )
char     ( -- c )
<c>      ( -- )
."       ( -- )
text"    ( -- 32 )
bl       ( -- 32 )
```

Zahlen Ein/Ausgabe

```
#        ( n1 -- n2 )
#>      ( n1 -- addr n2 )
#s      ( n1 -- 0 )
<#      ( -- )
decimal ( -- )
hex     ( -- )
hld     ( -- addr )
hold    ( c -- )
sign    ( n -- )
base    ( -- addr )
digit   ( c base -- n flag )
number  ( addr -- n )
.        ( n -- )
```

Definierende Worte

```
constant <name> ( n -- )
          <name> ( -- n )
user <name>      ( n -- )
          <name> ( -- addr )
variable <name> ( -- )
          <name> ( -- addr )
create <name>   ( -- )
          <name> ( -- addr )
does>          ( -- )
: <name>       ( -- )
;              ( -- )
```

```
>mark    ( -- addr )
>resolve ( addr -- )
throw    ( n -- )
catch    ( xt -- )
abort    ( nx -- )
```

Wörterbuch

```
' <name> ( -- xt )
find      ( addr -- addr 0 | xt flag )
```

Compiler

```
compile   ( -- )
<name>    ( -- )
word      ( c -- addr )
['        ( -- addr )
<name>    ( -- )
allot     ( n -- )
immediate ( -- )
literal   ( n -- )
<n>       ( -- )
[         ( -- )
]         ( -- )
state     ( -- addr )
,         ( x -- )
```

Systemzeiger

```
dp        ( -- eaddr )
edp       ( -- eaddr )
head      ( -- eaddr )
heap      ( -- eaddr )
here      ( -- addr )
up        ( -- addr )
'turnkey  ( -- eaddr )
handler   ( -- addr )
'emit     ( -- eaddr )
'emit?    ( -- eaddr )
'key      ( -- eaddr )
'key?     ( -- eaddr )
```

Stacks

```
rp!       ( n -- )
rp0       ( -- addr )
rp@       ( -- n )
sp!       ( addr -- )
sp0       ( -- addr )
sp@       ( -- n )
```

Sonstiges

```
noop      ( -- )
pause     ( -- )
quit      ( -- )
interpret ( -- )
words     ( -- )
idump     ( addr len -- )
pad       ( -- addr )
(         ( -- )
\         ( -- )
.s        ( -- )
lms       ( -- )
ver       ( -- )
>in       ( -- addr )
```