Befehlswort	Adressierung	Bytezahl	Code Hex	Code Dez	Taktzyklen	Beeinflussung von Flaggen
AND	absolut	3	2D	45	4	N, Z
	0-page-abs	2	25	37	3	N, Z
	unmittelbar	2	29	41	2	N, Z
	abs.-X-indiz.	3	3D	61	4 *	N, Z
	abs.-Y-indiz.	3	39	57	4 *	N, Z
	indiz.-indir.	2	21	33	6	N, Z
	indir.-indiz.	2	31	49	5 *	N, Z
	0-page-X-indiz	2	35	53	4	N, Z
ORA	absolut	3	0D	13	4	N, Z
	0-page-abs.	2	05	05	3	N, Z
	unmittelbar	2	09	09	2	N, Z
	abs.-X-indiz.	3	1D	29	4 *	N, Z
	abs.-Y-indiz.	3	19	25	4 *	N, Z
	indiz.-indir.	2	01	01	6	N, Z
	indir.-indiz.	2	11	17	5 *	N, Z
	0-page-X-indiz	2	15	21	4	N, Z
EOR	absolut	3	4D	77	4	N, Z
	0-page abs.	2	45	69	3	N, Z
	unmittelbar	2	49	73	2	N, Z
	abs.-X-indiz.	3	5D	93	4 *	N, Z
	abs.-Y-indiz.	3	59	89	4 *	N, Z
	indiz.-indir.	2	41	65	6	N, Z
	indir.-indiz.	2	51	81	5 *	N, Z
	0-page-X-indiz	2	55	85	4	N, Z
ASL	»Akkumulator«	1	0A	10	2	N, Z, C
	absolut	3	0E	14	6	N, Z, C
	0-page-abs.	2	06	06	5	N, Z, C
	abs.-X-indiz.	3	1E	30	7	N, Z, C
	0-page-X-indiz	2	16	22	6	N, Z, C

*bedeutet: Bei seitenüberschreitenden Indizierungen muß noch ein Taktzyklus dazugerechnet werden.

Tabelle 4. Alles Wissenswerte der neuen Assembler-Befehle

```
600E    ASL
600F    ASL
6010    ASL
```

Nun erfolgt das Zusammenzählen beider Zwischenergebnisse. Dabei ist ja 8xZ noch im Akku.

```
6011    CLC
6012    ADC 6101
```

Damit ist die Aufgabe gelöst. Das Ergebnis steht im Akku und kann nun weiter verwendet werden.

Auf diese Weise kann man immer einen Mulitplikator in eine Zweierpotenz (2, 4, 8, 16,...) und weitere Summanden zerlegen. Dies ist allerdings eine zwar schnelle, aber doch recht eingeschränkte Art der Multiplikation. Außerdem haben Sie noch nicht erfahren, wohin man denn nun am besten mit BCC verzweigt, wenn die 8 Bits des Ergebnisses überlaufen.

Das alles aber, liebe Leser, erfahren Sie erst beim nächsten mal. Für heute war's lang und schwer genug! Wir werden dann einen eleganten Weg der 16-Bit-Multiplikation und Division erarbeiten. Und unser Programmprojekt soll natürlich endlich weiter gedeihen. Abschließend finden Sie in Tabelle 4 noch alles Wissenswerte zu den neuen Befehlen.

(Heimo Ponnath/gk)

Memory Map mit Wandervorschlägen

Weiter geht's mit der Erkundung der Speicherlandschaft. Die Adressen 57 bis 79 tragen zum Funktionieren eines Basic-Programms bei.

Heute sind eine Gruppe von Zeigern an der Reihe, die vom Betriebssystem des Computers während der Abarbeitung einzelner Programmzeilen verwendet werden.

Adresse 57 und 58 ($39 und $3A)

Nummer der laufenden Basic-Programmzeile

Diese Speicherzellen enthalten die Zeilennummer in Low-/High-Byte-Darstellung derjenigen Basic-Anweisung, welche gerade ausgeführt wird.
Ein kurzes Programm macht das deutlich:

```
10 PRINT "ZEILE 10", PEEK(57)+256*PEEK(58)
20 A=3:PRINT A,PEEK(57)+256*PEEK(58)
30 B=5:PRINT B,PEEK(57)+256*PEEK(58)
40 PRINT A*B,PEEK(57)+256*PEEK(58)
```

In jeder Zeile wird zuerst etwas gePRINTet, nämlich Text, Variable und ein Rechenresultat. Durch das Komma getrennt wird in der 2. Bildschirmhälfte (VC 20) beziehungsweise Bildschirmviertel (C 64) der Inhalt der Speicherzellen 57/58 ausgedruckt. Das Resultat zeigt in der Tat die jeweilige Zeilennummer an.

Die Basic-Befehle GOTO, GOSUB-RETURN, FOR-NEXT, END, STOP, CONT und die Betätigung der STOP-Taste während eines Programmlaufes verwenden alle den Inhalt dieser Speicherzellen, um entweder zu der laufenden Zeile zurückzufinden oder um die Unterbrechung mit BREAK IN... anzuzeigen. Auch die meisten Fehlermeldungen verwenden diese Zellen.

In vielen Basic-Erweiterungen und Programmierhilfen wird ein Befehl TRACE oder STEP angeboten, welcher ein schrittweises Abarbeiten eines Programms bei gleichzeitiger Anzeige der gerade aktiven Zeilennummer erlaubt. Dieses TRACE verwendet natürlich auch den Inhalt der Zellen 57/58.

Schließlich sei noch erwähnt, daß im direkten Modus, also bei direkt eingetippten Aktionen des Computers ohne Programmzeilen, in der Zelle 58 immer die Zahl 255 steht. Diejenigen Basic-Befehle, welche im direkten Modus nicht erlaubt sind (INPUT, GET, DEF) prüfen in Zelle 58, ob sie im direkten Modus oder während eines Programmlaufes aufgetreten sind.

Adresse 59 und 60 ($3B und $3C)

Zeilennummer der letzten Programmunterbrechung

Immer dann, wenn ein Programmablauf durch die Befehle END oder STOP oder aber mit der STOP-Taste abgebrochen wird, wird die Nummer der gerade ausgeführten Programmzeile nach 59/60 gebracht und bleibt dort so lange, bis eine neue Unterbrechung erfolgt.

Das läßt sich am besten mit der STOP-Taste und nachfolgendem CONT zeigen. Nehmen Sie bitte dazu das kleine Demo-Programm der Zellen 57/58 und ändern Sie alle PEEK-Adressen in 59/60 um. Fügen Sie außerdem noch eine Zeile 50 hinzu:

```
50 GOTO 10
```

Den dadurch erzeugten kontinuierlichen Lauf des Programms bremsen Sie dann mit der STOP-Taste und lassen ihn danach mit CONT weiterlaufen.

Auf der rechten Seite erscheint jetzt die Zeilennummer, bei der das Programm vorher unterbrochen worden ist.

Adresse 61 und 62 ($3D und $3E)

Zeiger auf die Adresse, ab welcher der Text der laufenden Basic-Zeile abgespeichert ist.

Die Abarbeitung der einzelnen Basic-Zeilen während eines Programmlaufs wird von einem kleinen Maschinencode-Programm, welches in den Speicherzellen 115 bis 138 steht (wir kommen noch dahin), gesteuert. In den Zellen 122/123 enthält es die Adresse, ab der die gerade bearbeitete Basic-Zeile gespeichert ist.

Sobald eine neue Basic-Zeile verarbeitet wird, holt das Betriebssystem diese Adresse aus 122/123 und speichert sie in den hier zur Diskussion stehenden Speicherzellen 61/62 ab, wie üblich als Low-/High-Byte.

Dasselbe geschieht bei jedem Befehl END, STOP, bei Fehlern mit dem Befehl INPUT und durch das Drücken der STOP-Taste. Der Befehl CONT hingegen schaut in 61/62 nach und bringt die darin befindliche Adresse zurück in die Speicherzellen 122/123 zur Fortsetzung

des Programms. Wenn aber in Zelle 62 inzwischen eine 0 steht — und das geschieht bei einem LOAD-Befehl, durch Programm-Abbruch mit Fehlermeldung und durch Eingabe neuer Basic-Zeilen beziehungsweise deren Veränderungen mit abschließender RETURN-Taste — dann wird der CONT-Befehl nicht ausgeführt.

Zur besseren Erklärung dieser in 61/62 als Zeiger stehenden Adresse einer Basic-Zeile möchte ich Sie an den dritten Teil dieses Kurses in Ausgabe 1/85 erinnern, in dem ich in einem separaten Texteinschub den Basic-Programmspeicher »sichtbar« gemacht habe, um die Wirkung der Verschiebung des Zeigers in den Zellen 43/44 zu demonstrieren.

Wir nehmen dazu bitte noch einmal das kleine Demo-Programm für die Adressen 57/58 oben her und ersetzen die PEEK-Werte durch 61 und 62. Das Ausdrucken des Inhalts von 61/62 legen wir aber an den Anfang jeder Zeile. Das Programm sieht dann so aus:

```
10 PRINT PEEK(61)+256*PEEK
(62),"ZEILE 10"
20 PRINT PEEK(61)+256*PEEK
(62),:A=3:PRINT A
30 PRINT PEEK(61)+256*PEEK
(62),:B=5:PRINT B
40 PRINT PEEK(61)+256*PEEK
(62),A*B
```

Nach RUN erhalten wir jetzt auf der linken Seite Zahlen, die den jeweiligen Basic-Speicher angeben, ab dem diese Zeile gespeichert ist. Wenn Sie ab diesen Adressen mit der gerade erwähnten Methode aus der Ausgabe 1/85 nachschauen, finden Sie genau die Zeilen des kleinen Demo-Programms wieder. Zur Anwendung dieses Zeigers kann ich wenig sagen. Ihn durch POKE zu verändern, geht in Basic nicht, weil das Betriebssystem die richtigen Werte immer neu eingibt. Man kann ihn allerdings abfragen.

Adresse 63 und 64 ($3F und $40)

Zeilennummer eines gerade laufenden DATA-Befehls

Diese Speicherzellen enthalten die Nummer der Basic-Zeile, in der gerade ein DATA-Befehl mit READ gelesen wird. Sobald in einer DATA-Zeile ein Fehler gefunden wird, kommt diese Zeilennummer aus 63/64 in die Speicherzellen 57/58, um in der Fehlermeldung die fehlerhafte DATA-Zeile und nicht die laufende READ-Zeile anzuzeigen. Auf diese Weise werden Syntax-Fehler in einer DATA-Zeile angezeigt. Um andere Fehler, wie zum Beispiel ein fehlendes Komma zwischen zwei DATA-Angaben anzuzeigen, können die Speicherzellen 63/64 eingesetzt werden.

In dem folgenden Programm wird in Zeile 20 geprüft, ob die DATA-Angaben größer als 255 sind. Da bei einem fehlenden Komma die beiden Zahlen als eine Zahl gelesen werden, wird dieser Fall erkannt und mit einem F versehen die Nummer der DATA-Zeile ausgedruckt, in der das Komma fehlt.

```
10 FOR X=1 TO 10:READ
A:PRINT A
20 IF A>255 THEN PRINT "F"
PEEK(63) + 256*PEEK(64)
30 NEXT X
40 DATA 10,20,30
50 DATA 40,50,60
60 DATA 70,80,90,100
```

Sie können jetzt in den DATA-Zeilen Kommafehler einbauen, die vom Programm angezeigt werden. Ein anderer häufiger Fehler, nämlich ein Komma am Ende einer DATA-Zeile, kann damit leider nicht erkannt werden. Aber vielleicht fällt Ihnen eine Prüfformel dazu ein.

Adresse 65 und 66 ($41 und $42)

Zeiger auf die Adresse, ab der die laufende DATA-Angabe gespeichert ist.

Diese Speicherzellen enthalten in der Low-/High-Byte-Darstellung die Adresse im Basic-Programmspeicher, ab welcher der READ-Befehl nach der nächsten DATA-Zeile sucht.

Zu Beginn eines Programms steht in 65/66 als Adresse der Beginn des Basic-Speichers, also derselbe Wert wie in den Speicherzellen 43/44. Der Befehl RESTORE setzt den Zeiger immer auf diesen Anfangswert zurück. Ein Demo-Programm zeigt uns das an (die Kommata sind wichtig für das Format der Darstellung auf dem Bildschirm!):

```
10 PRINT, PEEK(65)+256*PEEK
(66)
20 FOR X=1 TO 10:READ A
30 PRINT A,PEEK(65)+256*
PEEK(66)
40 NEXT X
50 DATA 10,20,30,40,50,60,70,80,
90,100
60 RESTORE
70 PRINT,PEEK(65)+256*PEEK
(66)
```

Durch Verändern dieses Zeigers in 65/66 kann die Reihenfolge, mit der DATA-Angaben gelesen werden, verändert werden, allerdings nur zeilenweise.

Wir brauchen dazu die oben beschriebenen Speicherzellen 61/62, deren jeweiligen Inhalt wir ja mit PEEK abfragen können. Wenn wir das vor jeder DATA-Zeile machen und diesen Wert einer Variablen zuweisen, haben wir die Adresse gespeichert, hinter welcher die DATA-Zeile kommt. Durch POKEn dieser Adressen in die Speicherzellen 65/66 vor einem READ-Befehl, wird diesem READ die nächste DATA-Zeile vorgegeben und wir können so die Reihenfolge der DATA-Zeilen ändern.

```
10 A1=PEEK(61):B1=PEEK(62)
20 DATA DAS IST DIE 1. ZEILE
30 A2=PEEK(61):B2=PEEK(62)
40 DATA DAS IST DIE 2.ZEILE
50 A3=PEEK(61):B3=PEEK(62)
60 DATA DAS IST DIE 3.ZEILE
70 POKE 65,A3:POKE 66,B3:
READ A$:PRINT A$
80 POKE 65,A1:POKE 66,B1:
READ A$:PRINT A$
90 POKE 65,A2:POKE 66,B2:
READ A$:PRINT A$
```

Mit den Zeilen 70 bis 90 werden für jede DATA-Zeile eigene READ-Anweisungen gegeben. Welche DATA-Zeile gelesen werden soll, wird durch die Variablen Ax und Bx (x=1,2,3) bestimmt, mit denen der Zeiger in 65/66 »verbogen« wird.

Adresse 67 und 68 ($43 und $44)

Zeiger auf die Adresse, aus welcher die Befehle INPUT, GET und READ die Zeichen/Zahlen holen

INPUT und GET verlangen Angaben, die per Tastatur eingegeben werden. Tastatur-Eingaben im direkten Modus, also, wenn kein Programm läuft, werden im Eingabe-Pufferspeicher des Editors (der Teil des Betriebssystems, welcher für die Zeilendarstellung auf dem Bildschirm verantwortlich ist) ab Speicherzelle 512 bis 600 zwischengespeichert.

Der Zeiger in 67/68 zeigt auf die jeweilige Adresse in diesem Eingabe-Pufferspeicher. Bei READ ist 67/68 identisch mit 65/66. Der Inhalt dieser Speicherzellen kann mit PEEK ausgelesen werden.

Adresse 69 und 70 ($45 und $46)

Name der gerade aufgerufenen Basic-Variablen

Wenn beim Ablauf eines Programms eine Variable auftaucht, muß ihr derzeitiger Wert im Variablen-Speicher gesucht werden. Während dieses Suchvorgangs wird der Name der Variablen in 69/70 zwischengespeichert. Die Form der Zwischenspeicherung ist dieselbe 2-Byte-Darstellung, wie im Variablenspeicher, beschrieben bei der Behandlung der Speicherzellen 45/46 im 4. Teil des Kurses (Ausgabe 2/85).

Adresse 71 und 72 ($47 und $48)

Zeiger auf die Adresse des Wertes der gerade aufgerufenen Basic-Variablen

Ähnlich wie bei 69/70 wird hier während des Anrufes einer Variablen durch ein Programm ein Wert zwischengespeichert, diesmal aber nicht der Name der Variablen, sondern der 2-Byte-Wert, welcher direkt hinter dem Variablennamen steht. Nähere Einzelheiten sind im Text der Speicherzellen 45/46 beschrieben (Teil 4, Ausgabe 2/85).

Davon ausgenommen sind selbstdefinierte Funktionen. Wie im nebenstehenden Textblock »Darstellung der Variablen einer selbstdefinierten Funktion« gezeigt ist, erscheinen diese ebenfalls im Variablenspeicher in einer Darstellung, welche den normalen Variablen sehr ähnlich ist.

Damit nun eine normale oder Feld-Variable denselben Namen haben kann wie eine Funktion, wird die oben genannte Zwischenspeicherung in 69/70 bei Funktionen unterdrückt.

Adresse 73 und 74 ($49 und $4A)

Zwischenspeicher für Variable einer FOR-NEXT-Schleife und für diverse Basic-Befehle

Die Adresse einer Schleifenvariablen wird zuerst hier gespeichert, bevor sie auf den Stapelspeicher ab Speicherzelle 256 ($100) gebracht wird. Die Funktion und Arbeitsweise des Stapelspeichers werden wir bei diesen Adressen behandeln. Etliche Basic-Befehle, wie LIST, WAIT, GET, INPUT, OPEN, CLOSE und andere, verwenden die Speicherzellen 73/74 für Zwischenspeicherungen. Diese Adressen sind für den Basic-Programmierer daher nicht verwendbar.

Adresse 75 und 76 ($4B und $4C)

Zwischenspeicher für Zeiger bei READ und mathematischen Operationen

Während der Auswertung eines mathematischen Ausdrucks durch die Routine FRMEVL des Basic-Übersetzers, wird der Platz des betroffenen mathematischen Operators in einer Tabelle, hier in 75/76, zwischengespeichert. Dieser Platz wird dabei als Abstand zum Beginn der Tabelle dargestellt. Außerdem verwendet der READ-Befehl diese Adressen als Zwischenspeicher für einen Programmzeiger. Die Speicherzeilen 75/76 sind in Basic nicht verwendbar.

Adresse 77 ($4D)

Hilfsspeicher für Vergleichs-Operationen

Die bei 75/76 schon erwähnte Auswertungs-Routine FRMEVL erzeugt in der Speicherzelle 77 einen Wert, der angibt, ob es sich bei einer Vergleichsoperation um den Fall »Kleiner als« (<), »gleich wie« (=) oder »größer als« (>) handelt. Diese Speicherzelle ist nur im Maschinencode erreichbar.

Adresse 78 und 79 ($4E und $4F)

Zeiger auf Adresse, ab welcher der Wert der Variablen einer selbstdefinierten Funktion gespeichert ist.

Basic erlaubt es bekanntlich, mit dem Befehl DEF selbst erfundene Funktionen zu definieren, welche die Form FN gefolgt von einem Variablennamen haben, zum Beispiel
DEF FNAA(X).

Im nebenstehenden Textblock »Darstellung von Variablen selbstdefinierter Funktionen« wird gezeigt beziehungsweise sichtbar gemacht, wie derartige Funktionen und ihre Variablen abgespeichert werden. Während der Definition einer Funktion steht in 78/79 die Adresse, ab welcher die Funktion und der Wert ihrer Variablen abgespeichert ist. Der Inhalt dieser Adressen ist identisch mit den Zeichen hinter dem Namen der Funktion (1. Gruppe im nebenstehenden Beispiel).

Nach der Ausführung der Funktion sieht in 78/79 allerdings die Adresse, ab welche der Zahlenwert der Funktion selbst abgespeichert ist. Er ist identisch mit den Zeichen der 2. Gruppe.

Diesen Zusammenhang können Sie überprüfen, indem Sie im Programm des Texteinschubes folgende Zeilen hinzufügen:
25 PRINT PEEK(78)+256*PEEK (79)
35 PRINT PEEK(78)+256*PEEK (79)

Nach RUN erhalten Sie zwei Adressen, die Sie mit direkter Eingabe abfragen:
FOR I=0 TO 4:PRINT PEEK (1.Adresse+I);:NEXT I:
FOR J=0 TO 4:PRINT PEEK (2.Adresse+J);:NEXT J

Sie werden sehen, daß der Inhalt der beiden Adressen genau die Werte der Zeichen 3 bis 7 der beiden Gruppen entspricht, allerdings im Bildschirmcode.

Das nächste Mal machen wir mit Speicherzellen 80 und 81 weiter und werden dann zu dem schon öfter erwähnten FLOATING POINT ACCUMULATOR kommen.

(Dr. H. Hauck/aa)

Nachtrag zu den sichtbaren Variablen

Frau Dr. Beyer aus Düren, auf deren Methode ich meine Sichtbarmachung der gespeicherten Variablen in den letzten beiden Folgen des Kurses aufgebaut habe, hat mir freundlicherweise zwei Verbesserungen mitgeteilt, die ich Ihnen nicht vorenthalten möchte.

Die erste Verbesserung bezieht sich auf die Methode für den C 64, dargestellt in Teil 4 auf Seite 151, 3. Zeile von oben.
Statt POKE 46,4:POKE 48,4
kann man auch schreiben: POKE 46,4: CLR

Wie bei den Speicherzellen 55/56 erklärt, setzt der Befehl CLR auch die Zeiger in 48 und 50 zurück, so daß der 2. POKE entfallen kann.

Die zweite Verbesserung bezieht sich auf die Methode für den VC 20 in Teil 5 in Ausgabe 3/85.

Im Textblock »Darstellung der normalen Variablen beim VC 20« habe ich erklärt, daß wir zur Sichtbarmachung der Zeichen eine Farbe in den Farbspeicher POKEn müssen, und das wurde dann auch im Schritt 3 ausgeführt.

Dieser Schritt 3 kann entfallen. Schritt 4 — Löschen des Bildschirms — kann jetzt aber mit der CLR-Taste gemacht werden, was natürlich viel schneller geht. Vor dem Schritt 5 wird der Cursor per SPACE-Taste über die ersten vier bis sechs Zeilen gejagt. Dadurch werden diese Zeilen sozusagen mit einer unsichtbaren Farbe gefüllt. Die weiteren Aktionen bleiben gleich und die Zeichen erscheinen oben am Bildschirm.

Das ist natürlich die eleganteste Methode.

Darstellung der Variablen einer selbstdefinierten Funktion

In den vorigen Folgen habe ich Ihnen gezeigt, wie im Programmspeicher abgelegte normale Variablen und Felder-Variablen sichtbar gemacht werden können. Damit konnten wir den Aufbau und die Darstellung der einzelnen Variablenarten studieren.

Heute will ich einen weiteren Variablentyp vorstellen, nämlich den der selbstdefinierten Funktionen.

Sie erinnern sich vielleicht, mit dem Basic-Befehl »DEF FN (Name)(Variable)« können wir komplizierte Funktionen selbst erfinden, definieren und später als »FN (Name)(Variable)« weiter verarbeiten. Diesen Typ wollen wir uns anschauen, wie er im Speicher steht.

Im Prinzip verwenden wir dieselben Methoden zur Sichtbarmachung, wie die letzten Male, verbessert natürlich mit den Vorschlägen des Texteinschubs 1.

Aber ein Unterschied kommt noch dazu. Der Befehl DEF kann leider nicht direkt eingegeben werden, sondern muß immer als Teil einer Programmzeile mit einer Zeilennummer versehen sein.

Deshalb schreiben wir zuerst ein kleines Programm zur Definition der Funktion plus Variable, bevor wir den Variablenspeicher mit dem Bildschirmspeicher zusammenlegen:

```
10 DEF FNAA(X)=3*SIN(X)+COS(X)
20 X=5
30 PRINT FNAA(X)
```

Die Funktion hat also den Namen »AA«. Bevor wir weitermachen, überprüfen Sie bitte mit RUN, ob alles stimmt. Nun wird der Speicher verschoben.

Für den C 64 gilt:

1. POKE 46,4:CLR
2. Bildschirm löschen mit CLR-Taste
3. Cursor auf die Mitte fahren
4. LIST (es erscheint das Programm)
5. auf den 2. Zeichensatz umschalten (mit C= und SHIFT-Taste)
6. RUN

Für den VC 20 (ohne Erweiterung) gilt:

Nur den Bildschirm auf 4096 zu verschieben, wie das letzte Mal, geht diesmal nicht, da wir ja für DEF ein kleines Programm schreiben müssen.

Also legen wir Bild- und Variablenspeicher ab Adresse 5120 (5120/256=20).

1. POKE 46,20:CLR
2. POKE 648,20
3. STOP/RESTORE-Tasten, bis Cursor wieder da ist.
4. Bildschirm löschen mit CLR-Taste
5. die ersten vier bis sechs Zeilen mit SPACE-Taste überfahren.
6. Cursor ein paar Zeilen nach unten
7. LIST (es erscheint das Programm)
8. mit Commodore- und SHIFT-Taste auf 2. Zeichensatz umschalten
9. RUN

Wir sehen jetzt oben zwei Gruppen mit je sieben Zeichen, wie üblich.

Die erste Gruppe stellt die Funktion FNAA(x) dar. Sie ist gekennzeichnet durch das invertierte erste Zeichen des Namens, während das zweite Zeichen normal erscheint.

Das dritte und vierte Zeichen gibt in Low-/High-Byte-Darstellung (im Bildschirmcode) die Adresse an, ab der die Funktion FNAA(x) im Programmspeicher abgelegt ist. Mit PEEK(3.Zeichen)+256*PEEK(4.Zeichen) kann das abgefragt werden.

Das fünfte und sechste Zeichen nennt die Adresse, an welcher der Zahlenwert der Funktions-Variablen X anfängt. Das siebente Zeichen schließlich ist das erste Zeichen der Funktion selbst (in unserem Beispiel die 3).

Die zweite Gruppe beschreibt die Variable X der Funktion. Die normale Darstellung der beiden ersten Zeichen, die den Namen darstellen, gibt uns an, daß es sich um eine Gleitkomma-Variable handelt, deren Wert als Mantisse und Exponent dargestellt ist.

Der Aufbau einer Funktion läßt sich also so zusammenfassen:

<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>Erstes</td><td>Zweites</td><td>Low-</td><td>High-</td><td>Low-</td><td>High-</td><td rowspan="3">1. Zeichen der Funktion</td></tr>
<tr><td colspan="2">Zeichen des Funktionsnamens
8</td><td colspan="2">Byte der Adresse, ab der die Funktion abgespeichert ist
9</td><td colspan="2">Byte der Adresse, ab dem der jeweilige Wert der Funktionsvariablen X abgespeichert ist
10</td></tr>
<tr><td>ASCII-Wert + 128</td><td>ASCII-Wert</td><td colspan="4"></td></tr>
</table>