Ingenieurbüro
Klaus Kohl-Schöpe

Handbuch zum

mcFORTH

V1.0

Ein FORTH für viele Microcontroller

- **Das Konzept hinter dem mcFORTH**
- **Details zum mcFORTH**
- **mcFORTH-Tools:**
  - Terminalprogramm
- **Implementierungen:**
  - Virtueller mcFORTH-Prozessor VP32 (Windows)
  - Infineon XMC1xxx (XMC2Go und XMC1xxx-Bootkits)

Die Informationen im vorliegenden Handbuch werden ohne Rücksicht auf vorhandenen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.

Bei der Erstellung dieses Dokumentes wurde mit größter Sorgfalt vorgegangen und die Angaben laufend mit der erstellten Software abgeglichen. Trotzdem können Fehler nicht ausgeschlossen werden. Der Autor und das Ingenieurbüro Klaus Kohl-Schöpe kann deshalb keine juristische Verantwortung noch irgendwelche Haftung für Fehler übernehmen. Für Verbesserungsvorschläge ist der Autor dankbar.

Für private Zwecke und bei geringen Stückzahlen (<10) dürfen Applikationen, die mit mcFORTH entwickelt wurden, kostenlos auch ohne Offenlegung von Sourcen verwendet werden. Bitte fragen Sie nach einer Lizenz, wenn Sie diese Produkte in größerer Stückzahl oder kommerziellen Anwendungen einsetzen wollen und eine professionelle Unterstützung erwarten.

Kontakt:
Klaus Kohl-Schöpe
Prof.-Hamp-Str. 5
D-87745 Eppishausen
Tel. 08266/3609862
EMail: kks@designin.de

Inhaltsverzeichnis

1. Einleitung

Seit über 35 Jahre beschäftige ich mit Microcontroller. Damals waren es einer der ersten bezahlbaren, auf 6502 basierenden Rechner mit 8KByte RAM und Basic-Interpreter. Heute sind es schon GHz-schnelle Multicore-Prozessoren unter Betriebssysteme wie Linux, Android oder Windows.

In meiner beruflichen Laufbahn als Entwickler für Meßsysteme und anschließend als Microcontroller-FAE (**F**ield **A**pplication **E**ngineer = technischer Berater für Kunden) bei den größten Bauteile-Distributoren programmierte ich eine fast endlosen Liste von Prozessorfamilien wie x51, 6502, Z80, HC(S)08/11/12, MSP430, PIC16/24/32, AVR(32), 78K0(R), 78K4, Z8, C166/ST10, H8(S), R8C/M16C/M32C/R32C, x86 (8088-... Atom), ARM 7/9/11/M0(+)/M3/M4/R4/A5/A8/A9, 68K, Coldfire, PowerPC, SuperH, V850, RL78, RX, TMS320-DSP's und einigen FORTH-Prozessoren wie Zilog Super8, Novics NC4000 oder Harris RTX2000 in Assembler, BASIC, Pascal, Java, C und natürlich FORTH.

1.1. Vorgeschichte zum mcFORTH

Die Programmiersprache FORTH nutze ich dabei auch schon über 25 Jahre für private und berufliche Anwendungen. Auch da habe ich die Entwicklung von FIG-FORTH über diverse F83-Varianten (einschließlich volksFORTH) bis hin zu F-PC, GFORTH und WinFORTH miterlebt und teilweise aktiv unterstützt. Eigene FORTH-Versionen wie Super8-FORTH oder das KK-FORTH für x86, RTX, Z80 und 68K sind die logische Folge.

Dabei gärte im Hintergrund immer der Wunsch, ein einheitliches FORTH für möglichst viele Microcontroller-Familien zu haben, um Applikationen schnell auf die diversen Plattformen zu bringen. Da oft im ersten Schritt keine Erfahrung mit dem Zielprozessor und auch keine Tools wie FORTH-Assembler zur Verfügung stehen, spielt dabei die Idee eines virtuellen Prozessors, der in C realisiert schnell angepasst werden kann, eine wichtige Rolle.

Aber die Umsetzung dauerte vom ersten Konzept bis zu der hier vorliegenden Version v1.0 inzwischen schon über 10 Jahre und wurde beeinflusst von meinen Erfahrungen aus dem KK-FORTH aber auch den vielen Ideen des ANS-FORTH. Die Vorgaben für das jetzt vorliegende mcFORTH (Microcontroller-FORTH) war, dass es sehr leicht erlernbar und gut portierbar ist, ohne dabei die Vorteile von FORTH wie Kompaktheit und vor allem interaktives Arbeiten verloren geht.

Dazu wurden folgenden Eigenschaften definiert:

- ROM- bzw. Flash-Fähig (getrennter Variablenbereich)
- Kompakt (das Image sollte klein sein – zwischen 16-32K gewünscht)
- Portabel (nur wenige Befehle in Assembler – oder Bytecode-Interpreter in C für VP16/32)
- Leicht erlernbar (Befehlssatz überschaubar und angelehnt an ANS- und F83-FORTH)
- Sowohl 16-, 32- als auch 64Bit-Version möglich (bei 8-64 Bit-Befehlscode)
- HEAP für temporäre Header und Programme verfügbar

Da ich mich inzwischen meist mit 32Bit-Controller befasse und diese auch schon für <1€ erhältlich sind, ist ein 32Bit-FORTH die Ausgangsbasis für den Targetcompiler und für ein vollständiges Entwicklungssystem (Terminal mit Editor und Fileserver).

Jedoch gibt es parallel zum Vollausbau eine kleine mcFORTH-Variante für Microcontroller mit Speicher aber 16K Flash und 1K RAM. Dieses hat deutlich weniger Befehle und kein Fileinterface. Programme müssen deshalb über ein Terminal (z.B. mit Copy/Paste) heruntergeladen werden.

Eine dritte Variante für ganz kleine Mikrocontroller oder Entwicklungen, den man die FORTH-Basis nicht mehr ansehen soll, wird später folgen. In dieser Version werden nur die in der Applikation benötigten Programmteile im Target sein und die Compilierung erfolgt durch ein getrenntes mcFORTH/Terminal-Programm. Damit sollten dann Programme ab 1K Flash und 256 Byte RAM möglich sein.

1.2. Dokumentation

Dieses Handbuch beschreibt das generelle Konzept von mcFORTH und deren Implementierung auf den virtuellen FORTH-Prozessoren VP32. Außerdem wird ein in MASM32 aber auch in C realisierter Simulator für den VP32 erläutert. Zusätzlich gibt es für die einzelnen Implementierungen jeweils ein eigenes Anwender-Handbuch, dass auch in Kurzform den Befehlssatz und die Besonderheiten erwähnen.

1.3. Verfügbarkeit der Sourcen und Kosten

Momentan wird der Sourcecode für das mcFORTH und den Targetcompiler nicht weitergegeben, um die Portabilität zu erhalten und einen Wildwuchs der Varianten zu vermeiden. Bei den Lizenzen wird eine Politik verfolgt, die eine kostenlose Evaluierung und die private Verwendung ohne Gebühren erlauben. Erst wenn die Systemanzahl eine Grenze überschreitet (hier willkürlich mit 10 Stück angegeben) oder in kommerzielle Produkten verwendet wird, muss man eine günstigen Lizenz erwerben. Zusätzlich zu den Lizenzen gibt es auch ein Support-Model, welches in der höchsten Stufe auch die Auslieferung der Sourcecode und des Targetcompiler beinhaltet:

Einzel-Lizenzen pro Stück:

Stückzahlen	Preis/Stück	Kommentar
10-99	0.50€	Support über Email
100-999	0.40€	“
1000-9999	0.30€	“
10000+	0.20€	“

Bei diesen Lizenzen ist ein Report über verkauften Stückzahlen notwendig.

Lizenzpakete:

Paket	Preis	Kommentar
Ohne Stückzahlbegrenzung	5000€	Preis pro Gerät
Targetcompiler für ein mcF	10000€	Für beliebig viele Geräte
Targetcompiler-Sourcen	20000€	“ (mit den Sourcen für ein mcF)
Vor-Ort-Support	500€/Tag	zuzügig Reisekosten
Vor-Ort-Support	2000€/Woche	zuzügig Reisekosten

Bei den Lizenzpaketen ist neben dem Email-Support direkter Support für einen Tag enthalten. Reisekosten werden dazu getrennt abgerechnet. Bei einem Lizenzpaket ohne Stückzahlbegrenzung gilt die Vereinbarung nur für ein Gerät einer Firma. Sollten mehrere Geräte in hoher Stückzahl realisiert werden, so sollten Sie den Targetcompiler mit der Source für eine Architektur erwerben. Sie können dann dieses mcFORTH besser anpassen und für eine beliebige Anzahl verschiedener Geräte nutzen. Die Sourcen des Targetcompilers sind in diesem Paket noch nicht enthalten, können

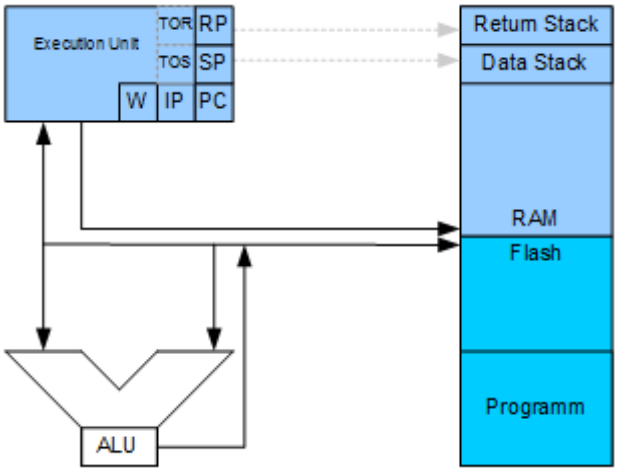
jedoch bei dem größten Lizenzpaket erworben werden und enthalten auch die unbegrenzte Stückzahl aller Ihrer Geräte die mcFORTH nutzen.

Für alle Lizenzpakete gilt, dass die Sourcen für mcFORTH und Targetcompiler nur innerhalb der Firma genutzt und nicht weitergegeben oder für kommerziell erwerbbare FORTH-Versionen oder -Compiler verwendet werden dürfen. Bei Verletzung dieser Vereinbarung können entsprechende Verluste meinerseits geltend gemacht werden.

2. Der virtuelle FORTH-Processor VP32

Einer der wichtigsten Fragen bei der Erstellung des mcFORTH war, wie man am besten die neuen Konzepte für ein 32Bit-FORTH testen kann. Vor allem wenn man (wegen Altlast KKFORTH) sein eigenes 16-Bit FORTH am besten kennt. Das Ergebnis war ein Simulator, der dann später mit MASM32 in Assembler nachgebildet wurde. Das dabei entstandene erste mcFORTH war dann auch die Basis der Targetcompiler für weitere mcFORTH-Versionen. Also war die Henne das KKFORTH und das inzwischen geschlüpfte Ei das neue mcFORTH.

Für einen Simulator braucht man auch einen überschaubaren Befehlssatz und evtl. eine Möglichkeit zum Aufruf von Betriebssystem-Befehle (z.B. für das Fileinterface). Auch entspricht es eher einem realen Prozessor, dass man im Gegensatz zu den FORTH-Prozessoren den Datenstack und Returnstach im RAM des Prozessors hält.



Herausgekommen ist ein Simulator mit folgenden Features:

- Zwei Speicherbereiche: Flash für Programm und RAM für Variablen und Stacks
- Eine Execution Einheit mit ALU (= Arithmetik Logic Unit) und folgenden Register:
  PC = Programmzeiger (zeigt immer auf nächsten (FORTH-)Befehl
  RP = Returnstack-Pointer (für Rücksprungadressen)
  SP = Datenstack-Pointer (für Parameter)
  IP = FORTH-Programmzeiger für (indirekt) gefädelten Code
  W = Arbeitsregister für den NEST-Befehl – wird bei VP32 auch als Carry-Flag genutzt

Standard für Simulation eines Microcontrollers ist getrennte Flash- und RAM-Version und wird deshalb auch beim PC genutzt. Jedoch könnte man auch eine reine RAM-Version realisieren, welche für OS wie Windows und Linux besser geeignet sind. Eines der Hauptprobleme für einen Controller mit Flash ist, das der Flash-Speicher nur in bestimmten Blockgrößen gelöscht und geschrieben werden können, was hier ebenfalls getestet werden kann.

Auch vom Simulator ist es abhängig, ob eine Neumann-Architektur (Befehle für Flash und RAM sind identisch) oder ob eine Harvard-Architektur (z.B. beim x51 oder DSP's) mit identischen Adressen unabhängiger Speicher für Programm und Daten simuliert wird. Deshalb war es auch notwendig, getrennte Befehle für Zugriff auf Flash, RAM, I/O und externen Speicher zu definieren.

Das letzte Thema ist dann noch das Alignment, weil bei einigen Prozessoren der Zugriff auf 32-Bit auch nur 32-Bit-Aligned (entsprechend bei 16-Bit) erfolgen kann. Dies muss bei Anlegen von Variablen aber auch Pointer im Programm berücksichtigt werden.

2.1. Befehle des virtuellen Prozessors

Mit Ausnahme der Systemaufrufe werden die Befehle des virtuellen Prozessors in einem Byte codiert. Sprungadressen sind immer relativ zur Adresse des nächsten Befehls angegeben (rrrr) und sind wie Literals (llll) immer 32-Bit (16-Bit bei VP16). Nicht belegte Flash-Bereiche sind mit $FF vordefiniert. Alle nicht definierten Opcodes führt zum Abbruch der Befehlsausführung des virtuellen Prozessors.

2.1.1. FORTH-Kernbefehle

Programmausführung und Spünge

OpCode	Assembler	FORTH	Beschreibung
$00 rrrr	(call rel		\ CALL relativ
$01 rrrr	(branch ±rel		\ BRANCH
$02 rrrr	(0branch ±rel		\ 0BRANCH
$03 rrrr	(nbranch ±rel		\ NBRANCH (für NEXT und LOOP)
$04 llll	(lit abs		\ Literal
$05	(exit	EXIT	\ EXIT
$06	(execute	EXECUTE	\ Befehlsadresse auf Stack ausführen (HW-Spezifisch)

Stacks

OpCode	Assembler	FORTH	Beschreibung
$07	(sp@	SP@	\ liefert SP (als Adresse)
$08	(sp!	SP!	\ setzt SP (als Adresse)
$09	(drop	DROP	\ DROP
$0A	(dup	DUP	\ DUP
$0B	(over	OVER	\ OVER
$0C	(swap	SWAP	\ SWAP
$0D	(rp@	RP@	\ liefert RP
$0E	(rp!	RP!	\ setzt RP
$0F	(>r	>R	\ >R !!! Achtung bei unterschiedliche Länge !!!
$10	(r>	R>	\ R> !!! Achtung bei unterschiedliche Länge !!!
$11	(r@	R@	\ R@ !!! Achtung bei unterschiedliche Länge !!!
$12	(rdrop	RDROP	\ R@ !!! Achtung bei unterschiedliche Länge !!!

Logik, Shift und Arithmetik

OpCode	Assembler	FORTH	Beschreibung
$13	(and	AND	\ AND
$14	(or	OR	\ OR
$15	(xor	XOR	\ XOR
$16	(+c		\ ADD liefert Carry in Bit 0 von W
$17	(-c		\ SUB liefert Carry in Bit 0 von W
$18	(lshift	LSHIFT	\ Shift: NOS << TOS (Letzter Übertrag in W)
$19	(rshift	RSHIFT	\ Shift: NOS >> TOS (Letzter Übertrag in W)
$1A	(ashift	ASHIFT	\ Shift: NOS >> TOS (wie (rshift, MSB bleibt)
$1B	(um*	UM*	\ vorzeichenlose Multiplikation
$1C	(um/mod	UM/MOD	\ vorzeichenlose Division
$1D	(0<	0<	\ Vorzeichen prüfen
$1E	(u<	U<	\ vorzeichenloser Vergleich
$1F	(<	<	\ Vergleich

Zugriff auf Variablen

OpCode	Assembler	FORTH	Beschreibung
$20	(c@	C@	\ Zeichen aus Datenbereich lesen (8-Bit)
$21	(c!	C!	\ Zeichen in Datenbereich schreiben (8-Bit)
$22	(w@	w@	\ Halbwort aus Datenbereich lesen (16 Bit)
$23	(w!	w!	\ Halbwort in Datenbereich schreiben (16-Bit)
$24	(@	@	\ Wort aus Datenbereich lesen (32-Bit)
$25	(!	!	\ Wort in Datenbereich schreiben (32-Bit)

Zugriff auf Programmbereich (mit Konstanten)

OpCode	Assembler	FORTH	Beschreibung
$26	(c@p	C@P	\ C@ im Programmbereich (8-Bit)
$27	(c!p	C!P	\ C! im Programmbereich (8-Bit)
$28	(w@p	w@p	\ w@ im Programmbereich (16 Bit)
$29	(w!p	w!p	\ w! im Programmbereich (16 Bit)
$2A	(@p	@P	\ @ im Programmbereich (32-Bit)
$2B	(!p	!P	\ ! im Programmbereich (32-Bit)

Zugriff auf Portbereich

OpCode	Assembler	FORTH	Beschreibung
$2C	(c@io	C@IO	\ C@ im Portbereich (8-Bit)
$2D	(c!io	C!IO	\ C! im Portbereich (8-Bit)
$2E	(w@io	W@IO	\ W@ im Portbereich (16 Bit)
$2F	(!wio	W!IO	\ W! im Portbereich (16 Bit)
$30	(@io	@IO	\ @ im Portbereich
$31	(!io	!IO	\ ! im Portbereich (32-Bit)

Zugriff auf externen Speicher

OpCode	Assembler	FORTH	Beschreibung
$32	(c@x	c@x	\ Lesen eines Bytes aus dem externen Speicher
$33	(c!x	c!x	\ Schreiben eines Bytes in den externen Speicher
$34	(w@x	w@x	\ Lesen eines Halbwortes aus dem externen Speicher
$35	(w!x	w!x	\ Schreiben eines Halbwortes in den externen Speicher
$36	(@x	@x	\ Lesen eines Wortes aus dem externen Speicher
$37	(!x	!x	\ Schreiben eines Wortes in den externen Speicher

Erweiterung für indirekt gefädelten Code – verwendet IP als eigentlicher FORTH-Befehlszeiger

OpCode	Assembler	FORTH	Beschreibung
$38	(ip@		\ IP auslesen (für indirekt gefädelten Code)
$39	(ip!		\ IP setzen (für indirekt gefädelten Code)
$3A	(wp@		\ W auslesen (für indirekt gefädelten Code)
$3B	(wp!		\ W setzen (für indirekt gefädelten Code)
$3C	(nest		\ Für indirekt gefädelten Code: (IP++)=>W; (W++)=>PC
$3D	(code		\ folgenden Befehldurch realen Prozessor ausführen
$3E	(syscall		\ Betriebssystem-Aufruf (Command-Nummer auf Stack)
$3F	(nop		\ No Operation

Debugger bzw. ungenutzter Speicher

OpCode	Assembler	FORTH	Beschreibung
$40-$FF	???	???	\ Führt zum Abbruch

2.1.2. Betriebssystem

Betriebsysteme verwenden meist einen gemeinsamen Vektor für den Aufruf. Deshalb wird es auch hier im VP32 so gehändelt, dass der Befehl $3E = SYSCALL die Nummer der Funktion und die weiteren Parameter auf dem Stack erwartet. Zurückgeliefert werden immer ein Fehlerflag (OK = 0) und dann evtl. weitere Daten.

Minimale OS-Unterstützung

Command	Assembler	FORTH	Beschreibung	
$0000	(bye	BYE	\ Programm beenden	(–)
$0001	(debug	DEBUG	\ Setzt Debug-Funktion	(addr flag – 0)
$0002	(getpar	PAR@	\ Befehlszeile holen	(– addr 0)
$0003	(irqoff	IRQOFF	\ Interrupt aus	
$0004	(irqon	IRQON	\ Interrupt ein	
$0005	(irqset	IRQSET	\ Interruptvektor ändern/abfragen	
$0006	(v	V	\ Editor mit Fehlerposition aufrufen	

Zugriff auf externen Speicher

Command	Assembler	FORTH	Beschreibung
$0100	(xinit	xinit	\ Externer Speicher initialisieren
$0101	(xalloc	xalloc	\ Reserviert Speicher
$0102	(xrelloc	xrelloc	\ Verändert Speichergröße
$0103	(xfree	xfree	\ Gibt Speicher frei
$0104	(>x	>x	\ Ermittelt Pointer aus RAM-Adresse
$0105	(p>x	p>x	\ Ermittelt Pointer aus ROM-Adresse
$0106	(s>x	s>x	\ Ermittelt Pointer aus Stack-Adresse (oder 0)
$0107	(r>x	r>x	\ Ermittelt Pointer aus Returnstack-Adresse (oder 0)
$0108	(flash-se	flash-se	\ Liefert Sektoradresse und Größe beim Löschen
$0109	(flash-sw	flash-sw	\ Liefert Sektoradresse und Größe beim Schreiben
$010A	(flash-e	flash-e	\ Löschen eines Flash-Bereiches
$010B	(flash-w	flash-w	\ Schreiben eines Flash-Bereiches

Timer und Zeit

Command	Assembler	FORTH	Beschreibung
$0200	(cinit	cinit	\ Timer/Counter/RTC initialisieren
$0201	(systick	SYSTICK@	\ Liefert Systemticker und Frequenz in Hz (tick freq)
$0202	(time@	TIME@	\ Liefert aktuelle Zeit (ms s m h)
$0203	(time!	TIME!	\ Setzt aktuelle Zeit (ms s m h) - nicht im Simulator
$0204	(date@	DATE@	\ Liefert aktuelles Datum (wd t m y)
$0205	(date!	DATE!	\ Setzt aktuelles Datum (wd t m y) - nicht im Simulator
$0206	(waitms	WAITMS	\ wartet n Millisekunden

Tastatur abfragen und Zeichen ausgeben

Command	Assembler	FORTH	Beschreibung
$0300	(tinit	TINIT	\ Initialisieren des Terminals
$0301	(key	KEY	\ Auf Taste warten
$0302	(emit	EMIT	\ Zeichen ausgeben
$0303	(key?	KEY?	\ Testen, ob Taste gedrückt (möglichst nicht nutzen)
$0304	(emit?	EMIT?	\ Testen, ob Ausgabe möglich (möglichst nicht nutzen)
$0305	(emit?	EMIT?	\ Testen, ob Ausgabe möglich (möglichst nicht nutzen)
$0306	(page	PAGE	\ Bildschirm löschen
$0307	(atxymax	ATXYMAX	\ Bildschirmgröße abfragen
$0308	(atxy?	ATXY?	\ Bildschirmposition setzen
$0309	(atxy	ATXY	\ Bildschirmposition abfragen
$030A	(color@	COLOR@	\ Farbe abfragen (bzw. Attribut)
$030B	(color!	COLOR!	\ Farbe setzen (bzw. Attribut)
$030C	(at@	AT@	\ Zeichen und Farbe abfragen (aktuelle Position)
$030D	(at!	AT!	\ Zeichen und Farbe setzen (aktuelle Position)

File öffnen und schließen

Command	Assembler	FORTH	Beschreibung	
$0400	(finit	FINIT	\ Initialisieren das Fileinterface	
$0401	(fcreate	FCREATE	\ File anlegen	(csa/0 -- id 0 | error)
$0402	(frename	FRENAME	\ Fileumbenennen	(csa1/0 csa2/0 -- error)
$0403	(fdelete	FDELETE	\ File löschen	(csa/0 -- id 0 | error)
$0404	(fopen	FOPEN	\ File öffnen	(csa/0 flag -- id 0 | error)
$0405	(fclose	FCLOSE	\ File schließen	(id -- error)
$0406	(fsize@	FSIZE@	\ Filegröße abfragen	(id -- d 0 | error)
$0407	(fsize!	FSIZE!	\ Filegröße setzen	(d id -- error)
$0408	(fpos@	FPOS@	\ Fileposition abfragen	(id -- d 0 | error)
$0409	(fpos!	FPOS!	\ Fileposition setzen	(d dir id -- 0 | error)
$040A	(fread	FREAD	\ File auslesen	(xptr len id --len2 0 | error)
$040B	(fwrite	FWRITE	\ File schreiben	(xptr len id --0 | error)
$040C	(ffirst	FFIRST	\ File suchen	(csa/0 attr -- dta 0 | error)
$040D	(fnext	FNEXT	\ Nächstes File	(-- dta 0 | error)
$040E	(fdir@	FDIR@	\ Aktuelles Verzeichnis abfragen	(– csa0 0 | error)
$040F	(fdir!	FDIR!	\ Verzeichnis setzen	(csa/0 – error)

$0410	**(fdrv@**	**FDRV@**	**\ Aktuelles Laufwerk abfragen (– csa0 0 | error)**	
$0411	**(fsetdrv**	**FDRV!**	**\ Laufwerk setzen**	**(csa/0 – error)**
$0412	**(ffree**	**FFREE**	**\ Freie Größe ermitteln**	**(dev/0 -- free. max. 0 | error)**
$0413	**(fpar@**	**FPAR@**	**\ Parameter abfragen**	**(... handle – error)**
$0414	**(fpar!**	**FPAR!**	**\ Parameter setzen**	**(... handle – error)**

2.2. mcFSimVP32.exe – VP32-Simulator in Assembler

Damit eine vernünftige Geschwindigkeit auf Windows-Maschinen erreicht wird, wurde ein VP32-Simulator in MASM32 realisiert, dessen Sourcecode ebenfalls im Umfang des mcFORTH ausgeliefert wird.

Dieser Simulator realisiert alle Befehle des virtuellen Prozessors VP32 und auch die notwendigen Schnittstellenbefehle. Für den Test wurden aber Schreibzugriffe auf Flash blockiert, damit zum Ändern des Flash die speziellen Flash-Befehle flash-w und flash-e genutzt werden müssen. Aktuell ist der Simulator so eingestellt, dass er zum Löschen immer 256 Bytes und zum Schreiben je 16 Bytes erwartet (entspricht der Vorgabe des Infineon XMC1xxx). Dies Parameter werden zwar mit flash-se und flash-sw zurückgeliefert, aber bei flash-e und flash-e nicht geprüft.

Bei Aufruf ohne Parameter (bzw. nur mit dem Filename) wird folgende Konfiguration verwendet:

- Ein Flash mit 2MByte ab Adresse 0 wird eingerichtet und mit File-Image gefüllt (Rest $FF)
- Ein RAM mit 1MByte ab Adresse $2000.0000 wird mit auch mit $FF gefüllt
- Es wird eine Harward-Architektur angenommen (+h - @ für RAM und @p für Flash)
- Programme dürfen nicht im RAM sein (wegen Harward)
- Es ist kein Alignmend notwendig (-A)

Aufruf:

```
 * mcFSimVP32 Filename                          \ Default-Konfiguration
 * mcFSimVP32 \Filename                         \ Startet mit Debugger in Single-Step
 * mcFSimVP32 /Filename                         \ Startet mit Debugger in Trance-Mode
 * mcFSimVP32 -parameter1 -parameter2 … Filename
     +pxxxxxxxx.yyyyyyyy                        \ Anfang/Länge des Flash-Bereiches
     -p                                         \ kein Flash vorhanden
     +vxxxxxxxx.yyyyyyyy                        \ Anfang/Länge des RAM-Bereiches
     +h                                         \ Harward-Architektur (Default)
     +a                                         \ Align erforderlich
     -h                                         \ Neumann-Architektur
     -a                                         \ kein Align erforderlich (Default)
```

Bei Harward-Architektur ist es auch möglich, das Flash und RAM an gleichen Adressen zu führen. Bei Neumann sollte auf nicht überlappende Speicherbereiche geachtet werden. Bitte beachten, dass im aktuellen mcFVP32d.mcf die Default-Konfiguration erwartet wird. Jedoch kann mit einer kleinen Umstellung von IRF (Setzen des Bits 2) und Verwendung des Parameter -h beim Start des Simulator ein Programme auch ins RAM compiliert und ausgeführt werden:

Start des mcFORTH:

```
    mcFSimVP32.exe -h mcfvp32d.mcf      \ bzw. mcf32d.mcf (mit Editor)
```

Eingaben:

```
irf @  4 or  irf !          \ Unified-Flag setzen
>ram                        \ Compiler auf RAM wechseln
Variable test               \ eine Variable im RAM
>rom                        \ Compiler wieder im Flash
Variable test2              \ eine Variable im ROM
words                       \ Wortliste
```

3. Das mcFORTH

Das mcFORTH besteht aus einem maschinennahen Set von Befehlen, der für die Ausführung auf einem realen (oder für Test virtuellen) Prozessor übernimmt. Die restlichen, hier aufgeführten Befehle dienen dem Compiler zur Verwaltung des Dictionary und Erstellung neuer Befehle mit entsprechenden Programmstrukturen. Mit diesem Befehlssatz (ca. 20K auf dem VP32D) sind die Grundfunktionen des mcFORTH einschließlich Compilierung, Test und Speicherung möglich. Mit Editor hat das mcf32d.mcf dann ca. 32KByte.

3.1. Verwendung der File-Extensions

Es werden neben der Endung „.f“ für sequenzielle FORTH-Programme auch die Endung ".scr" für FORTH-Screens mit dem üblichen 1K-Aufbau von 16 Zeilen a 64 Zeichen ohne Zeilenumbruch verwendet.

Darüber hinaus werden folgende Endungen verwendet:

.mcf	Virtueller 32Bit-FORTH-Prozessor – für den 32bit-Simulator (Endung wird aber auch für die Images anderer mcFORTH verwendet)
.mcfa	VP32-FORTH mit Alignment

Bei den Files sind oft folgende Ergänzungen im Verwendung

_def	Konstantendefinitionen (z.B. vp32_def.scr bei Liste der Opcodes für VP32)
_asm	Assembler (z.B. m0_asm.scr für den Cortex-M0-Assembler)
_dis	Disassembler
_sim	Simulator

Es gibt natürlich unterschiedliche Realisierungen beim mcFORTH – und dafür verwendete Zeichen:

- D = Direkt gefädeltes FORTH (STC) erwartet schon Befehlscode in der Codefeld-Adresse
- I = Indirekt gefädeltes FORTH (ITC) speicher die Adresse der Assemblerroutine in der CFA

Die Images für die entsprechenden mcFORTH-Versionen heißen dann:

– mcvp32d.mcf	Das 32-Bit mcFORTH für VP32-FORTH (direkt gefädelt)
– mcvp32i.mcfa	Das indirekt gefädelte und aligned VP32-FORTH
– mcf32d.mcf	Zur Bequemlichkeit: ein mcFORTH mit Screen-Editor
– mcfxmc2go.mcf	Das Image für das XMC2GO (ab Adresse $10000100 (Besser: mcfxmc2go.hex ist das entsprechende Intel-Hex-File)

3.2. Kurzübersicht der mcFORTH-Befehlsliste

Hier eine kurze Zusammenfassung der mcFORTH-Befehle in Gruppen. Die meisten FORTHler werden viele der Befehle kennen oder erahnen, welche Funktion sie haben. Eine ausführliche Beschreibung ist im Anhang (alphabetisch sortiert) zu finden.

\ Konstanten

```
  TRUE   FALSE   BL                          ( Konstanten für Vergleiche, Leerzeichen )
  #MSB   #BITS                               ( Wortgröße )
  #TB   #C/TB   #FB   #C/FB                  ( Größe von Terminal- und Filepuffer )
  #PRGEND   #VAREND                          ( Endadressen von Programm und RAM )
```

\ Speicherzugriffe

```
  XFREE   XALLOC                             ( Externer Speicher reservieren )
```

```
  X+  >X  P>X                                  ( Umrechnung FORTH-Speicher zu extern )
  !X  C!X  @X  C@X                             ( Zugriff auf externen Speicher )
  COUNT  2!  2@  !  C!  @  C@  ALIGNED         ( Zugriff auf Variablenbereich )
  COUNTP  !P  C!P  @P  C@P  ALIGNEDP           ( Zugriff auf Programmbereich )
  CMOVEP  CMOVE                                ( Speicher verschieben )
  CELLS  CELL+  CHARS  CHAR+                   ( nächste Adresse / Größe ermitteln )
```

\ Stackmanipulation

```
  RDEPTH  RINIT                                ( Returnstackgröße und Initialisierung )
  RDROP  R@  R>  >R                            ( Manipulation )
  2R>  2R@  2>R                                ( Doppeltgenaue Werte )
  SDEPTH  SINIT                                ( Stackgröße und Initialisierung )
  DROP  SWAP  OVER  DUP                        ( Manipulation )
  -ROT  ROT  NIP  TUCK  ?DUP                   ( weitere Manipulation )
  2DROP  2OVER  2SWAP  2DUP                    ( Doppeltgenaue Werte )
```

\ Arithmetik, Logik und Vergleiche

```
  1-  1+  -  +  D-  D+                         ( Addition und Subtraktion )
  2/  U2/  2*  D2/  DU2/  D2*                  ( Shift )
  XOR  OR  AND                                 ( Logik )
  UMIN  ABS  DUMIN  DABS  UPC                  ( Auswahl )
  WITHIN  D<  DU<  D0<  D=  D0=  <  U<  0<  =  0=      ( Vergleich )
  /  M/MOD    *  M*  UM*                        ( Multiplikation und Division )
```

\ Schnittstellen

```
  IOVEC  IOINIT  STDIO                         ( IO-Vektor und Default )
  KEY?  KEY                                    ( Eingabe-Grundroutine )
  ACCEPT                                       ( Eingabe einer Zeile )
  EMIT?  EMIT                                  ( Ausgabe-Grundroutine )
  TYPE  TYPEP  CR  SPACES  SPACE               ( Ausgabe von Zeichen und Strings )
  AT-MAX  AT-XY  AT-XY?  PAGE                  ( Bildschirm-Manipulation )
  DECIMAL  HEX  BASE                           ( Zahlenbasis umschalten )
  #>  SIGN  #S  #  HOLD  <#  HLD               ( Zahl in String umwandeln )
  .  .R  U.  U.R  0U.R  D.  D.R                ( Zahl ausgeben )
  $>NUMBER?  >NUMBER  DPL                      ( String in Zahl umwandeln )
  STDFILE  FILEVEC  FINIT                      ( File-Vektor und Default )
  BIN  R/W  W/O  R/O                           ( Filetyp )
  FWRITE  FREAD  FSETPOS                       ( File lesen und schreiben )
  FGETSIZE  FCLOSE  FOPEN  FCREATE             ( File anlegen, öffnen und schließen )
  FCB  FID                                     ( Variablen für Filehandler und Filename )
  LOADFROM  THRU  LOAD  CAPACITY               ( Laden von FORTH-Screens )
  INCLUDE  OPEN  CLOSE                         ( Laden von sequenzellen Files )
```

\ Struktur

```
  J'  J  I'  I  ?LEAVE  LEAVE  UNLOOP  +LOOP  LOOP  DO  ?DO
  NEXT  ?FOR  FOR                              ( Einfaches Zählen n-1 … 0 )
  BEGIN  WHILE  UNTIL  REPEAT  AGAIN           ( Schleifen )
  IF  ELSE  THEN                               ( Bedingungen )
  AHEAD  RECURSE                               ( Vorwärts-Sprung )
```

\ Vokabulare

```
  VOC  CONTEXT  CURRENT                        ( Speicher für Vokabularverwaltung )
  ORDER  VOCS                                  ( Anzeige der Vokabulare und Reihenfolge )
  PREVIOUS  ALSO                               ( Suchreihenfolge erweitern, verkürzen )
  ONLYFORTH  FORTH                             ( Standard oder FORTH-Vokabular )
  VOCABULARY  DEFINITIONS                      ( neues Vokabular anlegen )
```

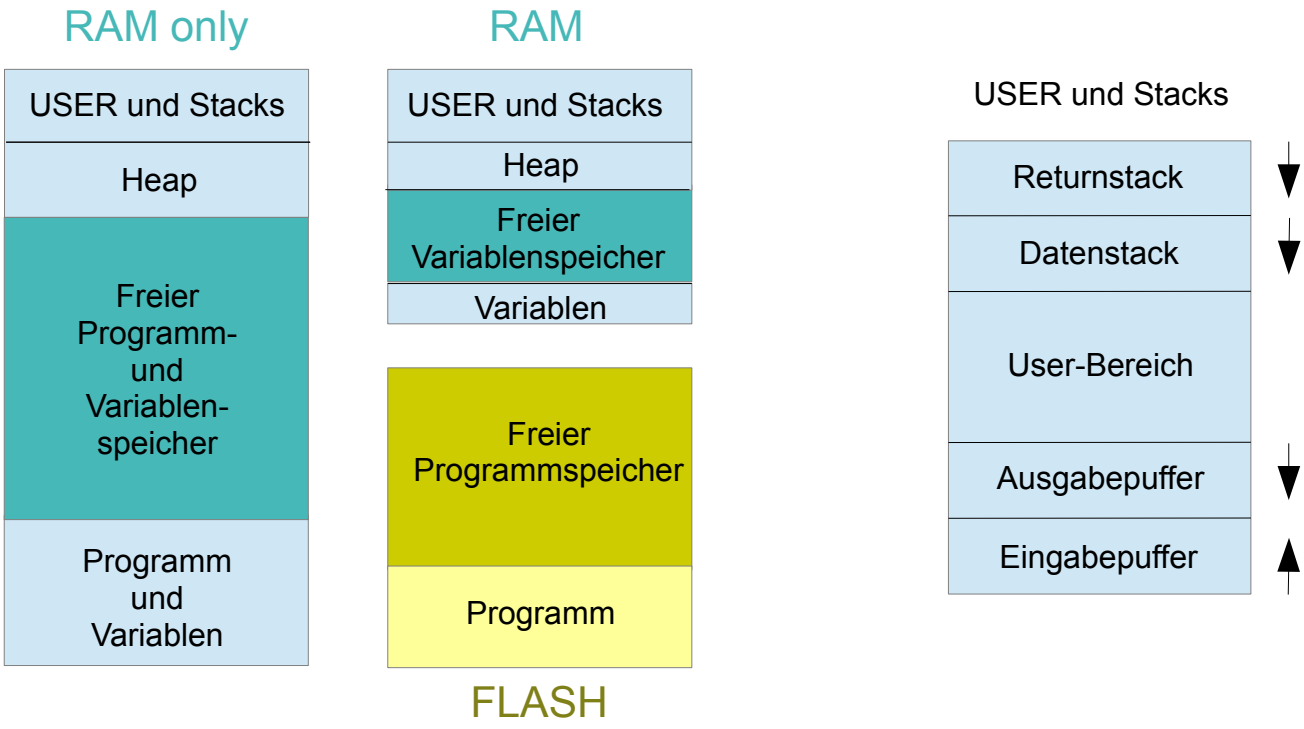
\ FORTH-Interpreter

```
  BYE  'BOOT  QUIT                                ( Programm beenden oder neu starten)
  SAVE                                            ( Programm speichern )
  ?EXIT  EXIT  EXECUTE  NOP                       ( Befehl ausführen oder verlassen / NOP )
  EVALUATE  INTERPRET                             ( String oder Befehl interpretieren )
  REFILL  SOURCE  >IN  SOURCE-ID                  ( Verwalten des Source-Zeigers )
  SFIND  (SFIND  -PARSE  PARSE                    ( Befehl extrahieren und suchen )
  POSTPONE  [']  '                                ( Befehle suchen und einbinden )
  NAME>  >NAME  BODY>  >BODY                      ( In den Befehlen bewegen )
  >ROM  >RAM  >RAM?  ROM?  UNIFIED?               ( Manipulation des Systems )
  ]  [  STATE  REVEAL                             ( Interpreter-Status )
  CREATEP  CREATE  DOES>                          ( Datentypen definieren )
  PRG  UNUSEDP  HEREP                             ( Programm-Pointer )
  $,P  ,P  C,P  ALIGNP  ALLOTP                    ( Programmbereich manipulieren )
  2CONSTANT  CONSTANT  ALIAS  IS  DEFER           ( Konstanten, Alias und Defers )
  2LITERAL  LITERAL                               ( Inline-Literals )
  ;  :  :NONAME  IMMEDIATE  RESTRICT             ( Highlevel-Definition )
  RAM  UNUSED  HERE                               ( RAM-Pointer )
  $,  ,  C,  ALIGN  ALLOT                         ( Programmbereich manipulieren )
  HEAP  UNUSEDH  ALLOTH  CLEARH                   ( HEAP-Pointer )
  >HEAP  HEAP>                                    ( Variablenbereich und HEAP-Nutzung )
  $,  ,  C,  ALIGN  ALLOT                         ( Variablenbereich manipulieren )
  2VARIABLE  VARIABLE                             ( Variablen-Definition )
  ."  P"  S"                                      ( String-Händling )
  \\  \  (                                        ( Kommentare )
  [IFNDEF]  [IFDEF]  [IF]  [ELSE]  [THEN]         ( Bedingte Ausführung )
  CHAR                                            ( Ein Zeichen ermitteln )
  'ABORT  ABORT  ABORT"  ?ABORT  CATCH  THROW      ( Fehlerbehandlung )
  .S  DUMP  DUMPP  DUMPX  WORDS                   ( Anzeige von Stack, Speicher, Befehle)
```

3.2.1. RAM oder Flash/RAM ?

Das mcFORTH soll auch auf Microcontroller verwendbar ein, die ihre Programme im Flash ablegen. Um kostbaren RAM-Speicherplatz nicht zusätzlich durch eine Kopie des Programmes zu belegen, ist die Ausführung des mcFORTH im Flash möglich. Zusätzlich ist zu berücksichtigen, dass bei der Compilierung des Programmes im Zielsystem das Flash nur einmal - meist mit gesonderten Befehlen - beschrieben werden kann. Ob ROM bzw. Flash vorhanden ist, kann mit `ROM?` abgefragt werden. Es ist auch möglich, die Compilierung zwischen RAM (Befehl `>RAM`) und Flash (Befehl `>ROM`) umzuschalten, wenn es sich um ein Neumann-System (siehe oben) handelt.

Deshalb hier die beiden Möglichkeiten:



Bei einer reinen RAM-Version sind die Variablen mitten im Programm und es gibt keinen eigenen Variablenbereich. Am Speicherende (typischerweise - muß aber nicht immer so sein) gibt es dann noch den USER-Bereich mit den Stacks und Arbeitsbereiche für Terminal und Interpreter.

Bei einer Flash/RAM-Version ist das Programm meist im Flash und die Variablen im RAM-Bereich. Beim Speichern des Programmes wird eine Kopie des Variablen-Bereiches mit ins Flash übernommen und beim Start von mcFORTH wieder ins RAM kopiert.

Der User-Bereich ist aktuell nur für Arbeitsbereich zur Ablage von Programmteilen, die für das Flash gedacht sind (zwischen BEGIN-CREATE und END-CREATE.

3.2.2. HEAP

Der Heap ist ein Arbeitsbereich am Ende vom RAM unterhalb des USER-Bereiches. Er kann genutzt werden, um für Programme ein geschützten Bereich zu reservieren. Eine zweite Möglichkeit ist die Ablage von nur temporär verfügbaren Header oder ganzen Programme. Dabei muß berücksichtigt werden, das Wörter im „normalen“ Programmbereich diese Heap-Befehle nicht compilieren dürfen, weil nach einem Neustart des mcFORTH oder Löschen des Heaps diese evtl. nicht mehr vorhanden sind.

Eine Besonderheit ist auch die Verwendung von headerlosen Befehle, deren Namen im Heap gespeichert sind. Hierzu wird neben der üblichen Verkettung im Programm eine zweite Kette im Heap verwendet, welche die Reihenfolge der Definition durch die Adresse des Programms erkennt.

Da das mcFORTH einige Daten (z.B. Filenamen für FOPEN) im RAM erwartet, müssen diese vom Flash ins RAM kopiert werden. Mit dem Befehl $p>heap (addr len – addr2 len) wird ein String aus dem Flash in den Speicher unterhalb des Heaps kopiert und dabei die Längenangabe unterhalb

addr2 und eine 0 ans Ende gestellt, ohne das der Heap-Zeiger verändert wird. Dies wird implizit auch bei S“ String“ genutzt und erlaubt die interaktive Nutzung des Strings.

3.2.3. Indirekt gefädeltes FORTH ?

Echte FORTH-Prozessoren führen die wichtigsten Befehle in einem Takt aus. Deshalb fehlt oft in diesen Systemen das sogenannte Codefeld (siehe Einführung FORTH und Befehlsaufbau) und die meisten Befehle werden Inline (sprich: innerhalb eines Wortes) codiert. Komplexere Befehle wie `WORDS` sind dann mit Unterprogramm-Aufrufe eingebunden. Bei den meisten anderen Prozessoren hat sich aber herausgestellt, dass die Ausführung schneller ist, wenn neben des üblichen Programmzeigers PC einen FORTH-Zeiger IP existiert und eine Routine mit Namen NEST (ist kein FORTH-Befehl) den nächsten Befehl aufruft. Vorteil ist (zumindest bei 16-Bit-Systeme), dass nur 16 Bit pro Befehl (Ausnahme Literals und Sprünge) benötigt wird.

3.3. Änderungen wegen der ROM-Fähigkeit

Eines der Hauptprobleme von Flash ist, dass es nach dem Löschen nur einmal – oft nur in größeren Blöcken (z.B. wegen ECC oder Prozessor-Caches) - programmierbar ist. Deshalb musste ein Mechanismus gefunden werden, der auf der einen Seite möglichst wenig an der üblichen Art der Befehlserzeugung ändert und trotzdem dem System eindeutig sagt, wenn ein Befehl abgeschlossen wird. Dies geschieht bei mcFORTH auf folgenden Weg:

- Änderungen im Flash müssen mit Begin-Create und End-Create eingeschlossen sein
- Alle Befehle werden ohne sofort sichtbaren Header erstellt (also auch bei Create)
- End-Create schließt alle Schreiboperationen ab und macht den neuen Befehl sichtbar
- Die Befehle CONSTANT VARIABLE ; DOES> … verwendet Start-/End-Create (damit sind diese Änderungen im normalen FORTH-Code nicht auffällig)
- IMMEDIATE und RESTRICT sind nicht mehr verfügbar und werden durch (I und (R ersetzt. Diese müssen vor der Definition eines Befehls – also auch vor Create - angegeben werden. Alternativ ist bei Highlevel-Befehle die Verwendung von `I:` , `R:` oder `IR:` möglich.

Es gibt Prozessoren, die relativ große Schreib-Sektoren (16 Bytes bei XMC1100) haben. Da bei `begin-create` immer der Programmzeiger auf den nächsten Sektor gesetzt wird. Damit man den Programmcode auch kompakter halten kann, ist `begin-create` und `end-create` ineinander schachtelbar und schreibt das Flash erst beim letzten `end-create` zurück. Damit können Programmteile mit maximaler Größe des Flash-Puffers (1K) realisiert werden. Jedoch ist zu beachten, dass die noch im RAM befindlichen Programmteile nicht aufgerufen werden dürfen und deshalb diese Befehle noch nicht sichtbar sind.

3.4. Vektorisierung

Wenn man mit einem FORTH eine Applikation entwickelt, wird mit dem Grundsystem gestartet, das Programm geladen und dann getestet. Dafür ist es gut, dass sich das System interaktiv am Terminal verhält, das Filesystem des Terminals nutzt und sich bei Fehler wieder zurückmeldet.

Wenn aber das Programm abgeschlossen ist, kann ein vollständig anderes Verhalten erforderlich sein. Dann sollte sofort die Anwendung starten, die Ausgabe z.B. auf einem angebundenen LCD-Display erfolgen und nur auf ein paar Tasten reagiert werden. Bei Fehler sollte entweder das Programm selbst das Problem beheben oder das System eine Warnung z.B. über Funk schicken.

Daten können dabei z.B. statt auf dem Filesystem der Festplatte in einer wechselbaren SD-Karte abgelegt sein.

Dies ist aber nur möglich, wenn das Verhalten des FORTH gemäß den obigen Wünschen verändert bzw. erweitert werden kann. Im mcFORTH werden dazu sogenannte DEFER's und Vektoren verwendet, die an entscheidenden Schnittstellen des System mit einem Standard-Verhalten vorbelegt sind. Unter anderem ist möglich, nach der Initialisierung ein eigenes Programm aufzurufen, auf Fehler selbst zu reagieren und Terminal oder Filesystem mit eigenen Funktionen zu belegen.

3.4.1. Programmstart und -ende

Beim Programmstart sind folgender Defer für die Aktionen zuständig: 'BOOT
Vor schließen aller Files und Ende des FORTH wird auch ein Defer aufgerufen: 'BYE

Beim Start des mcFORTH innerhalb des Simulators wird die Befehlszeile in den FORTH-Eingabepuffer übernommen. Eigene Anwendungen können deshalb mit PARSE bzw. PARSE-WORD direkt darauf zugreifen. Ist 'BOOT nicht genutzt (' NOP IS 'BOOT) wird die Eingabezeile direkt vom mcFORTH interpretiert/compiliert.

3.4.2. Ausgaben während des Compilierens

Alle Ausgaben nutzen das Standard-Interface zur Benachrichtigung des Benutzers. Um die Ursache von Fehlern zu sehen, wird bei Textfiles (z.B. nach Include time.f) der Text ausgegeben. Bei Screen-Files nur ein „>“.

3.4.3. Fehlerbehandlung

Die Fehlerbehandlung entspricht den Standard einschließlich CATCH und THROW. Wurden diese nicht genutzt, erfolgt üblicherweise nach einem Fehler die Ausgabe des Verursachers, der Text zum Fehlerstring (mittels ERROR.) und Rücksprung zu QUIT (bei ABORT ohne Ausgaben).

3.4.4. Ein-/Ausgabe

Beim VP32 wurde die Ein-/Ausgabe direkt über das Windows realisiert. Deshalb kann die Ausgabe des Simulators auch umgeleitet werden. Leider funktioniert der Empfang der Befehle aus einem File noch nicht.

3.4.5. Filehandling

Der Simulator erlaubt den direkten Zugriff auf die Files unter Windows oder auch aus dem Microcontroller über das entsprechende Terminalprogramm. Dabei wird zwischen sequenziellen Files und Screenfiles unterschieden:

include Filename	Sequenzielles File laden (z.B. Include life.f)
1 loadfrom Filename	Screen 1 eines Files laden (z.B. 1 Loadfrom WinEditor.scr)

3.5. Bedingungen und Schleifen

Natürlich kommen FORTH-Programme auch nicht ohne Bedingungen und Schleifen aus. Deshalb ist das ganze Set von Befehlen dazu realisiert. Diese Befehle sind schon dem FORTH-Standard 2012 angepaßt und können auch abweichend von dem üblichen IF ELSE THEN bzw. BEGIN WHILE REPEAT fast beliebig geschachtelt werden.

Zusätzlich ist auch eine CASE-Struktur und neben DO … LOOP eine schnellere FOR … NEXT-Schleife realisiert. Dabei wird abweichend von der Verwendung durch Charles Moore die Schleife automatisch um 1 erniedrigt und deshalb bei 1 FOR die Schleife einmal und bei 0 FOR sogar 2^n (n=Bitanzahl) mal durchlaufen. Sollte bei 0 die Schleife nicht durchlaufen werden, so muss ?FOR verwendet werden.

Hier die Beschreibung, was bei den Strukturbefehlen während des Compilierens passiert. Die Funktion dazu wird weiter unten in der Befehlsliste erwähnt.

Befehl	**Compilierung**	**Ausführung**
BEGIN	(-- addr 1)	(--)
	Merkt sich die Adresse für einen Rücksprung	
AHEAD	(-- addr -2)	(--)
	Compiliert unbedingter Vorwärts-Sprung – wird z.B. bei ELSE verwendet	
IF	(-- addr 3)	(f --)
	Compiliert bedingten Vorwärts-Sprung	
ELSE	(addr 3-- addr2 2)	(--)
	Compiliert unbedingter Vorwärtssprung, und vervollständigt den bedingten Vorwärts-Sprung von IF	
THEN	(addr 2/3 --)	(--)
	Auflösung von AHEAD (ELSE) bzw. IF	
WHILE	(x y --addr 3 x y)	(f --)
	wie IF, aber danach 2SWAP bei Compilierung	
AGAIN	(addr 1 --)	(--)
	Compiliert bedingungsloser Rücksprung	
REPEAT	(addr 2/3 addr 1 --)	(--)
	Zuerst AGAIN, dann THEN	
FOR	(-- addr 4)	(n -- ; R: -- n-1)
	Compiliert 1- und dann >R	
?FOR	(-- addr -4)	(n -- ; R -- n)
	Compiliert >R und einen unbedingten Vorwärtsspung (wie bei AHEAD, jedoch Flag -4)	
N	(--)	(n -- ; R -- n)
	Liefert den Schleifenwert der äußeren FOR-Schleife.	
NN	(-- n)	(m -- ; R – m n)
	Liefert den Schleifenwert der nächst inneren FOR-Schleife	
NEXT	(addr ±4 --)	(R: n – n-1 |)
	Löst Vorwärtssprung von ?FOR auf und compiliert NBRANCH	
NDROP	(--)	(R: n --)
	Entfernt den obersten Schleifenwert (kann in einem Prozessorregister sein)	
>N	(n --)	(R: -- n)
	Verwendet n als neuen Schleifenwert (kann auf Returnstack oder Register sein)	
N>	(-- n)	(R: n --)
	Bringt den obersten Schleifenwert zum Stack.	
DO	(--addr1 addr2 5)	(n2 n1 -- ; R: -- addr end count)
	Compiliert (DO und unbedingten Vorwärtssprung, legt danach aktuelle Adresse und Flag 5 auf den Stack	
?DO	(-- addr1 addr2 5)	(n2 n1 --)
	Compiliert (?DO und unbedingten Vorwärtssprung, legt danach aktuelle Adresse und Flag 5 auf den Stack	
LOOP	(addr1 addr2 5 --)	(-- ; R: addr end count – addr end count-1 |)
	Compiliert NBRANCH , danach zweimal RDROP und löst Vorwärtssprung von DO auf (addr1)	
+LOOP	(addr1 addr2 5 --)	(n -- ; R: addr end count – addr end count-n |)
	Compiliert (+LOOP , danach bedingter Rücksprung zu addr2, ansonsten wie LOOP (zweimal RDROP ...)	
I	(--)	(-- n ; R: addr end count – addr end count)
	Liefert den Schleifenwert der äußeren Schleife (entspricht end–count)	
J	(--)	(-- n ; R: a e c x x x – a ec x x x)
	Liefert den Schleifenwert der inneren Schleife (entspricht e-c)	

CASE (-- 6 6) (--)
Anfang einer CASE-Struktur
OF (… 6 – … addr -6) (n c – n |)
Compiliert OVER = und bedingten Vorwärtssprung.
OFR (… 6 – … addr -6) (n min max – n |)
Compiliert (OFR und bedingten Vorwärtssprung.
ENDOF (addr -6 – addr2 2 6) (--)
Compiliert unbedingten Vorwärtssprung und löst unbedingten Vorwärtssprung von OF auf
ENDCASE (6 … 6 --) (n --)
Compiliert DROP und löst alle unbedingten Vorwärts-Sprünge von ENDOF auf

4. Sonstige Besonderheiten im mcFORTH

4.1. Aufbau der Befehle Createp ... Doesp> bzw. Create ... Does>

Die Create-Does-Struktur des FORTH ist die Basis für alle vom Anwender realisierten Datenstrukturen. Dabei werden im (Teil1) die Datenstrukturen aufgebaut und vom (Teil2) wieder verarbeitet. Beim mcFORTH wird zwischen RAM und ROM/Flash unterschieden und deshalb auch ein Createp mit Doesp> realisiert. Das untere Beispiel kann z.B. Daten auf dem Datenstack als Konstanten in Flash ablegen und bei Aufruf des Wortes wieder zurück liefern.

Bei der Definition von dem Befehl `#DiceMax` wird der Parameter (hier 6) gespeichert. Bei jeder Verwendung von `#DiceMax` wird der Wert wieder zurückgeliefert.

Besonderheit in mcFORTH ist, dass wegen Flash auch CreateP mit End-Create abgeschlossen wird:

```
CreateP mcf$  s"mcF" $,p  End-Create
```

Definition einer CreateP ... Does> - Struktur ist wie folgt:

```
: X: CreateP (Teil1 = ,p) Does> (Teil2 = @p) ;
6 X: #DiceMax
```

Ergibt bei direkt gefädelten Code: **(xxx)** ist ein Adresse, auf die xxx zeigt (gleiches mit yyy)

```
mcf$        lit xxx  exit (xxx)  3 'm' 'c' 'F' 0
:createp    Header  r>  dup ,p  cell+  >r ;
:X:         call (createp  yyy .(Teil1). End-Create  exit
            (yyy) r> (Teil2) ;
#DiceMax    call xxx  6
```

Bei indirekt gefädelten Code (wenn aligned):

```
(docreatep  fw@  nest
mcf$        (docreatep  3 'm' 'c' 'F' 0
: (createp  Header  r>  dup ,p  cell+  >r ;
(does>      r>  fip@  >r  fip!  nest
X:          (createp  xxx .(Teil1). End-Create  exit
            2 allotp  (xxx)  fw@  call (does>  (Teil2) ;
#DiceMax :  xxx   6
```

Dies könnte man natürlich auch mit einer Konstante und einigen `,p` 's realisieren, aber wenn man den Wert im RAM ablegt und bei jeder Abfrage erhöht, könnte man sich ein Zähler oder (etwas aufwendiger) einen Zufallszahlengenerator realisieren. Für RAM werden die Standardbefehle Create und Does> verwendet. Alle Daten im RAM werden bei `SAVE` auch gespeichert.

Create reserviert keinen zusätzlichen Speicher, sondern liefert nur die Variablenadresse:

```
Create filename  24 allot  End-Create
```

Der Aufbau der Create ... Does> - Struktur ist wie folgt:

```
: C: Create 0 , Does> dup >r  @ 1+  dup r> ! ;
6 C: c1
```

Ergibt bei direkt gefädelten Code: **(xxx)** ist ein Programmlabel

```
filename:   lit xxx  exit  - im RAM: (xxx)  24 Byte reserviert
: (createp  Header  r>  dup  cell+  >r  @p compile, ;
C:          call (createp  xxx .(Teil1). End-Create  exit
            (xxx) r> @p (Teil2) exit
c1 :        call xxx  yyy – im RAM: (yyy) 0
```

Bei indirekt gefädelten Code:

```
docreatep   fw@  nest
filename    (docreatep xxx - im RAM: (xxx)  24 Byte reserviert
: (dcreatep Header  r>  dup  cell+  >r  @p compile, ;
C:          (createp  xxx .(Teil1). End-Create exit
            (xxx) fw@ @p ... ;
c1          call xxx   6
```

Bitte beachten, dass einige Systeme nur aligned Zugriffe erlauben. Deshalb empfiehlt sich, in Datenstrukturen mit `walign` bzw. `align` (oder mit ..p für Flash) für Alignment zu sorgen. Create sorgt dafür, das der Programmspeicher am Anfang aligned ist.

4.2. InitRegs: und ;InitRegs

Oft müssen bei Microcontroller eine Reihe von Ports initialisiert werden. Da dies schnell zu größeren Programmen führt, wurde eine Funktion realisiert, die ein Großteil dieser Aufgabe aus einer Tabelle entnimmt und sehr schnell ist, weil sie direkt in Assembler realisiert wurde. Diese entsprechende Routine findet man zusammen mit den Port-Definition im Vokabular HW.

Die Verwendung ist wie folgt:

```
InitRegs: ledsinit
  #ir|   ,p  #port2dir ,p  $00000001 ,p
  #ir&|  ,p  #port2val ,p  $FFFFFFFC ,p   $00000001 ,p
 ;InitRegs
```

Der erste Wert gibt die Funktion an, der 2 Wert die Adresse, die weiteren 1-2 Parameter sind zusätzliche Werte für eine Funktion. ;InitRegs ist notwendig, um den Befehl abzuschließen und auch mit dem Wert 0 die Tabelle abzuschließen. Die aktuell definierten Funktionen sind:

0:	Ende der Tabelle (wird automatisch bei ;InitRegs eingefügt	
#ir!	Set:	Der Port wird auf Wert1 gesetzt
#ir&	AND:	Aktuelle Portwert AND Wert1 wird zurückgeschrieben
#ir|	OR:	Aktuelle Portwert OR Wert1 wird zurückgeschrieben.
#ir&|	ANDOR:	Aktueller Portwerd AND Wert1 OR Wert2 wird zurückgeschrieben
#ir&=	AND=:	Warten, bis aktueller Portwert AND Wert1 gleich Wert2 ist
#ir&<>	AND<>:	Warten, bis aktueller Portwert AND Wert1 ungleich Wert2 ist
#irw	Wait:	Adresse gibt an, wieviele Schleifendurchläufe gewartet werden soll

Falls nur Byte oder 16-Bit-Werte verwendet werden, wird b (#irb!) bzw w (#irw!) eingefügt.

4.3. mcFORTH Befehle

Diese Befehlsliste ist allgemein für alle mcFORTH-Versionen gültig. Dabei ist zu beachten, dass evtl. durch unterschiedliche Länge des Datenwortes hier auch z.B. c, und , identisch ein kann. Nur bei 32Bit-FORTH gibt es auch `w,` bzw. `w,p` für den 16-Bit-Zugriff.

Aufbau:

: (-- ; C: -- 0; Name) Immediate, Restrict

Einleitung eines neuen Highlevel-FORTH-Befehls mit der Bezeichnung „Name“.

- Befehlsname
- Beschreibung des Stack-Verhaltens und evtl. benötigte Parameter:
  - -- Beschreibt den Datenstack vor und nach Ausführung des Befehls
  - C: -- Beschreibt den Datenstack während des Compilierens
  - R: -- Beschreibt den Returnstack vor und nach Ausführung des Befehls
  - Name Gibt an, ob nach dem Befehl (durch Leerzeichen getrennt) noch etwas folgt
- Befehlstyp
  - Immediate Der Befehl wird auch innerhalb :-Definitionen ausgeführt
  - Restrict Dieser Befehl darf nur innerhalb von :-Definitionen verwendet werden.
  - !!! Achtung: Hier kann die Wortlänge des Prozessors zu Problemen führen.
  - PS Prozessor spezifisch (kann in einzelnen Implementierungen abweichen)
- Beschreibung des Befehls

Für eine bessere Lesbarkeit kennzeichnen hier Groß-/Kleinschreibung folgende Besonderheiten:

- WORT Kontrollstrukturen wie `IF` in Großbuchstaben
- Wort Bei Definitionsbefehle wie `Constant` ist nur der erste Buchstabe groß
- wort Alle anderen Befehle werden klein geschrieben

Verwendete Abkürzungen:

Byte	Ein 8Bit-Wert
Wert	Abhängig von der Datengröße entweder 8, 16, 24 oder 32Bit
c	8Bit-Zeichen (meist Character)
n, n1 …	Beliebige vorzeichenbehaftete Werte
u, u1 …	Beliebige vorzeichenlose Werte
d, d1 …	Beliebige vorzeichenbehaftete doppeltgenaue Werte
ud, ud1 …	Beliebige vorzeichenlose doppeltgenaue Werte
addr, addr1 …	Adresse im Daten- oder Programmbereich
xptr	Pointer in den externen Speicher (benötigt zwei Stackeinträge)
csa	String im Variablenbereich mit Längenbyte und 0-Ende (zählt nicht mit)
f	Wert, der entweder 0 (für Falsch) oder <>0 (meist –1 für Wahr) sein kann

' (Name ; -- csa)

Liefert Codefeldadresse des nachfolgenden Befehls.

'ABORT (error -- error)

Dieses mit `NOP` vorbelegte `DEFER` wird immer bei einem Fehler aufgerufen und kann zur Einbindung eigener Fehlerbehandlungen genutzt werden. Wird das Programm nicht an einer anderen Stelle fortgesetzt, so sollte die verwendete Routine die Fehlernummer auf dem Stack lassen.

'BOOT (--)
Dieser ebenfalls mit NOP vorbelegte DEFER wird beim Start von mcFORTH (vor Ausgabe der Versionnummer und Aufruf von QUIT) aufgerufen und kann zur Realisierung von Autostart-Programmen herangezogen werden.

- (n1 n2 -- n)
Subtraktion zweier Werte: n = n1-n2

-ROT (n1 n2 n3 -- n3 n1 n2)
Rotiert den obersten Stackeintrag unter den vorletzten Eintrag.

! (n addr --)
Speicherung des Wertes n ab Adresse addr im Datenbereich.

!P (n addr --)
Speicherung des Wertes n ab Adresse addr im Programmbereich.

!X (n xptr --)
Speicherung des Wertes n ab Adresse xptr im externen Speicher.

(ud1 -- ud2)
Eine Ziffer (Rest der Division von d1 durch vbase) wird vor den Ausgabestring gestellt.

#> (d -- addr len)
Ende der Zahlenumwandlung löscht Rest und liefert Adresse und Länge des Ausgabestrings.

#BITS (-- n)
Diese Konstante liefert die Anzahl der Bits pro Datenwort (z.B. 16 oder 32).

#C/FB (-- n)
Diese Konstante liefert die Anzahl der möglichen Zeichen im Filepuffer.

#C/TB (-- n)
Diese Konstante liefert die Anzahl der möglichen Zeichen im Terminal-Puffer.

#FB (-- addr)
Diese Konstante liefert die Speicheradresse des Filepuffers.

#MSB (-- u)
In dieser Konstante ist nur das höchstwertige Bit gesetzt (entspricht größten negativen Wert).

#PRGEND (-- addr)
Liefert die höchste mögliche Programmadresse zurück.

#S (ud -- 0.)
wird solange verwendet, bis Ergebnis 0 ist.

#TB (-- addr)
Diese Konstante liefert die Speicheradresse des Terminal-Puffers.

#VAREND (-- addr)
Liefert die höchste mögliche Variablenadresse zurück.

$, (addr len --)
Compiliert den angegebenen String mit Längenangabe und 0-Ende in den Variablenbereich.

$,P (addr len --)
Compiliert den angegebenen String mit Längenangabe und 0-Ende in den Programmbereich.

`$>NUMBER?` (addr len -- addr 0 | d 0> | n --1)

Umwandlung eines Strings in eine Zahl. Dabei wird zuerst auf $ (Hex-Zahlen), & (Dezimal) und % (Binar) geprüft. Ein Punkt im String kennzeichnen doppeltgenaue Werte.

`(` (String) ; --) Immediate

Ignoriert Kommentar bis zum abschließenden) oder bis Zeilenende

`(SFIND` (addr len lfa -- addr len 0 | cfa f)

Sucht nach dem mit addr und len angegebenen Befehl in einem Vokabular (oberste Linkfeldadresse LFA ist angegeben). Wird der Befehl nicht gefunden, verbleiben Adresse und Länge zusammen mit 0 auf dem Stack. Ansonsten wird die Codfeldadresse CFA des Befehls und ein Flag zurückgeliefert, dass entweder 1, 2 (für Restricted) oder -1 bzw. -2 (wenn Immediate) ist.

`*` (n1 n2 -- n3)

Multiplikation der vorzeichenbehaften Werte n1 und n2 zum Ergebnis n3. Überläufe werden dabei ignoriert.

`,` (n --)

Ablage eines Wertes n im Datenbereich.
Der Variablenzeiger `var` wird entsprechend erhöht.

`,P` (n --)

Ablage eines Wertes n im Programmbereichbereich.
Der Dictionaryzeiger `prg` wird entsprechend erhöht.

`.` (n --)

Ausgabe des vorzeichenbehafteten Wertes mit aktueller Zahlenbasis.

`."` (-- ; String") Immediate

Ausgabe eines Strings.

`.R` (n r --)

Ausgabe der vorzeichenbehafteten Zahl in einem Feld der Länge r.

`.S` (--)

Ausgabe der Stacktiefe und der obersten Werte

`/` (n1 n2 -- n3)

Division von n1 durch n2 liefert das Ergebnis n3.

`:` (-- ; C: -- 0; Name)

Einleitung eines neuen Highlevel-FORTH-Befehls mit der Bezeichnung „Name“.
Während des Compilierens wird als Kontrollwert 0 auf dem Stack abgelegt.

`:NONAME` (-- ; C: -- cfa 0)

Ist ähnlich wie : die Einleitung eines Highlevel-FORTH-Befehl. Es wird aber kein Name angegeben und nach dem abschließenden ; bleibt die Codefeldadresse CFA des neuen Befehls auf dem Stack.

`;` (-- ; C: 0 --) Immediate, Restrict

Abschluß des Highlevel-FORTH-Befehls. Es wird während des Compilierens durch die Abfrage der 0 aufgeprüft, ob es am Ende einer :-Definition verwendet wird und keine Programmstrukturen mehr offen ist.

`?ABORT` (error | 0 --)
Bei 0 wird nur überprüft, ob der Stack nicht übergelaufen ist. Ansonsten wird zuerst 'ABORT aufgerufen und nach Ausgabe der entsprechenden Fehlermeldung über QUIT der Interpreter aufgerufen.

`?DO` (end start --) Immediate, Restrict
Die DO-LOOP-Schleife wird nur dann ausgeführt, wenn end <> start ist..

`?DUP` (n | 0 -- n n | 0)
Verdoppelt obersten Stackwert, wenn er ungleich 0 ist.

`?EXIT` (f --) Immediate, Restrict
Verdoppelt obersten Stackwert, wenn er ungleich 0 ist.

`?LEAVE` (f --) Immediate, Restrict
Die DO-LOOP-Schleife wird sofort verlassen, wenn das Flag f <> 0 ist.

`@` (addr -- n)
Lesen eines Wertes ab der Adresse addr aus dem Datenbereich.

`@p` (addr -- n)
Lesen eines Wertes ab der Adresse addr aus dem Programmbereich.

`@X` (xptr -- n)
Lesen eines Wertes ab der Adresse xptr aus dem externen Speicher.

`[` (--) Immediate
Wechsel in den Interpreter-Mode.

`[']` (Name ; -- cfa) Immediate, Restrict
Die Codefeldadresse des nachfolgender Befehl wird als Literal compiliert.

`\` (--) Immediate
Rest der Eingabe oder bis Zeilenende ignorieren

`\\` (--) Immediate
Rest der Eingabe oder bis Screenende ignorieren.

`\ELSE` (--) Immediate
Alternativ-Zweig zu `\IF` , `\IFDEF` , oder `\IFNDEF` .

`\IF` (f --) Immediate
Wenn Flag f <> 0 ist, wird nachfolgendes Sourcecode bis zum `\ELSE` oder dem `\THEN` bearbeitet. Bei f = 0 wird alles bis zum `\ELSE` oder dem `\THEN` ignoriert. Dabei kann `\IF` … `\ELSE` … `\THEN` auch ineinander verschachtelt sein..

`\IFDEF` (name; --) Immediate
Wenn nachfolgender Befehl existiert, wird nachfolgendes Sourcecode bis zum `\ELSE` oder dem `\THEN` bearbeitet. Bei f = 0 wird alles bis zum `\ELSE` oder dem `\THEN` ignoriert. Dabei kann `\IF` … `\ELSE` … `\THEN` auch ineinander verschachtelt sein.

`\IFNDEF` (name; --) Immediate
Wenn nachfolgender Befehl fehlt, wird nachfolgendes Sourcecode bis zum `\ELSE` oder dem `\THEN` bearbeitet. Bei f = 0 wird alles bis zum `\ELSE` oder dem `\THEN` ignoriert. Dabei kann `\IF` … `\ELSE` … `\THEN` auch ineinander verschachtelt sein.

`\THEN` (--) Immediate
Abschluß eines mit `\IF` , `\IFDEF` oder `\IFNDEF` eingeleiteten Textabschnitt.

`]` (--)
Wechsel in den Compiler-Mode.

`+` (n1 n2 -- n)
Addition zweier Werte: n = n1 + n2.

`+LOOP` (n --)
n wird zum Wert der aktuellen `DO-LOOP`-Schleife, solange der Endwert nicht erreicht oder überschritten ist, wird das Programm bei `DO` bzw. `?DO` fortgesetzt.

`<` (n1 n2 -- f)
Vergleich zweier Werte mit f=-1 (sonst 0), wenn n1 < n2 ist.

`<#` (--)
Anfang einer Zahlenumwandlung initialisiert vhld.

`=` (n1 n2 -- f)
Vergleich zweier Werte mit f=-1 (sonst 0), wenn n1 = n2 ist.

`>BODY` (cfa-- pfa)
Ermittelt die Parameterfeld-Adresse aus der Codefeld-Adresse. Der Abstand ist abhängig von der Implementierung z.B. Wortlänge bei indirekt gefädelten Code.

`>IN` (-- n)
Liefert den aktuellen Offset im Eingabepuffer (Tastatur oder File).

`>NAME` (cfa -- nfa)
Ermittelt die Namensfeldadresse eines Befehls oder 0, wenn der Befehl nicht gefunden wurde..

`>NUMBER` (d1 addr1 len1 -- d2 addr2 len2)
Eingabestring (im Variablenbereich) wird in eine Zahl umgewandelt. Dabei wird die aktuelle Zahlenbasis berücksichtigt und die Position des ersten nicht auswertbaren Zeichens bzw. Stringende mit Restlänge (oder 0) zurückgeliefert.

`>R` (n -- ; R: -- n) Restrict
Übertragung eines Wertes vom Datenstack auf dem Returnstack

`>RAM` (--)
Die Compilierung des Programmes erfolgt zusammen mit den Variablen ins RAM. Dies ist nur möglich, wenn es sich bei dem System um eine Neumann-Architektur (Befehl `UNIFIED?`) handelt.

`>RAM?` (-- f)
Liefert -1, wenn das Programm ins RAM compiliert wird.

`>ROM` (--)
Das Programm wird wieder ins Flash compiliert.

`>X` (addr -- xptr)
Ermittelt die externe Adresse eines Variablenspeichers.

`0<` (n -- f)
Testet das Vorzeichen von n und liefert –1 bei Werten < 0 zurück, sonst 0.

`0=` (n -- f)
Testet n und liefert –1 bei Wert = 0 zurück, sonst 0.

`0U.R` (u len --)
Ausgabe des vorzeichenlosen Wertes in len Zeichen mit führenden Nullen.
Dieser Befehl wird z.B. genutzt für Stackanzeige und Speicherdumps.

`1-` (n -- n-1)
Wert um 1 erniedrigen.

`1+` (n -- n+1)
Wert um 1 erhöhen.

`2!` (d addr --)
Speicherung eines doppeltgenauen Wertes.

`2*` (n1 -- n2)
Schieben des Wertes um ein Bit nach links.

`2/` (n1 -- n2)
Schieben des Wertes um ein Bit nach rechts.

`2@` (addr -- d)
Lesen eines doppeltgenauen Wertes aus dem RAM.

`2>R` (d -- ; R: -- d)
Bewegt einen doppeltgenauen Wert vom Datenstack zum Returnstack

`2CONSTANT` (Name ; -- d ; c: d --)
Doppeltgenaue Konstante anlegen.

`2DROP` (d --)
Doppeltgenauen Werten verwerfen.

`2DUP` (d -- d d)
Kopieren des obersten doppeltgenauen Wert.

`2LITERAL` (-- d ; c: d --) Restrict
Doppeltgenauen Wert als Literal im Programm ablegen.

`2OVER` (d1 d2 -- d1 d2 d1)
Kopieren des zweiten doppeltgenauen Wert.

`2R@` (-- d ; R: d -- d)
Kopieren einen doppeltgenauen Wert vom Returnstack zum Datenstack

`2R>` (-- d ; R: d --)
Bewegt einen doppeltgenauen Wert vom Returnstack zum Datenstack

`2SWAP` (d1 d2 -- d2 d1)
Vertauschung der obersten beiden doppeltgenauen Werten.

`2VARIABLE` (name ; c: -- ; -- addr)
Anlage einer doppeltgenauen Variable mit angegebenen Name und Anfangswert 0.

`ABORT` (--)
Ohne weitere Fehlerausgabe wird der Interpretermode aktiviert und `QUIT` aufgerufen. Intern wird dafür die Fehlernummer -2 (z.B. bei `'ABORT`) verwendet.

`ABORT"` (String" ; --)
Nach der Ausgabe des Strings wird der Interpretermode aktiviert und `QUIT` aufgerufen.

`ABS` (n -- u)
Ermittelt den Absolutwert von n.

`ACCEPT` (addr maxlen -- len)
Erwartet einen String mit maximaler Länge maxlen vom Terminal und legt ihn bei addr ab. Zurückgeliefert wird die tatsächlichen Länge.

`AGAIN` (-- ; c: addr 1 --) Immediate, Restrict
Ende einer BEGIN WHILE AGAIN -Schleife.

`AHEAD` (-- ; c: -- addr 2) Immediate, Restrict
Dieser Vorwärts-Sprung wird von `THEN` aufgelöst und z.B. von `ELSE` verwendet.

`ALIAS` (name; cfa --)
Angegebene Codefeldadresse wird als Alias unter neuen Befehlsnamen angelegt.

`ALIGN` (--)
Variablenbereich wird auf die nächste verwendbare Wortadresse aligned, falls notwendig.

`ALIGED` (addr1 -- addr2)
Angegebene Variablenadresse wird auf die nächste Wortadresse aligned, falls notwendig.

`ALIGEP` (addr1 -- addr2)
Angegebene Programmadresse wird auf die nächste Wortadresse aligned, falls notwendig.

`ALIGNP` (--)
Programmbereich wird auf die nächste verwendbare Wortadresse aligned, falls notwendig.

`ALLOT` (n --)
Reserviert n Bytes im Variablenbereich.

`ALLOTP` (n --)
Reserviert n Bytes im Programmbereich.

`ALSO` (--)
Oberstes Context-Vokabular wird kopiert und verbleibt damit auch nach Wechsel des aktuellen Vokabulars weiterhin durchsucht.

`AND` (u1 u2 -- u)
Logische AND-Verknüpfung der Werte u1 und u2.

`ASHIFT` (n count – n2)
n wird um count Bits nach rechts geschoben, wobei das höchstwertige Bit erhalten bleibt. Dabei sind für count nur Werte zwischen 0 und 31 möglich, da die oberen Bits von count abgeschnitten werden.

`AT-MAX` (-- x y)
Ermittelt die maximale Größe des Bildschirms.

`AT-XY` (x y --)
Setzt den Cursor an die angegebene Bildschirmposition.

`AT-XY?` (-- x y)
Abfrage der aktuellen Bildschirmposition.

`BASE` (-- n)
BASE enthält die Zahlenbasis für Umwandlung von String in Zahl und umgekehrt.

`BEGIN` (-- ; C: addr -- 2) Immediate, Restrict
Anfang einer `BEGIN` … `UNTIL` bzw. `BEGIN` … `WHILE` ... `REPEAT`-Struktur.
Während der Kompilierung werden zum Test der Struktur Werte auf dem Stack abgelegt

`BIN` (f1 -- f2)
Modifiziert angegebenens Flag für ein binäres File.

`BL` (-- $20)
Konstante mit dem ASCII-Wert des Leerzeichen (meist $20).

`BODY>` (pfa -- cfa)
Ermittelt die Codefeldadresse aus der Datenfeldadresse.

`BYE` (--)
Verlassen des FORTH und evtl. Neustart des Microcontroller-Systems.
Ein evtl. noch offenes File (FID <> 0) wird vorher noch geschlossen.

`C!` (c addr --)
Speicherung des Zeichens ab Adresse addr im Datenbereich.

`C!P` (c addr --)
Speicherung des Zeichens ab Adresse addr im Programmbereich

`C!X` (c xptr --)
Speicherung des Zeichens ab Adresse xptr im externen Speicher.

`C,` (c --)
Ablage des Zeichens im Datenbereich.
Der Variablenzeiger vdp wird entsprechend erhöht.

`C,P` (c --)
Ablage des Zeichens im Programmbereich.
Der Programmzeiger `PRG` wird entsprechend erhöht.

`C@` (addr -- c)
Lesen eines Zeichens ab Adresse addr aus dem Datenbereich.

`C@P` (addr -- c)
Lesen eines Zeichens ab Adresse addr aus dem Programmbereich.

`C@X` (xptr -- c)
Lesen eines Zeichens ab Adresse xptr aus dem externen Speicher.

`CAPACITY` (-- len)
Liefert die Anzahl der Screens im aktuell geöffneten File.

`CELL+` (addr1 -- addr2)
Erhöht die Adresse um die Länge eines Datenwortes.

`CELLS` (n -- len)
Ermittelt die Anzahl von Adressen für n Datenworte.

`CHAR` (Name ; -- char) Immediate
Liefert den ASCII-Wert des ersten Zeichens im nachfolgenden String.

`CHAR+` (addr1 -- addr2)
Erhöht die Adresse um die Länge eines Zeichens.

`CHARS` (n -- len)
Ermittelt die Anzahl von Adressen für n Zeichen.

`CLOSE` (--)
Ein evtl. noch offenes File (`FID` <> 0) wird geschlossen.

`CMOVE` (addr1 addr2 len --)
Kopieren des Speicherbereiches im RAM von addr1 nach addr2.

`CMOVEP` (addr1 addr2 len --)
Kopieren des Speicherbereiches von addr1 im ROM zum RAM ab addr2.

`CONSTANT` (n -- ; Name)
Definition der Konstante Name mit dem Wert n.

`CONTEXT` (-- addr)
Diese Variable enthält die Adresse des Vokabulars, in der aktuell die neuen Befehle compiliert werden. Verändert wird dieser Wert nur durch die Befehle `ONLYFORTH` und `DEFINITIONS`.

`COUNT` (addr -- addr+1 c)
Lesen eines Zeichens ab Adresse addr aus dem Datenbereich.
Die Adresse des nächsten Zeichens bleibt dabei auf dem Datenstack.

`COUNTP` (addr -- addr+1 c)
Lesen eines Zeichens ab Adresse addr aus dem Programmbereich.
Die Adresse des nächsten Zeichens bleibt dabei auf dem Datenstack.

`CR` (--)
Ausgabe von Carriage Return ($0D dann $0A) auf dem Terminal.

`CREATE` (-- ; C: -- 9 ; Name) Immediate
Definition des Befehls Name. Es ist innerhalb :-Definitionen immer ein Doesv>-Teil notwendig, der die Adresse im Variablenbereich zurückgeliefert. Dieser Befehl wird verwendet, um z.B. Datenstrukturen im Variablenbereich aufzubauen.
Achtung: Verwendung von `Doesp>` und `,p` bzw. `c,p` führen zu schwer auffindbaren Fehler

Beispiel:
```
: Counter:  ( name ; -- ; -- n )
   Create 0 , Does> dup >r  @ 1 +  dup r> ! ;
Counter: nextnum
```

Mittels Counter: definiert man den Befehl `nextnum`. Bei jedem Aufruf von `nextnum` wird beginnend mit 1 die nächst größere Zahl geliefert.

Createv kann auch außerhalb :-Definitionen verwendet werden, um einen Befehl zu definieren, der die nächste Variablenadresse zurückliefert.

`CREATEP` (-- ; C: -- addr 8 ; Name) Immediate
Definition des Befehls Name, welches die nachfolgende Programmadresse zurückliefert. Ist innerhalb der `CreateP`-Definitionen ein `Doesp>`-Teil angegeben, kann dieser die Adresse verwendet. Solche Definitionen werden verwendet, um z.B. Tabellen im Programmbereich aufzubauen.
Achtung: Verwendung von `Does>` und `,` bzw. `c,` führen zu schwer auffindbaren Fehler

Beispiel:
```
: X:  ( n -- ; -- n )
   Createp 1 - ,p   Doesp> @p  FOR $40 emit NEXT ;
6 X: 6@
8 X: 8@
```

Mittels `X:` definiert man die Befehle `6@` und `8@`, welche die angegebene Zahl (-1 da `FOR`-Schleifen dies so braucht) speichern und bei Aufruf den `Doesp>`-Teil von `X:` ausführen und damit die Zahlen lesen und die entsprechende Anzahl von „@“ (=$41) ausgeben.

Doesp> kann auch außerhalb der Createp-Definitonen verwendet werden, erwartet aber als letzte Definition ein Wort, dass mit Createp definiert wurde.

`CURRENT` (-- addr)

Diese Variable enthält den Offset ab CURRENT CELL+ auf das oberste Suchvokabular. Aktuell werden bis zu 6 Vokabulare in diesem Speicherbereich verwaltet.

`D-` (d1 d2 -- d3)

Ermittelt Differenz d1-d2 zweier doppeltgenauer Werte.

`D.` (d --)

Ausgabe des doppeltgenaue Werte.

`D.R` (d n --)

Ausgabe des doppeltgenaue Werte rechtsbündig in einem Feld mit n Zeichen.

`D+` (d1 d2 -- d3)

Addiert zwei doppeltgenaue Werte.

`D<` (d1 d2 -- f)

Der Vergleich zweier doppeltgenaue Werte liefert TRUE (sonst FALSE), wenn d1 < d2.

`D=` (d1 d2 -- f)

Der Vergleich zweier doppeltgenaue Werte liefert TRUE (sonst FALSE), wenn d1 = d2.

`D0<` (d -- f)

Liefert TRUE (sonst FALSE), wenn der doppeltgenaue Wert d negativ ist.

`D0=` (d -- f)

Liefert TRUE (sonst FALSE), wenn der doppeltgenaue Wert d = 0 ist.

`D2*` (d1 -- d2)

Verdoppelt ein doppeltgenauen Wert durch Shifted um ein Bit nach Links.

`D2/` (d1 -- d2)

Halbieren ein doppeltgenauen Wert durch arithmetisches Shifted um ein Bit nach Rechts. Dabei bleibt das höchstwertige Bit erhalten.

`DABS` (d -- ud)

Ermittelt den Absolutwert der doppeltgenauen Zahl d.

`DECIMAL` (--)

Setzt die Zahlenbasis `BASE` auf 10 und erzwingt damit eine dezimale Ein-/Ausgabe.

`DEFER` (name; --)

Nachfolgender Befehl wird eine Definition, die am Anfang eine Fehlermeldung verursacht. Mit IS kann ein anderer Befehl diesem Defer zugeordnet werden.

Beispiel: Zeichenroutine mit veränderbarer Funktion

```
Defer fn            \ Funktion
: printfn   ( -- ) \ Ausgabe von Funktionswerte
   10 0 DO i 3 .r  10 spaces  i fn 3 .r cr LOOP ;
: linear ;   ( n -- n ) \ Lineare Zuordnung
: invers   ( n -- 10-n ) \ Umkehrung
   10 swap - ;
: Demo   ( -- )
  cr  ." Lineare Zuordnung : "  cr
  ['] linear is fn  printfn
  cr  ." Inverse Zuordnung : "  cr
  ['] invers is fn  printfn ;
demo
```

DEFER wird meist bei Befehle verwendet, die entweder erst später definiert oder während eines Programmlaufes (z.B. bei einer State-Maschine) verändert wird.

`DEFINITIONS` (--)

Oberstes Suchvokabular wird zum Vokabular, in das die nächsten Befehle compiliert werden.

`DO` (end start --) Immediate, Restrict

Anfang einer DO-LOOP-Schleife erwartet Endwert+1 und Anfangswert.

`DOES>` (-- addr ; C: 9 --) Immediate

Ausführungsteil einer `Vcreate-Vdoes>`-Struktur hat die Anfangsadresse des Datenbereiches bei Definition des Wortes auf dem Datenstack. Beim Compilieren wird abgefragt, ob es eine `Doesv`-Definition mit aufgelösten Befehlsstrukturen ist.

`DOESP>` (-- addr ; C: 8 --) Immediate, Restrict

Ausführungsteil einer `Createp-Doesp>`-Struktur hat die Anfangsadresse des Programmbereiches des mit `Createp` definierten Wortes auf dem Datenstack. Beim Compilieren wird abgefragt, ob es eine `Doesv`-Definition mit aufgelösten Befehlsstrukturen ist.

`DPL` (-- n)

`DPL` enthält die Position des Punktes bei Umwandlung von String in Zahl (oder –1).

`DROP` (n --)

Entfernt den obersten Datenstackeintrag.

`DU<` (ud1 ud2 -- f)

Vergleich zweier vorzeichenlosen doppelgenauen Werten liefert TRUE, wenn ud1 < ud2.

`DU2/` (ud1 -- ud2)

Halbieren eines doppeltgenauen Wertes durch Schieben nach rechts. Dabei wird das höchstwertige Bit 0.

`DUMIN` (ud1 ud2 -- ud1 | ud2)

Hinterläßt den kleineren Wert der beiden vorzeichenlosen doppeltgenauen Werte.

`DUMP` (addr len --)

Speicherdump des Variablenbereiches.

`DUMPP` (addr len --)

Speicherdump des Programmbereiches.

`DUMPX` (xptr len --)

Speicherdump des externen Speicherbereiches kann auch für Programm (`addr p>x`) oder Variablen (`addr >x`) verwendet werden.

`DUP` (n -- n n)

Kopiert den obersten Datenstackeintrag.

`ELSE` (-- ; C: addr1 -- addr2 –1) Immediate, Restrict

Mit ELSE beginnt der alternative Teil einer IF-ELSE-THEN-Struktur.
Der Compiler nutzt den Datenstack für Korrektur von Sprungadressen und Flags.

`EMIT` (c --)

Ausgabe eines Zeichens auf dem Terminal

`EMIT?` (-- f)

Abfrage, ob eineZeichen auf dem Terminal ausgegeben werden kann.

`END-CREATE` (--)

Schließt ein Befehl ab, schreibt alle gepufferten Daten in das Flash und macht den letzten Befehl sichtbar. Ist nur bei Verwendung von Create bzw. CreateP erforderlich, weil es bei allen Definitionsbefehlen (: mit ; bzw. Variable und Constant …) implizit verwendet wird.

`EVALUATE` (addr len --)

Angegebener String wird ausgewertet (Interpretiert oder Compiliert).

`EXECUTE` (cfa --)

Ausführen eines FORTH-Befehls ab Adresse cfa.

`EXIT` (--) Restrict

Verlassen des aktuellen Befehls.

`FALSE` (-- 0)

Liefert den Wert für FALSE - bei mcFORTH meist 0 - zurück.

`FCB` (-- addr)

Variable enthält einen Zeiger auf den aktuellen Filename.

`FCLOSE` (fid -- error)

Schließen des offenen Files.

`FCREATE` (csa -- fid 0 | error)

Erzeugt ein neues File mit Filename ab csa (Achtung: 0-Ende notwendig) und liefert Filehandle oder Fehlernummer zurück.

`FID` (-- fid)

`FID` enthält die aktuelle File-ID, aus dem das Programm geladen wird.

`FILEVEC` (name ; max -1 | n -- ; --)

Mit dieses mit Vector definierte Wort ist das Filehandling vektorisiert worden. Mit n=0 bis 11 wurden die Befehle FINIT bis FWRITE definiert worden. Die genaue Verwendung von Vektor und den Implementierungen des Terminal- und File-Handling ist in der obigen Beschreibung nachzulesen.

`FINIT` (n --)

Je nach n wird das Fileinterface zum ersten mal initialisiert (n=-1 bei Neustart) oder wieder deinitalisiert (n=0 bei Ende des Programmes in `BYE`). Dieser Befehl kann z.B. dazu verwendet werden, die Stromversorgung für SD-Card oder Festplatte ein-/auszuschalten und Interrupts zu initialisieren oder wieder freizugeben.

`FOPEN` (csa rw -- fid 0 | error)

Öffnet ein vorhandenes File und liefert die File-ID oder eine Fehlermeldung zurück. Das Flag rw gibt an, ob das File nur zum Lesen (`R/O`), zum Schreiben (`W/O`) oder zur freien Bearbeitung (`R/W`) evtl. im binären Modus (`BIN`) geöffnet wird.

`FOR` (u -- ; R: -- u ; C: -- addr 3) Immediate, Restrict

Der Anfang einer FOR-NEXT-Schleife legt den Schleifenwert (meist Inline mit >R) auf den Returnstack. Die Schleife wird u+1 mal durchlaufen. Der aktuelle Schleifenwert kann mit r@ abgefragt werden. Der Compiler nutzt den Datenstack für Korrektur von Sprungadressen und Flags.

`FORTH` (--)

Das Vokabular `FORTH` wird als `CURRENT` gesetzt und ist damit das erste in die Suchreihenfolge.

`FPOS!` (d dir fid – d2 0 | error)

Positionierung des Lese/Schreib-Zeigers im File. Es wird ab Fileanfang (dir=0), aktueller Position (dir=1) oder Fileende (dir=2) positioniert. Ist kein Fehler aufgetreten, wird die aktuelle Position zurückgeliefert.

`FREAD` (xptr len fid --len2 0 | error)

Lesen von len Zeichen aus dem File nach xptr . Wenn kein Fehler aufgetreten ist, wird die Anzahl der gelesenen Zeichen zurückgeliefert, die auch ungleich der angebenen len sein kann (z.B. wenn kein Zeichen von Terminal COMx verfügbar ist).

`FSIZE@` (-- d)

Liefert die Länge des aktuell geöffneten Files als doppeltgenaue Zahl zurück.

`FWRITE` (xptr addr len fid –0 | error)

Schreiben von len Zeichen ab xptr in das File.

`HERE` (-- addr)

Liefert die Adresse zurück, ab der weitere Variablen ins RAM abgelegt werden. Dieser Wert entspricht immer dem Inhalt von Variable `VAR`.

`HEREP` (-- addr)

Liefert die Adresse zurück, ab der weitere Programme compiliert werden. Dabei handelt es sich entweder um eine Flash-Adresse (`RAM?` liefert 0) oder um RAM. Der Wert wird entweder aus der Variable `VAR` oder aus `PRG` gelesen.

`HEX` (--)

Setzt die Zahlenbasis `BASE` auf 16 und erzwingt damit eine hexadezimale Ein-/Ausgabe.

`HLD` (-- n)

Variable `hld` enthält die aktuelle Adresse des Stringanfangs bei Zahlenausgabe.

`HOLD` (char --)

Zeichen wird in den aktuellen Ausgabestring eingebaut.

`I` (-- n)

Liefert den aktuellen Wert der obersten DO-LOOP-Schleife zurück. Dabei darf aber der Returnstack nicht mit zusätzlichen Werten belegt sein.

`I'` (-- n)

Liefert den Endwert der obersten DO-LOOP-Schleife zurück. Dabei darf aber der Returnstack nicht mit zusätzlichen Werten belegt sein.

`I:` (--)

Ersatz für das normalerweise verwendete `IMMEDIATE` nach dem Befehl, da in mcFORTH das ROM nicht mehr nachträglich verändert werden kann/soll. Wird genutzt, wenn nächster Befehl als Immediate deklariert werden soll. So gekennzeichnete Befehle werden auch innerhalb :-Definition ausgeführt.

`IF` (f -- ; C: -- addr 1) Immediate, Restrict

Der Anfang der IF-ELSE-THEN-Bedingung prüft das Flag f auf dem Datenstack und führt den IF-Teil bei Werten ungleich 0 aus. Ansonsten wird der ELSE-Teil ausgeführt oder direkt zu THEN gesprungen.
Der Compiler nutzt den Datenstack für Korrektur von Sprungadressen und Flags.

`INCLUDE` (File ; --)

Öffnet das nachfolgend angegebene File und laden den Screen 1.

`INTERPRET` (--)

Der aktuelle Programmquelle (Terminal, File oder String aus `EVALUATE`) wird bis zum Ende interpretiert (`STATE` ist 0) oder compilert.

`IOINIT` (n --)

Je nach n wird das Fileinterface zum ersten mal initialisiert (n=-1 bei Neustart) oder wieder deinitalisiert (n=0 bei Ende des Programmes in `BYE`). Dieser Befehl kann z.B. dazu verwendet werden, die Stromversorgung für SD-Card oder Festplatte ein-/auszuschalten und Interrupts zu initialisieren oder wieder freizugeben.

`IOVEC` (Name ; n -- | max -1 --)

Über diesen Vektor sind die Befehle für das Terminal vektorisiert worden (n=0..9 für IOINIT … AT-MAX

`IS` (Name ; cfa --)

Der angegebene Defer-Befehl wird auf eine neue Codefeldadresse umgeleitet. Die Verwendung ist bei `DEFER` beschrieben.

`J` (-- n)

Liefert den aktuellen Wert der nächst inneren DO-LOOP-Schleife zurück. Dabei darf aber der Returnstack nicht mit zusätzlichen Werten belegt sein.

`J'` (-- n)

Liefert den Endwert der nächst inneren DO-LOOP-Schleife zurück. Dabei darf aber der Returnstack nicht mit zusätzlichen Werten belegt sein.

`KEY` (-- c)

Wartet, bis ein Zeichen verfügbar ist und legt den Wert auf dem Stack ab.

`KEY?` (-- f)

Flag, ob ein Zeichen verfügbar ist.

`LEAVE` (-- ; R: ??? --) Restrict

Die äußerste DO-LOOP-Schleife wird sofort verlassen und die entsprechenden Daten vom Returnstack entfernt. Dieser Befehl darf nur innerhalb von Programmen verwendet werden und kann bei zusätzlichen Daten auf dem Returnstack zum Absturz führen.

`LITERAL` (n --) Immediate, Restrict

Compiliert den entsprechenden Wert in das Programm

`LOAD` (n --)

Entsprechender Screen wird aus dem aktuell geöffneten File geladen.

`LOADFROM` (File ; n --)

Entsprechender Screen wird aus dem angegebenen File geladen.

`LOOP` (--) Restrict

Der Wert der äußerste DO-LOOP-Schleife wird um 1 erhöht und das Programm bei DO bzw ?DO fortgesetzt, solange der Endwert nicht erreicht ist.

`LSHIFT` (u count – u2)

u wird um count Bits nach links geschoben. Dabei sind für count nur Werte zwischen 0 und 31 möglich, da die oberen Bits von count abgeschnitten werden.

`M*` (n1 n2 -- d)

Zwei vorzeichenbehaftete Werte werden zu einem doppeltgenauen Wert multipliziert.

`M/MOD` (d n -- r q)

Der doppeltgenaue vorzeichenbehaftete Wert d wird durch n geteilt, wobei neben dem Quotient q auch der Rest r zurückgeliefert wird.

`NAME>` (nfa -- cfa)

Aus der Namensfeldadresse wird die Codefeldadresse ermittelt.

`NEXT` (-- ; u -- u-1 | ; C: addr 3 --) Immediate, Restrict

Das Ende einer FOR-NEXT-Schleife prüft den Scheifenwert auf dem Returnstack und erniedrigt ihn um 1. Ist er schon 0, so wird nach Entfernung des Wertes das Programm hinter NEXT fortgesetzt. Ansonsten wird die Schleife ab FOR wiederholt. Der Compiler nutzt den Datenstack für Abfrage der Kontrollstruktur und der Korrektur der Sprungadresse.

`NIP` (n1 n2 -- n2)

Zweiter Stackwert entfernen.

`NOP` (--)

Dieser Befehl wird z.B. bei `DEFER`'s verwendet und hat keine Funktion.

`ONLYFORTH` (--)

`FORTH` wird sowohl zum `CURRENT` als auch zum einzigen `CONTEXT`-Vokabular. Alle Befehle werden im `FORTH`-Vokubular abgelegt und es wird auch nur `FORTH` durchsucht.

`OPEN` (Name ; --)

Öffnet das angegebene Files zum Lesen.
Das zuvor offene File (`FID`) wird vorher geschlossen und der Puffer als leer markiert.

`OR` (u1 u2 -- u)

Logische OR-Verknüpfung der Werte u1 und u2.

`ORDER` (--)

Es wird zuerst das `CURRENT`-Vokabular (in dem der nächste Befehl geschrieben wird) und dann alle `CONTEXT`-Vokabulare (oberstes, zuerst durchsuchtes zuletzt) ausgegeben.

`OVER` (n1 n2 -- n1 n2 n1)

Kopiert den zweiten Stackeintrag.

`P"` (String" -- ; -- csa) Restricted, Immediate

Nachfolgender String (ohne abschließendes ")wird mit Längenbyte und 0-Ende in das Programm übernommen. Bei Ausführung wird die Adresse des Längenbytes zurückgeliefert.

`P>X` (addr -- xptr)

Umrechnung der Programmadresse in eine externe Adresse, die z.B. mit XDUMP angezeigt werden kann..

`PAGE` (--)

Bildschirm löschen und Cursor an die oberste linke Position setzen.

`PARSE` (char -- addr len)

Aus dem aktuellen Quellcode (Terminal, File oder `EXECUTE`-String) wird der nächste, durch das angegebene Zeichen begrenzte String extrahiert. Bei `-PARSE` werden im Gegensatz zu `PARSE` die führenden Zeichen nicht ignoriert und führen zu einem String mit Länge 0.

`PARSE-WORD` (-- addr len)

Aus dem aktuellen Quellcode (Terminal, File oder `EXECUTE`-String) wird der nächste, durch das angegebene Zeichen begrenzte String extrahiert. Bei `PARSE-WORD` werden im Gegensatz zu `PARSE` die führenden Zeichen ignoriert.

`POSTPONE` (Name ; --) Immediate, Restrict

Nachfolgender Befehl wird compiliert.

`PREVIOUS` (--)

Aus der Suchreihenfolge CONTEXT wird das oberste Vokabular entfernt.

`PRG` (-- addr)

Diese Variable enthält die Adresse der ersten freien Programmadresse.

`QUIT` (-- ; R: ??? --)

Beenden der Interpretation, löschen des Returnstack und Einsprung in den Interpreter.

`R/O` (-- n)

Liefert ein Flag, dass beim Öffnen eines Files angibt, dass nur gelesen wird.

`R/W` (-- n)

Dieses Flag zum Öffnen eines Files angibt, dass sowohl gelesen als auch geschrieben wird.

`R:` (--)

Ersatz für das normalerweise verwendete `RESTRICT` nach dem Befehl, da in mcFORTH das ROM nicht mehr nachträglich verändert werden kann/soll. Diese Befehle sind dann nur innerhalb :-Definitionen zulässig.

`R@` (-- n ; R: n -- n) Restrict

Kopieren des obersten Returnstackeintrages.

`R>` (-- n ; R: n --) Restrict

Holt den obersten Returnstackeintrag zum Datenstack

`RDEPTH` (-- n)
Ermittelt die Anzahl von Einträgen auf den Returnstack.

`RDROP` (-- ; R: n --)
Entfernen des obersten Returnstackeintrages.

`RECURSE` (--) Restrict
Ruft die aktuelle Definition rekursive auf.

`REFILL` (-- f)
Zum Auffüllen des Source-Puffers wird entweder vom Terminal die nächste Eingabe erwartet (`SOURCE-ID` ist 0) oder aus dem aktuellen File die nächste Zeile gelesen. Sind neue Daten verfügbar, so wird `TRUE` zurückgeliefert. Bei Fileende oder wenn bisher ein `EVALUATE`-String verwendet wurde (`SOURCE-ID` ist -1), so wird `FALSE` zurückgeliefert.

`REPEAT` (-- ; C: ??? ±2 --) Immediate, Restrict
Ende eine BEGIN-WHILE-REPEAT-Schleife mit Rücksprung zu BEGIN ohne Bedingung. Der Compiler nutzt die Werte auf dem Datenstack für Korrektur von Sprungadressen.

`RINIT` (-- ; R: ??? --) Restrict
Der Returnstack wird gelöscht.

`ROM?` (-- f)
Dieser Befehl liefert TRUE zurück, wenn das mcFORTH über einen Flash- bzw. ROM-Bereich verfügt und damit Programm und Daten getrennt sein können.

`ROT` (n1 n2 n3 -- n2 n3 n1)
Rotiert die obersten drei Stackeinträge

`RSHIFT` (u count – u2)
u wird um count Bits nach rechts geschoben, wobei das höchstwertige Bit mit 0 aufgefüllt wird. Dabei sind für count nur Werte zwischen 0 und 31 möglich, da die oberen Bits von count abgeschnitten werden.

S“ (String“ -- addr len) Immediate
Nachfolgender String (ohne abschließendes “)wird mit 0-Ende zum Ende des Variablenbereiches kopiert und die Adresse des Anfangs zurückgeliefert. Beim Compilieren wird der String inline (im Programmbereich) abgelegt aber bei Ausführung wieder an das Ende des Variablenbereichs kopiert.

`SAVE` (Name ; --)
Speichert das aktuelle Programm als ausführbares Image im angegebenen File.

`SDEPT` (-- n)
Liefert die Anzahl der Werte auf dem Datenstack zurück.

`SFIND` (csa -- csa 0 | cfa f)
Sucht nach dem Befehl und liefert bei Erfolg die Codefeldadresse und ein Flag zurück.
Flag = 2 : Befehl ist restricted (ansonsten 1).
Negative Werte kennzeichnen Immediate-Befehle (auch hier –1 oder –2)

`SIGN` (n --)
Einbinden des „-“ – Zeichens in den Ausgabestring, wenn n negativ ist.

`SINIT` (??? --)
Initialisiert den Datenstack.

`SOURCE` (-- addr len)

Liefert Anfangsadresse und Länge des aktuellen Source-Codes. Um die aktuelle Position zu ermitteln, muß noch der Offset in `>IN` berücksichtigt werden.

`SOURCE-ID` (-- addr)

Diese Variable enthält das Flag für die aktuelle Quelle des Source-Codes. Dieser kann entweder -1 für ein `EVALUATE`-String, 0 für das Terminal, 1 für ein sequenzielles File oder 2 für ein Screen-File sein. Im Anschluß an dieses Flag sind im Variablenbereich noch aktueller Offset und Restlänge als doppeltgenaue Werte abgelegt.

`SPACE` (--)

Ausgabe eines Leerzeichens auf dem Terminal.

`SPACES` (n --)

Ausgabe von n Leerzeichens auf dem Terminal.

`STATE` (-- n)

`STATE` enthält das Flag, ob Interpretiert (`FALSE`) oder Compiliert (`TRUE`) wird.

`STDFILE` (--)

Dieser mit FILEVEC definierte Befehl aktiviert die Verwendung des Standard-Filesystems, was beim Simulator meist das Betriebssystem ist. Bei Microcontroller wird oft transparent über die RS232 auf das Filesystem des Terminalrechners zugegriffen.

`STDIO` (--)

Dieser mit FILEVEC definierte Befehl aktiviert die Verwendung der Standard-Ein-/Ausgabe, was beim Simulator meist die Tastatur und der Bildschirm ist. Bei Microcontroller wird oft transparent über die RS232 mit einem Terminalprogramm gearbeitet.

`SWAP` (n1 n2 -- n2 n1)

Vertauschung der obersten beiden Stackeinträge

`THEN` (-- ; C: addr ±1 --) Immediate, Restrict

Ende einer IF-(ELSE-)THEN-Struktur ändert die Stacks nicht. Der Compiler nutzt den Datenstack für Test der Kontrollstruktur und für die Korrektur von Sprungadresse.

`THRU` (n1 n2 --)

Ladet die Screens n1 bis einschließlich n2.

`TRUE` (-- -1)

Diese Konstante liefert den Wert für TRUE (meist -1) zurück.

`TUCK` (n1 n2 -- n2 n1 n2)

Kopiert den obersten Stackwert und legt ihn unter den zweiten Wert.

`TYPEP` (addr len --)

Ausgabe eines Strings im Programmbereich.

`TYPE` (addr len --)

Ausgabe eines Strings im Variablenbereich.

`U.` (u --)

Ausgabe des Wertes als vorzeichenlose Zahl gemäß aktueller Zahlenbasis.

`U.R` (u r--)

Ausgabe des vorzeichenlosen Wertes in einem Feld mit mindestens r Zeichen.

`U<` (u1 u2 -- f)
Vergleich zweier vorzeichenloser Werte mit f=-1 (sonst 0), wenn u1 < u2 ist.

`U2/` (u1 -- u2)
Das logische Schieben von u1 um ein Bit nach rechts entspricht einer Halbierung des Wertes.

`UM*` (u1 u2 -- ud)
Multiplikation zweier vorzeichenloser Werte mit doppeltgenauem Ergebnis.

`UM/MOD` (du u -- u1 u2)
Division einer doppelt genauer vorzeichenloser Zahl mit Rest u1 und Quotient u2

`UMIN` (u1 u2 -- u)
Liefert den kleineren der beiden vorzeichenlosen Werte zurück.

`UNIFIED?` (-- f)
Liefert `TRUE` zurück, wenn Daten und Programm mit gleichen Befehlen adressiert und deshalb auch ein Programm im RAM laufen kann. Bei `FALSE` wird der Befehl `>RAM` ignoriert und das Programm wird immer ins Flash compiliert.

`UNLOOP` (-- ; R: ??? --) Restrict
Wird innerhalb einer DO-LOOP-Schleife verwendet, um die Werte und Rücksprungadressen zu entfernen. Danach muß aber er Befehl vor dem `LOOP` verlassen werden.

`UNTIL` (f -- ; C: ??? ±2 --) Immediate, Restrict
Ende eine `BEGIN-WHILE-UNTIL`-Schleife mit Rücksprung zu `BEGIN` wenn f=0. Der Compiler nutzt den Datenstack für Test der Kontrollstruktur und für die Korrektur von Sprungadresse.

`UNUSED` (-- n)
Liefert die Größe des noch freien Bereiches im RAM zurück.

`UNUSEDP` (-- n)
Liefert die Größe des noch freien Bereiches im Flash zurück.

`UPC` (char1 -- char2)
Wandelt das angegebene Zeichen in die entsprechende Großschrift.

`USER` (name; --)
Anlage einer USER-Variable mit angegebenen Namen und Startwert 0.

`V` (--)
Ruft den Editor mit der letzten Fehlerposition auf..

`VAR` (-- addr)
Diese Variable enthält die nächste freie Adresse in Variablenbereich (RAM).

`VARIABLE` (name ; --)
Anlage einer Variable mit angegebenen Name und dem Startwert 0.

`VOC` (-- addr)
Diese Variable enthält einen Zeiger auf das zuletzt definierte Vokabular.

`VOCABULARY` (name ; --)
Anlage eines neuen, leeren Vokabulars mit angegebenen Name.

`VOCS` (--)
Anzeige aller im mcFORTH verfügbaren Vokabulare.

`W/O` (-- n)

Liefert ein Flag, dass beim Öffnen eines Files angibt, dass nur geschrieben wird.

`WHILE` (f -- ; C: ??? ±2 -- ??? ±2 addr –2) Immediate, Restrict

Bei `WHILE` wird eine `BEGIN-UNTIL` bzw. `BEGIN-REPEAT`-Schleife verlassen, wenn f=0 ist. Der Compiler nutzt den Datenstack für Ablage von Sprungadressen und zum Test der Controllstruktur.

`WITHIN` (n n1 n2 -- f)

Test, ob n innerhalb n1 bis n2 (zählt nicht mit) liegt.

`WORDS` (--)

Listet alle verfügbaren FORTH-Befehle auf.

`X+` (xptr1 u -- xptr2)

Addiert zum Zeiger in den externe Speicher den angegebenen Offset.

`XALLOC` (len. -- xptr 0 | error)

Reserviert im externen Speicher die hier als doppeltgenau angegebene Zahl von Adressen und liefert einen Pointer zurück.

`XFREE` (xptr. -- error)

Gibt den vorher reserviert externen Speicher wieder frei.

`XOR` (u1 u2 -- u)

Logische XOR-Verknüpfung der Werte u1 und u2.

5. Erweiterungen für viele mcFORTH-Versionen

5.1. Der Assembler (vp32_asm.scr bzw. m0_asm.scr)

Der Assembler bzw. Targetassembler übersetzt die Mnemonics des Prozessors in das Binärimage. Dabei ist er auch in der Lage, eine begrenzte Anzahl von lokalen Adressen als Labels zu verwalten und je nach Anforderung entweder eine Absolutadresse (bei Literals) oder Relativadressen (Sprünge) im OpCode abzulegen. Dabei soll der (Flash-)Speicher bis zur endgültigen Compilierung der Adresse nicht verändert werden. Zur höheren Sicherheit wird im Assembler auch der Stack überwacht und ein Fehler angezeigt, wenn am Ende eines Assemblerbereiches der Stack verändert ist.

5.1.1. Initialisierung und Test

Beim Eintritt in eine neue Assembler-Definition mit Proc, Code oder ;Code wird die Tabelle für Labels und Variablen gelöscht, das Flag für die Anzahl der Argumente auf 0 gesetzt und die Anzahl der momentan auf dem Datenstack abgelegten Daten gespeichert.

Am Ende einer Assembler-Definition mit End-Proc oder End-Code wird nochmals geprüft, ob der Stack wieder den Anfangszustand erreicht hat. Anschließend werden die noch offenen Labels aufgelöst und der Assembler beendet.

5.1.2. Labels und lokale Adressen

Im mcFORTH können headerlose Befehle oder Labels angelegt werden. Innerhalb von Lowlevel-Routinen sind lokale Adressen möglich, wobei beachtet werden muß, daß diese Sprünge als relative Adressen decodiert und bei Flash nur nach Auflösung der Adressen gespeichert werden dürfen. Dies führt dazu, daß im Targetcompiler alle Links bis zur Auflösung verwaltet werden müssen.

Beispiel:

```
Code test   ( n -- x ) \ Adresse von test, wenn n<>0, sonst 0
      1$:   2$ (0branch,
            1$ (lit,   (exit,
      2$:   0 (lit,   (exit,
End-Code
```

Ablauf: Bei $x: wird die aktuelle Adresse (in $tab) gespeichert
Bei $x wird die Adresse, Label-Nummer und Relativ-Flag (in $link) gespeichert
(Ist die Zieladresse schon bekannt, wird der Sprung sofort aufgelöst)
Bei End-Code werden die offenen Links in $link aufgelöst

5.2. Der mcFORTH Screen Editior (mcFEdWin.scr)

Bei F.I.G-, F83- und volksFORTH war der FORTH-Screen mit 16 Zeilen zu 64 Zeichen das Standard-Fileformat, was auch einfache Nutzung von EEPROM's und Flash für Filesystem erlaubt. Der Screen-Editor für mcFORTH ist eine spezielle Version, welche auch an kleinere Bildschirme bis herunter auf 6*20 Zeichen angepaßt werden kann.

Cursorsteuerung:

^E oder Cursor hoch	Eine Zeile höher
^X oder Cursor tief	Eine Zeile tiefer
^S oder Cursor links	Ein Zeichen links

^D oder Cursor rechts	Ein Zeichen rechts
TAB	Zur nächsten 4er-Teilung
^TAB	Zur vorherigen 4er-Teilung
^Cursor links	Zum Zeilenanfang
^Cursor rechts	Zum Zeilenende-1
POS1	Zum Screenanfang
Return	Zum nächsten Zeilenanfang oder erste Zeile
^R oder Bild_hoch	Zum vorhergehenden Screen
^C oder Bild_tief	Zum nächsten Screen
^Bild_hoch	Zum ersten Screen
^Bild_tief	Zum letzten Screen
POS1	Zum ersten Zeichen der Zeile
ENDE	Hinter das letzte Zeichen der Zeile
^POS1	Zum ersten Screens
^ENDE	Zum letzten Screens
^G	Goto (Screennummer wird abgefragt)

Zwischenspeicherung von Zeichen und Zeilen:

F1	Zeichen speichern und löschen
F2	Zeichen speichern, Cursor rechts
F3	Zeichen einfügen
F4	Zum zweiten File / Cursorposition umschalten
F5	Zeile speichern und löschen
F6	Zeile speichern, Cursor in die nächste Zeile
F7	Zeile einfügen
F8	Rest der Zeile löschen
F9	Suchen/Ersetzen (u=Option für Rückwärts-Suche) (für Suchen keinen Ersatztext eingeben)
F10	Eingabe der ID-Kennung (nicht bei EDITOR.COM)
^Backspace	Nächste Zeile ab Cursorposition übernehmen
^Return	Rest dieser Zeile in die nächste Zeile
^T	Nächstes Wort in Kleinschrift wandeln
^U	Nächstes Wort in Großschrift wandeln

Steuerung der Zeicheneingabe und des Löschens:

^V	Insert-Modus umschalten (Anzeige: O oder I)
Einfg	Ein Leerzeichen einfügen
Entf	Zeichen unter dem Cursor löschen
Backspace	Zeichen vor dem Cursor löschen
^Y	Aktuelle Zeile entfernen
^N	Leerzeile einfügen

Speicherung:

ESC + R	Änderungen in diesem Screen rückgängig machen
ESC + Q	Ohne Änderungen Editor verlassen
ESC + S	File speichern, Editor verlassen

Befehle im Forth:

n edit	Screen n editieren
n append	n leere Screens am akuellen File anhängen
n1 n2 copy	Screen n1 auf n2 kopieren (überschreiben)
n list	Screen n auflisten
hlist	Erste Zeile aller Screens auflisten (mit Screennummer)

6. Abkürzungen

mcFORTH	Microcontroller-FORTH
TOS	Top of Stack = Oberster Datenstack-Eintrag
NOS	Next of Stack = Zweiter Datenstack-Eintrag
TOR	Top of Returnstack = Oberster Returnstack-Eintrag
PC	Programm Counter = Zeiger auf den als nächstes auszuführender Befehl
SP	Stack Pointer = Zeiger auf den obersten Datenstack-Eintrag
RP	Returnstack Pointer = Zeiger auf den obersten Returnstack-Eintrag

