

Kommentare

Die allgemeine Form eines Kommentars sieht so aus:

```
\ Dies ist ein Text
```

\ schaltet den Parser aus. Er wird bei Zeilenende wieder eingeschaltet. Stackkommentare sehen so aus:

```
( N2 N1 - N3 )
```

(schaltet den Parser aus,) schaltet ihn wieder ein. Was dazwischen liegt wird ignoriert. Wenn die Stackkommentare hinter \ stehen, können damit mehrere Zeilen im Sourcecode deaktiviert werden:

```
(  
: TEST      \ ( --- )  
;  
)
```

Bei \ , (und) handelt es sich um FORTH-Befehle. Sie brauchen deshalb links und rechts SPACE oder CR. Vorsicht: leere \ ohne folgenden Kommentartext können die folgende Zeile löschen.

Der Stack

Wesentliches Kennzeichen von FORTH ist die Parameterübergabe auf einem Stack. In anderen Programmiersprachen werden dafür Variablen verwendet. FORTH hat durch den Stack einen kompakten, aber etwas undurchsichtigen Sourcecode. Ferner werden Laufzeit und der Verbrauch von RAM durch ihn minimiert.

Der Stack ist 16 Bit breit. Auch auf einer 8 Bit CPU programmiert man also auf einem virtuellen 16 Bit Stackprozessor. Um die Lesbarkeit zu verbessern sind Stackkommentare üblich.

```
: TEST      \ ( N2 N1 --- N4 N3 )  
            \ ( N2 N1 ---      )  
DROP       \ (      --- N2      )  
1+ 5 ;
```

Vor Ausführung des Befehls TEST waren zwei Werte auf dem Stack. Oben N1, darunter N2. Der Befehl verarbeitet beide Werte und gibt als Ergebnis die neuen Werte N4 und N3 zurück auf den Stack. Oben N3, darunter N4.

Wenn man innerhalb eines Befehls Kommentare anbringt, sollten sie, wenn sie links --- den Zustand zu Beginn der Zeile, rechts den Zustand am Ende der Zeile darstellen.

Zahlenformate

Es gibt neben den üblichen 16 Bit Worten auch 32 Bit Werte auf dem Stack. Diese bestehen aus zwei 16 Bit Teilen, das obere Wort oben auf dem Stack, das untere Wort darunter.

Nx einfach genaue 2er-Komplement Festkommazahl
 „Number“
 7FFF = +32767
 0000 = 0
 FFFF = -1
 8000 = -32768

UNx einfach genaue Zahl ohne Vorzeichen
 „Unsigned Number“
 FFFF = 65535
 0000 = 0

Dx doppelt genaue 2er-Komplement Festkommazahl
 „Double Length Number“
 zwei Stackwerte, 32-Bit breite Zahl
 7FFFFFFFF = +2147483647
 00000000 = 0
 FFFFFFFFF = -1
 80000000 = -2147483648

UDx doppelt genaue Zahl ohne Vorzeichen
 „Unsigned Double“
 FFFFFFFFF = 4294967295
 00000000 = 0

Cx Byte, oberes Byte ist immer 0
 „Character“
 00FF
 0000

ASCIIx wie „Character“, aber es handelt sich tatsächlich um einen 7 Bit ASCII Wert.
 „ASCII“
 007F
 0000

Addr absolute Adresse im Adreßraum der CPU
 „Address“
 FFFF = letztes Byte (Vektoren)
 0000 = erstes Byte (I/O)

Fx Flag, „Flag“
 alle 16 Bits = 0 = false
 irgendein Bit gesetzt = 1 = true

Zahlenbasis

Die Einstellung bezieht sich nur auf I/O zum Terminal, gespeichert werden alle Zahlen binär. Man kann sie global einstellen mit:

B% 1111000000101101 \ Binär
 H% ABC \ Hex
 D% 139 \ Dezimal

DECIMAL \ (-)
 HEX \ (-)

B% H% und D% sind FORTH-Befehle. Sie funktionieren auch innerhalb von Befehlen („state-smart“). Beispiel:

Hex ist nach Reset eingestellt. Der Stackausdruck erfolgt immer Hex.

```
---- | : X B% 10101010 BBBB ;
---- | X \ Test
---- 00AA BBBB |
```

Lokale Einstellungen gelten nur für eine folgende Zahl. Z.B.:

Stack-Befehle

Meist liegen die Daten nicht in der Reihenfolge auf dem Stack wie man sie braucht. Einige häufig benutzte Befehle sorgen für die Umordnung der Unordnung. Offensichtlich kann man nur die 3 obersten Stackwerte effizient verarbeiten. Reicht das nicht, sollte man Variablen anlegen.

Tabelle 1: Stackbefehle

DROP \ (N1 -)	Entfernen
DUP \ (N1 - N1 N1)	Verdoppeln
SWAP \ (N2 N1 - N1 N2)	Vertauschen
OVER \ (N2 N1 - N2 N1 N2)	2. Wert kopieren
ROT \ (N3 N2 N1 - N2 N1 N3)	3. Wert vorholen
HOPP \ (N3 N2 N1 - N3 N2 N1 N3)	3. Wert kopieren

Für 32 und 16 Bit Stackwerte:

2DROP \ (D1 -)	(N2 N1 -)
2DUP \ (D1 - D1 D1)	(N2 N1 - N2 N1 N2 N1)
2SWAP \ (D2 D1 - D1 D2)	(N4 N3 N2 N1 - N2 N1 N4 N3)
2OVER \ (D2 D1 - D2 D1 D2)	

Arithmetik

Im System sind nur die grundlegenden Befehle integriert.

Die Additions- und Subtraktionsbefehle sind für vorzeichenlose Zahlen und Zahlen im 2er-Komplement verwendbar. Multiplikation und Division nur für vorzeichenlose Zahlen.

Die Eingabe von z.B. D% -1234 ist nicht möglich, da der Parser keine negativen Zahlen verarbeitet. Man kann sich aber mit D% 1234 NEGATE behelfen. Man beachte aber, daß NEGATE für die negativste Zahl -32768d nicht definiert ist.

Die Befehle mit integriertem Operanden sind schneller und verbrauchen weniger Speicher. Kaskadieren von bis zu zwei derartiger Befehlen ist sinnvoll:

1+ 2+ statt 3 +

Multiplikation und Division sind langsam und sollten deshalb wo möglich durch Schiebeoperationen ersetzt werden. Um Überlauf zu vermeiden, muß 32 Bit Wortbreite verwendet werden.

Tabelle 2: Arithmetik

+	\ (N1 N2 - N3)	N1 + N2 = N3
-	\ (N1 N2 - N3)	N1 - N2 = N3
1+	\ (N1 - N2)	N1 + 1 = N2
1-	\ (N1 - N2)	N1 - 1 = N2
-1	\ (- N1)	Konstante -1 = FFFF
2+	\ (N1 - N2)	N1 + 2 = N2
2-	\ (N1 - N2)	N1 - 2 = N2
U*	\ (UN1 UN2 - UD1)	UN1 * UN2 = UD3
U/	\ (UD1 UN1 - UN2)	UD1 / UN1 = UN2
U/MOD	\ (UD1 UN1 - UN3 UN2)	UD1 / UN1 = UN2 UN3 = Rest
NEGATE	\ (N1 --- N2)	Vorzeichen wechseln für 2er-Komplementzahl

Der Rest bei U/MOD ist die Zahl, die nichtmehr weiter teilbar ist. Auf dem Taschenrechner: 5 / 2 = 2,5. Bei Integerarithmetik: 5 / 2 = 2, Rest = 1.

Es wird weder während des Compilierens, noch während der Ausführung geprüft ob durch einen Befehl der Wertebereich überschritten wird.

DECIMAL 20000 50000 +

ergibt nicht 70000 sondern einen Überlauf, da sich die Zahl nicht mit 16 Bit darstellen läßt.

Boolsche & Schiebebefehle

Die Zahl im Befehl gibt die Zahl der Schiebeoperationen. Der Pfeil die Richtung. Ein Carrybit steht allerdings nicht zur Verfügung. Freiwerdende Bits werden mit 0 gefüllt.

Mit Shifts können Multiplikation und Division durchgeführt werden. Division funktioniert jedoch nicht mit 2er-Komplementzahlen und hat unsauberes Rundungsverhalten.

Kaskadierung ergibt die gewünschte Funktion:

```
2* = 1<SHIFT
4* = 1<SHIFT 1<SHIFT
8* = 1<SHIFT 1<SHIFT
    1<SHIFT
```

Tabelle 3: Boolsche Bit-Befehle

```
AND \ ( UN2 UN1 - UN3 ) Bitweises AND
OR  \ ( UN2 UN1 - UN3 )      "      Inklusives OR
XOR \ ( UN2 UN1 - UN3 )      "      Exklusives OR
NOT  \ ( UN1      - UN2 ) Bitweise Invertierung,
      \ 1er-Komplement
```

Schiebebefehle

```
1SHIFT> \ ( UN1 - UN2 ) ca. 2 U/ z.B. 0100h -> 0080h
4SHIFT> \ ( UN1 - UN2 )    10h U/ z.B. 0100h -> 0010h
8SHIFT> \ ( UN1 - UN2 )   100h U/ z.B. 0100h -> 0001h
1<SHIFT \ ( UN1 - UN2 )    2 U* z.B. 0100h -> 0200h
4<SHIFT \ ( UN1 - UN2 )   10h U* z.B. 0100h -> 1000h
8<SHIFT \ ( UN1 - UN2 )  100h U* z.B. 0010h -> 1000h
```

Mischung mit Addition und Subtraktion ermöglicht schiefe Werte:

```
3* = DUP 1<SHIFT +
5* = DUP 1<SHIFT 1<SHIFT +
```

Es ist offensichtlich nicht günstig Arrays mit einer krummen Anzahl von Bytes z.B. 3 zu erzeugen. Besser sind 2, 4, und 8 Bytes, weil die Multiplikationen dafür leicht realisiert werden können.

Speicherzugriff

In Tabelle 4 sind die in FORTH üblichen Speicherzugriffsbefehle aufgeführt. Wegen des FLASHs mußten spezielle Befehle in Tabelle 5 definiert werden. Vgl. auch Kapitel 1.

Inkrementieren einer Speicherzelle ist mit +! und +C! möglich.

Wenn N1 eine negative Zahl ist, wird daraus eine Subtraktion um N1. Der Wert -1 wird dafür häufig verwendet, sodaß er als Konstante definiert ist.

Die 16 Bit Variable >C zeigt im Programmspeicher auf das nächste freie Byte. Daten werden meist sequentiell in den Programmspeicher geschrieben, d.h. an >C abgelegt und dann wird >C inkrementiert. Das ist die Funktion der Befehle , und C, . Verwendung insbesondere in Zusammenhang mit TABLE zur Erzeugung von Tabellen im Programmspeicher.

Bei den 16 Bit Befehlen ist die Reihenfolge der Bytes abhängig von der CPU. Beim 68HC08 wird zuerst das obere Byte, dann das untere Byte abgelegt („big-endian“):

Tabelle 4: Speicherzugriff

```
! \ ( UN1 Addr1 - ) Schreiben 16 Bit Wort
@ \ (      Addr1 - UN1 ) Lesen 16 Bit Wort
C! \ ( C1 Addr1 - ) Schreiben eines Bytes
C@ \ (      Addr1 - C1 ) Lesen eines Bytes
```

Mit Inkrement:

```
+! \ ( N1 Addr1 - ) Addr1: 16 Bit Variable
+C! \ ( N1 Addr1 - ) Addr1: Byte Variable
```

Schreiben des Programmspeichers (auch gelöschtes FLASH):

```
, \ ( N1      - ) compile N1 ( erased FLASH, RAM )
C, \ ( C1      - ) compile C1
```

Tabelle 5: Schreibzugriff FLASH

```
F! \ ( UN1 Addr1 - ) für gelöschtes FLASH
FC! \ ( C1 Addr1 - )
E! \ ( UN1 Addr1 - ) für „EEPROM“
EC! \ ( C1 Addr1 - )
!' \ ( UN1 Addr1 - ) für gelöschtes FLASH & RAM
C!' \ ( C1 Addr1 - )
```

FLASH

Während man FLASH problemlos mit @ und C@ lesen kann, sind für das Schreiben die Befehle F! und FC! nötig. Dabei wird vorausgesetzt, daß diese FLASH-Bytes auf FF gelöscht sind.

Die Befehle !' und C!' und auch , und C, passen ihr Verhalten automatisch an wenn sie auf RAM oder FLASH zugreifen. Sie können das anhand der Adresse erkennen.

Die Befehle E! und EC! überschreiben bereits beschriebenes FLASH.

```
---- ---- ---- | HEX 2345 200 ! \ 2345 in 0200 speichern
---- ---- ---- | 200 C@ 201 C@
---- 0023 0045 |
```

Bitbefehle

Für Bits in RAM oder I/O
 C1 = 0 - 7 und definiert das Bit auf das sich der Befehl bezieht. Addr1 zeigt auf das behandelte Byte.

Sie sind sinnvoll wenn Speicher knapp und Geschwindigkeit unkritisch ist. Normalerweise ist es günstiger ein ganzes Byte im Speicher als Flag zu verwenden.

Tabelle 6: Bitbefehle

B@ \ (Addr1 C1 - F1) Bit lesen
 B1! \ (Addr1 C1 -) Bit setzen
 B0! \ (Addr1 C1 -) Bit löschen

Die Tabelle BS-TAB wurde zur Definition der Befehle benötigt. Ihr Verwendung ist oft auch in Anwendungsprogrammen sinnvoll.

TABLE BS-TAB
 01 C, 02 C, 04 C, 08 C,
 10 C, 20 C, 40 C, 80 C,

Konstanten

Konstanten geben Zahlen Namen. Details zu gültigen Namen vgl. S. 2.6. Der Inhalt der Konstante liegt im Programmspeicher.

Man kann mit DCONSTANT nicht nur 32 Bit Werte sondern auch Bitkonstanten z.B. auch für Portpins erzeugen.

Tabellen werden mit TABLE definiert und C, und , gefüllt. Ihr Inhalt liegt im Programmspeicher. , legt einen 16 Bit Werte in den Speicher, deshalb erhöht sich der Adreßoffset jeweils + 2 . Hingegen legt C, nur Bytes in den Speicher.

Mit dem Befehl , " . . . " kann man ASCII's in Tabellen füllen:

TABLE MYTABLE , " Hallo"
 , " world"

Der Befehl , " braucht ein folgendes Leerzeichen das nicht abgespei-

Tabelle 7: Konstanten

UN1 CONSTANT name
 UD1 DCONSTANT name \ funktioniert auch für zwei UN
 TABLE \ erzeugt Kopf für Tabelle

```

---- ---- ---- | 1234 CONSTANT MYTEST \ compilieren
---- ---- ---- | MYTEST \ ausführen
---- ---- 1234 |
---- ---- ---- | PB 5 DCONSTANT MYPIN \ Portpin PB 5
---- ---- ---- | MYPIN
---- 0001 0005 |
---- ---- ---- | TABLE MYTABLE A000 , B000 , C000 ,
---- ---- ---- | MYTABLE 2+ @
---- ---- B000 |
---- ---- ---- | MYTABLE @
---- ---- A000 |
---- ---- ---- | TABLE MYTABLE 3B C, 2C C, 11 C,
---- ---- ---- | MYTABLE 2+ C@
---- ---- 0011 |
  
```

chert wird. Der Begrenzer " am begrenzt, das reicht aber für Zeilen Ende braucht es nicht. mit 80 Zeichen. Außerdem kann Die Länge des Textes in einer Zeile man ja weitere Zeilen anhängen. ist technisch auf etwa 250d Zeichen

Variablen

Während Konstanten und Tabellen Daten im Programmspeicher ablegen, reservieren Variablen Speicherplatz im RAM. N1 ist die Zahl der Bytes die reserviert werden sollen.

Im Beispiel wurden zwei Bytes reserviert. Wenn man MYRAM ausführt, legt es nur die Adresse des ersten reservierten Bytes auf den Stack. Das Lesen oder Schreiben der Variable müssen Speicherbefehle ausführen.

Normalerweise vermeidet man das Anlegen von Verlegenheitsvariablen und hält alle Werte auf dem Stack.

Tabelle 8: Variablen

N1 VARIABLE name \ Reserviert RAM
 N1 ZVARIABLE name \ Reserviert RAM in ZeroPage

```

---- ---- ---- | 2 VARIABLE MYRAM \ compilieren
---- ---- ---- | 33 MYRAM ! \ einspeichern
---- ---- ---- | MYRAM @ \ auslesen
---- ---- 0033 | H
  
```

Wenn jedoch zuviel Werte auf dem Stack bewegt werden müssen sind Variablen nicht nur wegen der Übersichtlichkeit, sondern auch wegen der Geschwindigkeit günstiger.

VARIABLE belegt RAM oberhalb der ZeroPage. Als Zeiger auf das nächste frei Byte dient für die ZeroPage die 16 Bit Variable >ZV und für VARIABLE die 16 Bit Variable >V.

Einzelne Variablen werden mit ZVARIABLE in der ZeroPage angelegt. Insbesondere für Assemblerprogramme die entsprechende Adressierung verwenden können.

Wenn man viel RAM braucht, sollte man die Startadresse nicht mit VARIABLE sondern explizit mit CONSTANT z.B. bei FBUF festlegen. Der Inhalt des RAM wird nach COLD-

Reset von FORTH mit 00 überschrieben. Initialisierung von Variablen sollte zusätzlich im Anwenderprogramm erfolgen.

Zum Initialisieren eignen sich die Befehle in Tabelle 10. FILL schreibt ein Muster.

Mit CMOVE wird ein Speicherblock der N1 Bytes Umfang hat von Addr1 nach Addr2 kopiert. Das erste Byte des Blocks an Addr1 wird zuerst bewegt.

Bei CMOVE> wird das letzte Byte zuerst bewegt. Selten benötigt, deshalb nicht im Grundsystem,

Tabelle 9: Speicherblöcke im RAM bearbeiten

```

\ Ab Addr1 N1 Bytes mit Muster C1 füllen
FILL \ ( Addr1 N1 C1 --- ) N1 = 0001 - FFFF
\ N1 Bytes ab Addr1 nach Addr2 verschieben
CMOVE \ ( Addr1 Addr2 N1 --- ) N1 = 0001 - FFFF
\ Addr1 zuerst bewegt

```

sondern nur als Erweiterung verfügbar. Will man sich überlappende Bereiche im Speicher hochschieben braucht man CMOVE> aber. In diesem besonderen Fall würde nämlich CMOVE den Speicher überschreiben bevor es ihn kopiert.

Man kann CMOVE benutzen um RAM aus einer Tabelle im FLASH zu initialisieren. Man sollte aber nicht I/O-Register so laden, da bei diesen oft Bytes in bestimmter Reihenfolge geschrieben werden müssen.

Flags

Während bei den boolschen Bitbefehlen alle 16 Bits einzeln behandelt werden, wird bei den boolschen Befehlen für Flags der gesamte Stackwert als ein „Bit“ angesehen. Er ist als Flag 0, wenn alle 16 Bits 0 sind.

Die boolschen Bitbefehle können natürlich auch zur Erzeugung von Flags verwendet werden:

```
02 AND \ ( UN1 - F1 )
```

Tabelle 10: Vergleichsbefehle

```

= \ ( UN2 UN1 - F1 ) F1 = 1 wenn UN2 = UN1
U< \ ( UN2 UN1 - F1 ) „ UN2 < UN1
U> \ ( UN2 UN1 - F1 ) „ UN2 > UN1
0= \ ( UN1 - F1 ) „ UN1 = 0
: 0< H% 8000 AND ; \ ( N1 - F1 ) nur 2er-Kompl.

```

Tabelle 11: Flags verknüpfen

```

LNOT \ ( F1 - F2 ) Flag invertieren. Alias zu 0=
LAND \ ( F2 F1 - F3 ) F3 = 1 wenn F1 und F2 1 sind
LOR \ ( F2 F1 - F3 ) F3 = 0 wenn F1 und F2 0 sind
\ Alias zu OR
: LXOR IF LNOT THEN ;

```

Terminal-I/O

Tabelle 12:

Fehlermeldungen

```

ERROR \ ( C1 - ) Bricht immer ab
?ERROR \ ( F1 C1 - ) Bricht ab, wenn F1 = 1
\ bei F1 = 0 läuft das Programm weiter
/?ERROR \ ( F1 C1 - ) Bricht ab, wenn F1 = 0
\ bei F1 = 1 läuft das Programm weiter

```

Verzögerung

```

MSEC \ ( UN1 - ) Verzögerung um UN1 Millisekunden
\ UN1 = 1 - FFFF

```

Lesen von UART

```

KEY \ ( - C1 ) Byte von V24 empfangen.
\ Bleibt stehen und wartet auf Byte.

```

Schreiben auf UART

```

EMIT \ ( C1 - ) Byte auf V24 senden
SPACE \ ( - ) SPACE 20h ausgeben
SPACES \ ( UN1 - ) UN1 mal SPACE ausgeben
\ UN1 = 0001 - FFFF
CR \ ( - ) Steuerzeichen für Zeilende ausgeben

```

ERROR unterbricht den Programmablauf, druckt C1 als Hexzahl aus und führt einen WARM-Reset aus. Eine Liste der vom System benutzten Fehlernummern findet sich Seite 2.11. Die beiden anderen Befehle führen nur abhängig von Flag ERROR aus.

MSEC beruht auf Verzögerungsschleife und hat kein exaktes Timing. Das Timing wird durch die EEPROM-Konstante UDLY gesteuert.

Für die V24 ist normalerweise 9600 Baud 8N1 eingestellt.

CR gibt mindestens 0Dh aus. Ob es zusätzlich 0Ah sendet, hängt davon ab ob es vom Terminal auch 0Dh 0Ah empfangen hat. Bzw. ob die EEPROM-Konstante LF' aktiviert wurde.

Zahlen ausgeben:

Hex

NH. \ (UN1 -) als 4stellige Hexzahl mit Vornullen ausdrucken. Es folgt ein SPACE
 \
 CH. \ (C1 -) als 2stellige Hexzahl mit Vornullen ausdrucken. Mit SPACE
 \
 2-DIGIT-EMIT \ (C1 -) CH. aber ohne den SPACE
 DIGIT \ (C1 - ASCII) Macht aus C1: 00 - 0F eine
 \ ASCII-Hex-Ziffer

Binär

NB. \ (UN1 -) 16 Bit Binär ausdrucken. Mit SPACE
 CB. \ (UC1 -) 8 Bit Binär ausdrucken. Mit SPACE

Dezimal

ND. \ (UN1 -) prints 5 chars, rightadjusted,
 \ without leading SPACES
 SND. \ (N1 -) prints sign, 5 chars rightadjusted,
 \ without leading SPACES

String ausgeben:

. " HALLO "

DIGIT und 2-DIGIT-EMIT wurden zur Definition von für CH. und NH. benötigt, können aber auch separat nützlich sein.

Die Ausgabe eines ASCII-Strings über EMIT erfolgt mit ". . . . ". Hinter ." muß ein SPACE folgen, vor dem Begrenzbyte ist keiner nötig (Vgl. S. 2.4 ". . . . "). Beispiel:

: TEST ." HALLO " ;

TEST wird HALLO auf das Terminal ausgeben. Funktioniert also „state-smart“ auch während Compilierung.

Befehle definieren

Befehle werden in FORTH oft auch als „Words“ bezeichnet. Sie entsprechen einem Unterprogramm in anderen Programmiersprachen. Es besteht in FORTH allerdings kein Unterschied zwischen Haupt- und Unterprogrammen. Was beim Testen den Vorteil hat, daß man beliebige Teilprogramme ausführen kann.

```
: name \ ( - )
... ;
```

: ist der Anfang des Befehls, darauf folgt der Name . Das Ende wird mit ; markiert. Dazwischen dürfen Zahlen und andere Befehle stehen. Der Stackkommentar (-) ist nicht zwingend erforderlich, aber sehr empfehlenswert.

Der Name kann maximal 16 Zeichen lang sein und darf keiner Hexzahl entsprechen. Die Einschränkungen bezüglich der verwendbaren ASCII's sind S. 2.9 im Parser beschrieben. Existiert ein Befehl mit gleichem Namen bereits im System wird dieser überschrieben. Beim Compilieren wird das mit ! quittiert. Normalerweise erscheint + , wenn der Name neu ist. Schon compilierte Software

```
---- ---- ---- | : 5+ 5 + ; \ ( N1 - N2 ) N2 = N1 + 5
---- ---- ---- | 4 5+
---- ---- 0009 |
```

wird mit dem alten Befehl weiterlaufen. Neu compilierte Software wird sich jedoch den neuen Befehl verwenden. Das Überschreiben alter Befehle ist also möglich, sollte aber meist vermieden werden.

Der Aufruf eines Befehls im Interpretermodus von FORTH erfolgt einfach durch Eintippen von Zahlen und Befehlen und Abschluß durch die RETURN-Taste. Der Aufruf innerhalb eines anderen Befehls erfolgt ebenfalls formlos:

```
: 9+ 4 + 5 + ;
```

Assemblerbefehle werden mit :CODE name ... CODE; erzeugt. Der Aufruf und die Ausführung von beiden erfolgt jedoch identisch. Sind sie erst einmal compiliert, ist kein Unterschied mehr feststellbar.

Solange man am Terminal Befehle einzeln eingibt und ihre Wirkung auf Stack und Speicher überprüfen kann ist Fehlersuche recht einfach. Auch während :CODE ... CODE; bleibt das System im Interpreterzustand. In : name ... ; schaltet es jedoch auf Compilermodus um.

Befehle dann werden nicht ausgeführt, sondern in den Speicher compiliert. Man kann dann jedoch zeitweise mit [in den gewohnten Interpreterzustand umschalten. Befehle werden jetzt ausgeführt, nicht compiliert. Dann sind Disassembler und Hexdumps möglich um sich den Speicher anzusehen. Zurückschalten ins Compilieren erfolgt mit] .

Das ist auch nützlich um „inline“ berechnete Konstanten zu compilieren:

```
3 CONSTANT DREI
: ACHT \ ( --- 08 )
[ 5 DREI + CODE-LIT ] ;
```

Es wird 8 im Interpretermodus berechnet und dann durch CODE-LIT als Literal compiliert.

Steuerung des Programmablaufs

FORTH kennt kein GOTO, hat aber sonst diverse Möglichkeiten den Programmablauf zu steuern. Strukturierung zwingt oft zu mehr Nachdenken beim Schreiben, erhöht jedoch dann die Funktionssicherheit der Software beträchtlich.

Die Verzweigungen können nur innerhalb von `: ... ;` verwendet werden, sie sind nicht im Interpretermodus ausführbar. Steuerung des Programmfloßes erfolgt über Flags die auf dem Stack bereitliegen müssen.

Bedingte Sprünge

`IF` holt ein Flag vom Stack. Wenn `F1 = 1` werden die Befehle zwischen `IF` und `THEN` ausgeführt. Danach die Befehle die hinter `THEN` stehen.

```
... \ ( - F1 )
IF \ bzw. IF.
... \ wenn F1 = 1
THEN
```

Hier wird bei `F1 = 0` der Abschnitt zwischen `ELSE` und `THEN` ausgeführt:

```
... \ ( - F1 )
IF \ bzw. IF.
... \ wenn F1 = 1
ELSE \ bzw. ELSE.
... \ wenn F1 = 0
THEN
```

`IF` und `ELSE` haben nur etwa 128d Byte Reichweite, was in den allermeisten Fällen genügt. In einigen Fällen sind sie deshalb durch die Varianten `IF.` bzw. `ELSE.` zu ersetzen, die den gesamten Speicher erreichen.

Schleifen

```
BEGIN
... \ ( - F1 )
UNTIL
```

Beispiel: `BEGIN ... UNTIL`

```
---- ---- ---- | : TEST 3 BEGIN 1- DUP DUP LNOT UNTIL DROP ;
---- ---- ---- | TEST
0002 0001 0000 |
```

Beispiel: `DO ... LOOP`

```
---- ---- ---- | : TEST 6 4 DO I LOOP ;
---- ---- ---- | TEST
0004 0005 0006 |
```

`UNTIL` holt sich ein Flag. Wenn `F1 = 0` ist wird zu `BEGIN` zurückgesprungen. Sonst wird die Schleife verlassen und die Befehle nach `UNTIL` werden ausgeführt.

```
BEGIN
... \ Auf ewig
AGAIN
```

`AGAIN` springt immer zurück zu `BEGIN`. Diese Schleife wird niemals verlassen.

DO ... LOOPS

```
... \ ( - UN2 UN1 )
DO
...
LOOP
```

`UN1` gibt den Startwert, `UN2` den Endwert der Schleife vor. `UN2` muß deshalb größer als `UN1` sein. Innerhalb der Schleife, kann man mit `I` den Zähler auslesen:

```
I \ ( - UN3 )
```

`I` hat also ursprünglich den Wert von `UN1`. Wird dann in jedem Durchlauf inkrementiert, und hat als letzten Wert `UN2`. Dann wird die Schleife verlas-

Beispiel: `DO ... +LOOP`

```
---- ---- ---- | : TEST 6 2 DO I 2 +LOOP ;
---- ---- ---- | TEST
0002 0004 0006 |
```

sen. `DO ...` `LOOPS` sind beliebig tief schachtelbar (zumindest bis der CPU-Stack überläuft), aber `I` gibt immer nur den Wert der innersten Schleife.

Mit `+LOOP` ist statt Inkrementierung `+1` auch `+UN4` möglich. Wenn `UN4` eine negative Zahl ist, z.B. `-1`, erfolgt effektiv eine Subtraktion. Man sollte darauf achten, daß der Zielwert `UN2` exakt getroffen wird.

```
DO
... \ ( - UN4 )
+LOOP
```

Mit `LEAVE` wird ein vorgezogener Abbruch erzwungen. Allerdings wird nicht von `LEAVE` direkt herausgesprungen. Der Abbruch wird in `LOOP` durchgeführt, wenn das Programm `LOOP` das nächste mal erreicht.

```
DO
... \ ( - F1 )
IF LEAVE THEN
...
LOOP
```

`DO...LOOP` ist `BEGIN...UNTIL` überlegen, weil der Stack nicht mit einer Zählervariable verstopft wird. Es ist jedoch langsamer.

Beispiel : Sprungzieltabelle

```
: HERE 1 ;
: THERE 2 ;
: ANYWHERE 3 ;
```

```
TABLE MYJUMP
^ HERE ,
^ THERE ,
^ ANYWHERE ,
```

```
: ?WHERE \ ( C1 - )
\ C1 = 0, 2, 4
MYJUMP + @ EXECUTE ;
```

Verzweigung über Tabellen

Mit dem Befehl `TABLE` kann man Sprungtabellen anlegen wie sie z.B. für Zustandsmaschinen gebraucht werden.

`^ name` bestimmt die CFA-Adresse des Befehls und legt sie auf den Stack `EXECUTE` führt eine CFA-Adresse aus die es auf dem Stack vorfindet.

Im Beispiel wird anhand von `C1` über `?WHERE` eines der drei Programme ausgeführt.

Sourcecode kompilieren

Sourcecode wird mit einem Editor als Textfile auf einem PC erstellt und dann im Terminalprogramm als Klartext hochgeladen. Am Anfang und Ende des Files müssen dafür diese FORTH-Befehle stehen:

```
<| \ ( - ) Fileanfang
|> \ ( - ) Fileende
```

Zwischen ihnen wird das Echo der V24 abgeschaltet und nur die Compilermeldungen angezeigt. Ein Beispielprogramm wäre:

```
<| \ Test
HEX
1 ^ \ optional FileMarker
   \ setzen
4F CONSTANT Y
: X ." HALL" Y EMIT CR ;
| \ SaveMarker
|>
```

Hinter |> muß ein CR oder SPACE folgen, damit der Compiler sofort started. Dann erfolgt der Upload durch das Terminalprogramm. Der Compiler wird den Zugang der neuen Befehle so melden:

```
+ ^
+ Y
+ X
— — — |
```

Am Anfang des eigentlichen Sourcecodes ist die Einstellung der Zahlenbasis sinnvoll: HEX oder DECIMAL. Man kann natürlich im Text immer noch bei Bedarf hin- und herschalten.

Der Programmanfang, normalerweise FLASH ab 8100 kann am Anfang des ersten Files modifiziert werden. Z.B. kann man versuchsweise ins RAM bei FCODE kompilieren

```
FCODE >C !
```

Am Anfang eines Programms werden Konstanten und Variablen definiert. Es folgen die „Primitives“, Befehle die wenig komplex sind und

in sich kaum Programmverzweigungen haben. Z.B. für den Zugriff auf Ports. Häufig werden sie nicht in FORTH, sondern in Assembler definiert. Am Ende erscheinen die komplexen Befehle.

Da FORTH nur einen Compilerdurchlauf hat, muß jeder Befehl definiert worden sein, bevor er von einem anderem Befehl verwendet werden kann. Deshalb hat FORTH auch keinen eigenen Linker. Das ist keine so große Einschränkung wie es den Anschein hat. In nanoFORTH selbst z.B. mußte nur ERROR gepatcht werden.

Umfangreiche Anwendungen müssen normalerweise in mehrere Files zerlegt werden. Um beim Kompilieren die Orientierung nicht zu verlieren, sollte man Ende des Files durch eine Hilfsmeldung das nächste File anzeigen. Z.B.:

```
." -> TEST2.F08"
```

FileMarker

Wenn man in einem File etwas ändert, muß man es und die nachfolgenden neu kompilieren.

Die neuen Befehle gleichen Namens würden dann in der Befehlsliste angehängt. Die alten Befehle sind dann zwar unwirksam, verbrauchen aber Programmspeicher. Auch RAM das mit ZVARIABLE und VARIABLE belegt wurde ist nicht mehr verfügbar.

Besonders bei umfangreichen Applikationen die aus mehreren Files bestehen will man aber, daß das alte Programm gelöscht wird und das RAM freigegeben wird. Dafür wird in der Entwicklungsphase den Marker-Befehl ^ am Fileanfang verwendet

```
^ \ ( C1 - )
```

Wobei C1 eine Zahl von 01 - FF ist, die typisch der Nummer des Files entspricht.

Der Befehl ^ sucht ob es schon einen Marker mit dieser Nummer gibt. Wenn ja löscht er das FLASH des Programmspeichers bis zu diesem und gibt mit VARIABLE und ZVARIABLE belegtes RAM frei.

Danach kompiliert er an einen neuen Marker-Befehl der wieder die aktuellen Werte der Zeiger >ZV und >V enthält.

Man kann einen Marker durch den Befehl

```
FORGET \ ( C1 - )
```

gezielt löschen. C1 ist die Nummer des mit ^ markierten Files.

```
0 FORGET
```

löscht den Applikationsspeicher komplett.

SaveMarker

Alternativ zu SAVE kann man den Marker | mit identischer Funktion auch am Ende eines Files anbringen. Kompiliert schneller und schon das „EEPROM“. Wenn der Bereich oberhalb bis B7FF gelöscht FF ist, wird dieser Marker nach Resets gefunden und das RAM mit seinen Werten geladen.

Fehlersuche

```
WORDS \ ( - )
```

Mit WORDS kann man sich den Inhalt der Befehlsliste ansehen. Es werden 10d Namen ausgegeben. Angefangen wird bei den neuesten Befehlen. Will man mehr sehen, drückt man SPACE. Abgebrochen wird mit jeder anderen Taste.

Für Assemblerliste: [CODE WORDS UNC name

UNC disassembliert den FORTH- oder Assemblerbefehl name. Es werden 10d Zeilen ausgegeben. Mit SPACE wird weiterdisassembliert, mit jeder anderen Taste abgebrochen.

```
DIS \ ( Addr1 - )
```

```
DUMP \ ( Addr1 - )
```

DIS disassembliert ein Assemblerprogramm ab Addr1. DUMP gibt einen Hex-Dump. Bedienung jeweils wie UNC.

Aufräumen kann man mit dem dem Befehl COLD der bezüglich Software einem Hardware-Reset entspricht:

```
— — — | COLD
RESET
— — — |
```

Beispiel: DUMP

```
— — — | E004 DUMP
      0 1 2 3 4 5 6 7 8 9 A B C D E F
E000                56 C4 22 B5 20 A8 C4 22 C5 22 87 32
E010 CA A9 00 CA A9 17 12 20 69 CC 22 1B 20 33 C4 B0
```

Parser

Das System verarbeitet einen kontinuierlich ASCII-Datenstrom über die V24 vom Terminalprogramm auf dem PC.

ASCII

Alle ankommenden Bytes werden als ASCII interpretiert (Tabelle 12). Zeichen die dort nicht definiert sind werden in Leerzeichen 20h umgewandelt. Die Bytes von 21h bis 7Eh sind druckende Zeichen die sich nominell für die Namen von Befehlen eignen. nanoFORTH wandelt jedoch automatisch die Kleinbuchstaben 61h bis 7Ah auf Großbuchstaben um. Das vermeidet Tippfehler beim interaktiven Arbeiten am Terminal. Die Zeichen 7B 7F und 7E, d.h. } { ~ sind auf deutscher Computertastatur bei Laptops oft nicht direkt erreichbar und sollten deshalb vermieden werden. 27h ^ wird in FORTH verwendet. Um Verwechslungen zu vermeiden sollte deshalb 60h ` vermieden werden.

Steuerzeichen

CR („carriage return“) und LF („line feed“) zeigen Zeilenende an. Es gibt mindestens 3 Varianten der Darstellung eines Zeilenendzeichens:

```
CR LF MS-DOS, CP/M
CR Mac, OS-9
LF Unix
```

Beim Empfang wird deshalb im System LF auf CR gewandelt, sodaß Zeilenende sicher erkannt wird. Der einzige Befehl der darauf angewiesen ist, ist der Kommentar \ . Saubere Erkennung ist aber auch im Terminalmodus des Systems erforderlich.

Tabelle 12: ASCII-Zeichen

	0	1	2	3	4	5	6	7
0 0000					SP	@	P	` p
1 0001		XON	!	1	A	Q	a	q
2 0010			„	2	B	R	b	r
3 0011		XOFF	#	3	C	S	c	s
4 0100			\$	4	D	T	d	t
5 0101			%	5	E	U	e	u
6 0110			&	6	F	V	f	v
7 0111			^	7	G	W	g	w
8 1000	BS		(8	H	X	h	x
9 1001)	9	I	Y	i	y
A 1010	LF		*	:	J	Z	j	z
B 1011			+	;	K	[k	{
C 1100			,	<	L	\	l	
D 1101	CR		-	=	M]	m	}
E 1110			.	>	N	^	n	~
F 1111			/	?	O	_	o	DEL

Beim Senden durch den Befehl CR wird ein Zeichen CR gesendet. Wenn der Parser beim Empfang schonmal LF empfangen hat, wird nach CR ein LF angehängt. Alternativ erzwingt die EEPROM-Konstante LF^ das Senden von LF nach CR.

Die Zeichen BS („backspace“) und DEL („delete“) führen zum Löschen eines Zeichens im Terminalbetrieb.

Zur Untersuchung von Problemen mit Tastencodes an Terminalprogrammen sind KEY und EMIT hilfreich.

XON und XOFF werden für Datenflußsteuerung bei Übertragung von Files verwendet. Das System sendet selbst XON bzw. XOFF, aber ignoriert deren Empfang. Wenn man das Senden bremsen muß sollte man das z.B. mit dem Verzögerungsbefehl MSEC machen.

Terminalbetrieb

XON und XOFF sind ausgeschaltet und links wird der Stackinhalt angezeigt. Die von der Tastatur eingegebenen Texte können mit der Backspace-Taste korrigiert werden. Erst durch einen CR erfolgt die Verarbeitung.

Laden von Files

Innerhalb der Klammern <| und >| ist das System auf das Laden und Compilieren von ASCII-Files eingestellt. Da TIB („Terminal Input Buffer“) nur 64d Byte umfaßt wird typisch nach Empfang eines Namens ein XOFF gesendet und dieser dann verarbeitet. Danach wird mit Steuerzeichen XON das Senden neuer Daten angefordert.

Zeit- & Speicherbedarf

Tabelle 13: Laufzeit der Nucleus-Befehle

	Byte	us		Byte	us		Byte	us		Byte	us
!	3	71	8SHIFT>	3	16	DROP	3	11	LOOP	3	39
+	3	28	=	3	37	DUP	3	23	LOR	3	30
+!	3	95	@	3	65	ELSE	5	19	NOT	3	16
+C!	3	62	AGAIN	6	3	ELSE.	6	19	OR	3	27
+LOOP	3	54	AND	3	27	EXECUTE	3	27	OVER	3	23
-	3	28	B0!	3	73	FILL	3		ROT	3	55
0=	3	29	B1!	3	72	10h bytes	490		SWAP	3	43
1+	3	19	B@	3	59	100h bytes	14k		THEN	0	0
1-	3	23	BEGIN	0	0	HOPP	3	23	U*	3	65
1<SHIFT	3	16	C!	3	44	I	3	21	U/	3	980
1SHIFT>	3	16	C@	3	42	IF	5	19	U/MOD	3	980
2+	3	24	CMOVE	3		IF.	6	19	U<	3	34
2-	3	24	10h byte	650		LAND	3	30	U>	3	34
2DUP	3	46	100h byte	9k		LEAVE	3	17	UNTIL	6	19
2DROP	3	13	DO	3	75	literal:			variable	3	19
2OVER	3	37	constant	3		16 bit	8	10	XOR	3	27
2SWAP	3	109	cyc=literal+9			8 bit	6	8	zvariable	3	17
4<SHIFT	3	37	dconstant	3		0000h	4	6			
4SHIFT>	3	37	cyc=2*literal+9			LNOT	3	26			
8<SHIFT	3	17									

Der Speicherbedarf ist in Byte bezogen auf Compilierung in einem Befehl angegeben.

Die Laufzeit in Mikrosekunden bezieht sich auf 2,45 MHz Busfrequenz. Bei datenabhängig variabler Laufzeit ist ca. die maximale Laufzeit angegeben.

Befehle wie FILL und CMOVE sind natürlich stark abhängig vom Umfang des zu verarbeitenden Speicherbereichs, weshalb hier zwei Werte als grobe Orientierung angegeben sind.

Literals werden „inline“ compiliert. Eine Zahl wie 1234 ist nicht so effizient darstellbar wie 0012 oder 0000.

Laufzeitmessung bei komplexeren Befehlen erfolgte über das angegebene Testprogramm das vor Ausführung des Befehls einen Portpin setzt und danach löscht. Damit kann man bequem per Oszilloskop messen.

<| \ Test Runtime

FBUF >C ! \ compile to RAM
DPB 1 B1! \ Port PB1 = output

```
: X
BEGIN
1 MSEC \ delay
FFFF FFFF \ data to stack
[CODE PB 1 MBS, CODE] \ PB1 = 1
U/MOD \ test
[CODE PB 1 MBC, CODE] \ PB1 = 0
2DROP \ clear stack
AGAIN ;
```

X \ execute immediatly after
\ compilation

|>

Fehlermeldungen

Die in nanoFORTH definierten Fehler (Tabelle 14) belegen die Kennzahlen 00 - 3F und drucken ihre Kennzahl, Name bei dessen Verarbeitung der Fehler auftrat und eine kurze Fehlerbeschreibung.

Die mit ERROR definierten Fehlermeldungen für Anwendungen sollten im Bereich 40 - FF liegen. Sie erzeugen keine Fehlerbeschreibung.

Die aufgeführten Fehler sind in die Gruppen FORTH, Assembler und FLASH eingeteilt. Einige dieser Fehler treten praktisch nie, andere sehr häufig auf. Insbesondere:

FORTH:

- 05 Ein Name konnte nicht in der Befehlsliste gefunden werden.
- 06 Ein neuer Name ist ungültig weil er eine Hexzahl sein könnte.
- 07 Langer Vorwärtssprung nötig: IF durch IF. und ELSE durch ELSE. ersetzen.
- 08, 09 sind Meldungen des Parsers. Können auf Probleme mit dem Terminalprogramm hindeuten. Z.B. Überlauf des TIB-Buffers.

Assembler:

- 47 Sprungweite des relativen Sprungbefehls überschritten: JMP, nehmen
- 4E innerhalb langer Assemblerprogramme mit SOLVE\$ Sprünge auflösen und damit Speicher freigeben. Oder Programmteile in Unterprogramme auslagern.
- 51 Operand ist 16 Bit, hätte aber Byte sein müssen.
- 52 Adressierungsart gibts für den Opcode nicht.

FLASH:

- 73 eventuell muß der Jumper stecken. Immer muß das Interrupt-Flag im CCR disabled sein.
- 76 Compiler erwartete, daß er gelöschtes FLASH schreiben kann.

Tabelle 14: Fehlermeldungen nanoFORTH

Nr.	in Befehl	Fehlerbeschreibung
00	STACK.	FORTH stack
00	?CSP	FORTH stack
01	?EXEC	FORTH not executing
02	?COMP	FORTH not compiling
03	?PAIRS	FORTH token wrong
04	NUMBER?	FORTH base undefined
05	' '	FORTH undefined name
05	COMPILE-STRING	
05	INTERPRET-STRING	
06	(HEADER)	FORTH could be number
07	CODE-BR>	FORTH branch too long
08	INTERPRETER	FORTH TIB overflow
09	INTERPRETER	FORTH string too long
0A	ERASE>	FORTH wrong addr
0B	FORGET^	FORTH undefined ^
47	REL-ERR?	ASM branch too long
47	(R)	
4D	ABS/REL@	ASM branch ?
4E	\$H!	ASM \$HEAP full
4F	\$:	ASM multiple \$:
50	SOLVE\$	ASM \$:/\$ unresolved
51	AERR	ASM operand
52	ML1,	ASM adressing
52	STX,	
52	STA,	
52	J^	
73	IRQ?	FLASH IRQ / jumper
73	SYS?	
73	VECT!	
73	SYSEE!	
74	(ERASE)	FLASH erase failed
75	(EE)	FLASH write failed
75	((FC!))	
76	FC!	FLASH not erased
76	FN!	

Index FORTH-Befehle

!	2.3	>ZV	1.5	ERASE	1.3	PC	1.2
!'	2.3	?ERROR	2.5	ERASE>		PD	1.2
\$	3.3	@	2.3	ERROR	2.5	ROT	2.2
\$\$:	3.3	AGAIN	2.7	EXECUTE	2.7	SAVE	1.6
\$HEAP	1.3	AND	2.3	F!	1.3	SB	
'	2.7	APP	1.2		2.3	SB.	
	4.1	APP-EE	1.2	F,		SCAN-VOC	
''	4.1	B%	2.2	FBUF	1.2	SCAN-VOC^	
(2.1	B0!	2.4	FC!	1.3	SET	
)	2.1	B1!	2.4		2.3	SN	
+	2.2	B@	2.4	FCODE	1.2	SND.	2.6
+!	2.3	BEGIN	2.7	FIND^		SOLVE\$	3.4
+C!	2.3	BS-TAB	2.4	FILL	2.5	SPACE	2.5
+LOOP	2.7	C!	2.3	FLBPR	1.8	SPACES	2.5
,	2.3	C!'	2.3	FLETCHER	1.4	SPSAVE	
,"	2.3	C,	2.3	FN!		STACK.	
-	2.2	C@	2.3	FORGET	2.8	STRING	
-1	2.2	CB.	2.6	FORGET^		SWAP	2.2
."	2.6	CH.	2.6	G\$	3.3	SYS	1.2
/?ERROR	2.5	CHAR@		G\$:	3.3	SYS-EE	1.2
0=	2.5	CMOVE	2.5	GET-BYTE		SYSEE!	1.4
1+	2.2	CODE-LIT	2.6	H%	2.2	T.	
1-	2.2	CODE-JMP		HEX	2.2	TABLE	2.4
1<SHIFT	2.3	CODE-JSR		HOPP	2.2	THEN	2.7
1SHIFT>	2.3	CODE-RTS		I	2.7	TIB	1.2
2+	2.2	CODE;	3.2	IF	2.7	U*	2.2
2-	2.2	CODE]	3.2	IF.	2.7	U/	2.2
2-DIGIT-EMIT		COLD	1.5	J?		U/MOD	2.2
	2.6	CONSTANT	2.4	JUMPER?	1.4	U<	2.5
2DUP	2.2	CR	2.5	KEY	2.5	U>	2.5
2DROP	2.2	D%	2.2	LAND	2.5	UDLY	1.6
2OVER	2.2	DB.	2.6	LATEST	1.5	UNC	2.8
2SWAP	2.2	DCONSTANT	2.4		4.1	UNTIL	2.7
4<SHIFT	2.3	DECIMAL	2.2	LEAVE	2.7	VARIABLE	2.4
4SHIFT>	2.3	DIGIT	2.6	LF'	2.9	VECT!	1.8
500MSEC		DIS	2.8	LNOT	2.5	VECT@	1.8
8<SHIFT	2.3	DO	2.7	LOOP	2.7	VERSION	1.4
8SHIFT>	2.3	DPA	1.2	LOR	2.5	VN	
:	2.6	DPB	1.2	MSEC	2.5	WARM	1.5
:CODE	3.2	DPC	1.2	N	1.2	WORDS	2.8
:EA		DPD	1.2	NB.	2.6	XOR	2.3
;	2.6	DROP	2.2	ND.	2.6	XSAVE	3.2
<	2.8	DUMP	2.8	NEGATE	2.2	ZVARIABLE	2.4
=	2.5	DUP	2.2	NFA.	4.1	[2.6
>C	2.3	E!	1.3	NH.	2.6	[CODE	3.2
>C!		EC!	1.3	NOT	2.3	\	2.1
>V	1.5	ELSE	2.7	NOPC]	2.6
		ELSE.	2.7	OR	2.3	^	2.8
		EMIT	2.5	OVER	2.2		2.8
				PA	1.2	>	2.8
				PB	1.2		