Von Basic zu Assembler (Teil 13)

Eine Tabelle von Tabellen, das ist der Bereich des Basic-Speichers, der den Arrays vorbehalten ist. Dem Basic-Programmierer wohlvertraut, können diese indizierten Variablen auch dem Assemblerprogrammierer nützliche Dienste leisten.

Das gilt vor allem dann, wenn die Arrays im Basic-Rahmenprogramm eingerichtet und aus den Assemblerroutinen heraus angesteuert werden. Mehr noch als für die einfachen Variablen, deren Organisation wir in der letzten Folge kennengelernt haben, gilt es bei den indizierten Variablen, den Aufbau dieser Tabellen zu untersuchen, denn er ist wesentlich komplexer. Auch ist der Zugriff auf die einzelnen Elemente nicht so einfach zu realisieren. Wie also sind Arrays konstruiert? Wie kann man aus der Assemblerebene an die Elemente gelangen?

Der Header

Allen Arrays ist ein sogenannter Header (oder Kopf) gemeinsam. Bei den unterschiedlichen Feldtypen (darauf gehen wir später noch ein) unterscheiden sich lediglich die beiden ersten Byte dieses Kopfes, die den Namen des Feldes und eine Typkennung enthalten, welche mit den Typkennungen bei den einfachen Variablen identisch ist. Zur Erinnerung:

Bit 7 in	Byte 1	Byte 2
Integer-Kennung:	1	1
Fließkomma-Kennung:	0	0
String-Kennung:	0	1

Bild 1 zeigt Ihnen den grundlegenden Aufbau des Arraykopfes. Im einfachsten Fall ist so ein Header sieben Byte lang. Auf die beiden ersten Byte (Name und Kennung) folgt eine Längenangabe: In zwei Byte wird die Gesamtlänge des Arrays (inklusive Kopf) in der Form LSB/MSB festgehalten. Theoretisch könnte ein Feld also 65535 Byte lang werden. Das folgende fünfte Byte im Header gibt die Anzahl der Dimensionen an: Es wären somit — wiederum theoretisch, denn wer kann das noch überschauen! — bis zu 255 Dimensionen möglich. In jeweils zwei Byte — und zwar im etwas ungewöhnlichen Format MSB/LSB — finden wir danach Angaben über die Elementanzahl je einer Dimension. Zwei Dinge sind dabei noch zu beachten: Zum einen findet man hier immer eine um 1 höhere Elementanzahl als in der Dimensionierung. Ein Null-Element (in Programmiererkreisen gilt es als schick, nicht bei 1, sondern bei 0 mit dem Zählen anzufangen) wird mitgerechnet. Beispielsweise ergibt DIM A(4) hier die Zahl 5 (eben wegen der Reihenfolge A(0), A(1), A(2), A(3) und A(4)). Zum anderen aber gilt bei mehreren Dimensionen, daß für jede der genannten Dimensionen diese zwei Byte erscheinen, beginnend mit der zuletzt genannten Dimension. So ergibt beispielsweise DIM A(1,2,3) die folgende Belegung im Header vom 6.Byte an:

Byte 6: 0
Byte 7: 4 letzte genannte Dimension plus 1
Byte 8: 0
Byte 9: 3 vorletzte Dimension plus 1
Byte 10: 0
Byte 11: 2 erste Dimension plus 1

Die Länge des Kopfes ist also abhängig von der Anzahl der Dimensionen. Sie beträgt — wenn N diese Dimensionsanzahl symbolisiert — insgesamt:
Länge = 5 + 2*N

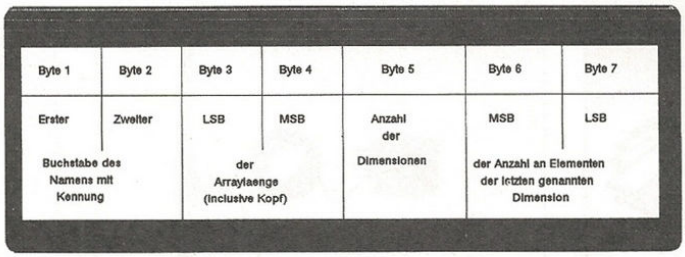

Bild 1. Dies ist ein Arraykopf (an das siebte Byte schließen sich weitere Elementanzahlen an)

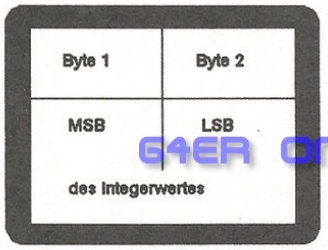

Bild 2. So sieht ein Element eines Integerfeldes aus ...

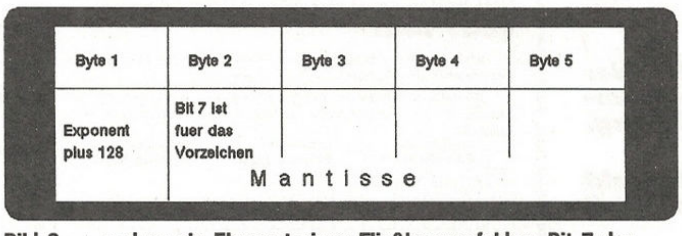

Bild 3. ... und so ein Element eines Fließkommafeldes. Bit 7 des zweiten Bytes dient als Vorzeichenkennung.

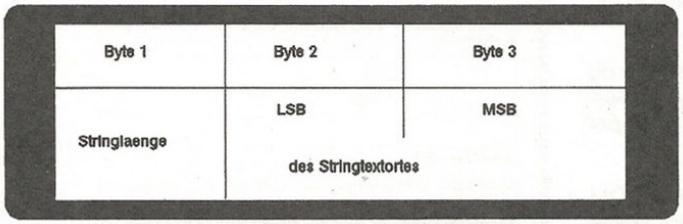

Bild 4. So ist ein Stringfeldelement aufgebaut

Zu jedem Kopf gehört auch ein Körper; hier sind das die Elemente des Arrays, die sich nahtlos anschließen.

Die Array-Elemente

Im Gegensatz zu den einfachen Variablen — dort belegt jede, gleich welchen Typs, immer sieben Byte — verbrauchen die Array-Elemente unterschiedlich viel Speicherplatz. Sehen wir uns zunächst das Element eines Integer-Arrays an: Der Zahlenwert ist hier im Zwei-Byte-Format festgehalten. Allerdings findet man auch hier wieder die ungewöhnliche Reihenfolge MSB/LSB. Bild 2 zeigt solch ein Integer-Element. Ein Element eines Fließkomma-Arrays ist im üblichen MFLPT-Format angeordnet. Erinnern Sie sich an die Folge 10 (64'er 1/87, Seite 149): Fünf Byte werden hier für eine Fließkommazahl benötigt, von denen das erste Byte dem Exponenten, die anderen vier der Mantisse zugeordnet sind. Das Bit 7 des zweiten Byte (also des ersten Mantissenbytes) dient als Vorzeichenkennung. Bild 3 zeigt Ihnen solch ein Fließkomma-Element.

Die Berechnung

Bleibt noch das String-Array-Element (Funktionen-Arrays — wie es die Einlagerung der Funktionen in die einfachen Variablen vermuten lassen würde — gibt es nicht). Solch ein String-Element besteht aus dem Stringdescriptor: Es ist daher drei Byte lang. Byte 1 gibt die Stringlänge, die Byte 2 und 3 den Stringtextort im Format LSB/MSB an. In Bild 4 finden Sie ein String-Array-Element.

Der Stringtext ist ebenso angeordnet wie bei den normalen Stringvariablen: Von der oberen Grenze des Basic-RAM abwärts. Auch hier findet sich die C 128-Besonderheit, daß im Anschluß an den Stringtext ein Zeiger auf den Stringdescriptor lokalisiert ist, der die Garbage-Collection beschleunigt (mehr darüber konnten Sie in der letzten Folge nachlesen.).

Jetzt können wir auch genau den Speicherplatzbedarf eines Feldes errechnen. Wenn — wie oben — N die Anzahl der genannten Dimensionen symbolisiert und D_N, D_{N-1}, ..., D_1 die Längen der einzelnen Dimensionen (also letzte Dimension mit der

Nummer N, vorletzte mit Nummer N-1, und so fort bis zur ersten Dimension mit der Nummer 1) sowie m den Platzbedarf eines Elementes (also m=2 für ein Integer-Element, m=5 für ein Fließkomma- und m=3 für ein String-Element) angibt, dann ergibt sich für die Länge des gesamten Array:

Länge = $5 + 2 * N + (D_N + 1) * (D_{N-1} + 1) * ... * (D_1 + 1) * m$

An einem Beispiel soll Ihnen das deutlich werden. Nehmen wir ein Fließkomma-Feld A(12,20,45), dann gilt:

Länge = $5 + 2 * 3 + (45 + 1) * (20 + 1) * (12 + 1) * 5 = 62801$

Hätten Sie das gedacht, daß solch ein Feld mit seinen 62801 Byte mit Sicherheit den Speicher sprengt? Als Integer-Array hätte es übrigens nur 25127 Byte verbraucht. So manchen Out of Memory-Error kann man sich ersparen, wenn man den Platzbedarf vor dem Programmlauf berechnet.

Wo befindet sich ein bestimmtes Element?

Um nun auf ein bestimmtes Element eines Arrays zugreifen zu können, muß man natürlich wissen, wo es sich befindet. Relativ einfach verhält sich das bei einem eindimensionalen Feld. Nehmen wir an, wir hätten durch DIM A(5) ein eindimensionales Fließkommazahlen-Array definiert und es dann mit Inhalt versehen. Ein Blick mittels eines Monitors in den Basicspeicher zeigt, daß die Elemente in der Reihenfolge

A(0), A(1), A(2), A(3), A(4), A(5)

angeordnet sind. Um also das Element A(i) zu finden, muß man das (i+1)ste Element ansteuern. Das erste Byte unseres Elementes ergibt sich aus der Addition von 7 (Länge des Headers) und i*5 (fünf Byte bilden ein Fließkomma-Element) zum Start des Arraykopfes.

Sehr viel komplexer wird das Wiederfinden eines Elementes schon, wenn wir ein zweidimensionales Feld betrachten. Nehmen wir an, ein String-Array wäre durch DIM A$(2,3) definiert und dann mit Elementen versehen worden, dann ist die übliche Darstellung (als 2,3-Matrix) in Bild 5 zu sehen.

Ebenfalls darin eingezeichnet ist die Reihenfolge, die man im Anschluß an den Header mittels eines Monitors beobachten kann:

A$(0,0); A$(1,0); A$(2,0); A$(0,1); A$(1,1); A$(2,1);...;A$(2,3)

Die Speicherung findet also Spalte für Spalte statt. Arbeiten wir mit den vorhin schon verwendeten Bezeichnungen (D_1 für die Elementanzahl in der ersten genannten Dimension und

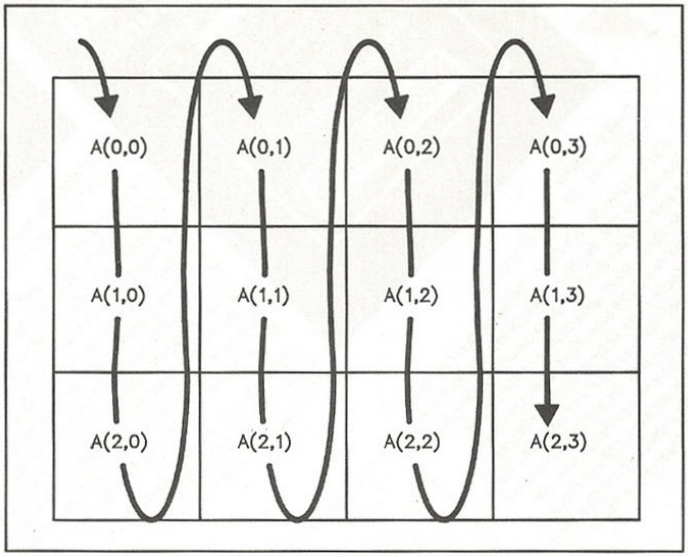

Bild 5. Die Elemente eines zweidimensionalen Feldes als (3,4)-Matrix und ihre Reihenfolge in der Feldtabelle

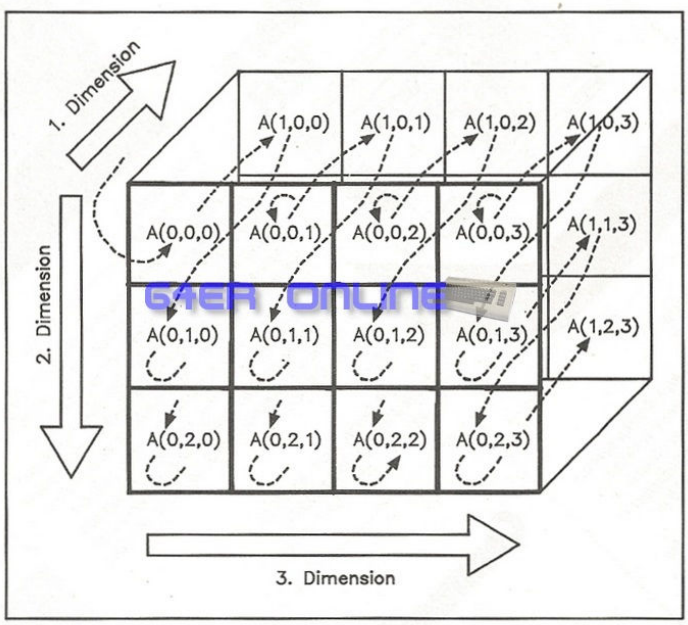

Bild 6. Die Elemente eines dreidimensionalen Feldes als (2,3,4)-Matrix. Die Reihenfolge ist durch die gestrichelten Pfeile angedeutet.

so fort), dann können wir ganz allgemein eine Formel für den Ort eines Elementes A$(i,j) bei vorher durch DIM A$($D_1$, D_2) dimensionierten Feldern angeben. Die Nummer n des Elementes A$(i,j) ist dann nämlich:

$n = (D_1 + 1) * j + i + 1$

Beispielsweise steht dann das Element A$(2,1) aus dem obigen Array (das wir durch DIM A$(2,3) definiert hatten) an der Stelle

$n = (2 + 1) * 1 + 2 + 1 = 6$

Dies können Sie schnell nachprüfen: Das sechste Element ist tatsächlich A$(2,1). Das erste Byte eines beliebigen Elementes mit den Indices i,j und der Elementlänge m (wie vorhin schon gehabt: m=2 beim Integer-, 5 beim Fließkomma- und 3 beim String-Array) ergibt sich aus der Formel:

Adresse 1. Byte = Headerstart + $9 + m * ((D_1 + 1) * j + i)$

Sie sehen: Es bedarf schon einiger Rechnereien, wenn man ein bestimmtes Element ansteuern möchte.

Die dritte Dimension

Durchaus nicht selten werden dreidimensionale Felder verwendet. Die Ansteuerung eines beliebigen Elementes ist hier noch etwas komplizierter. Gehen wir wieder von einem Beispiel aus: Durch DIM A(1,2,3) sei ein solches Array definiert worden und dann mit Fließkommazahlen gefüllt. Man kann es sich als einen Quader vorstellen, der die Tiefe von zwei hat (das entspräche der ersten genannten Dimension und betrifft die Indices (0,...,...) und (1,...,...)), die Höhe 3 (nämlich aus der zweiten Dimension mit den Indices (...,0,...), (...,1,...) und (...,2,...)) und die Breite 4 (hier ist es dann die dritte Dimension und die Indices (...,...,0), (...,...,1), (...,...,2) und (...,...,3)). Bild 6 zeigt Ihnen diese Gedankenstütze. Ebenfalls eingezeichnet als Linie ist die Reihenfolge der Elemente, die man mittels eines Monitors hinter dem Header entdeckt:

A(0,0,0); A(1,0,0); A(0,1,0); A(1,1,0); A(0,2,0); A(1,2,0); A(0,0,1); etc.

Es schält sich — wenn man die Elementanordnung durch die Dimensionen verfolgt — eine gewisse Gesetzmäßigkeit heraus: Offenbar wird die erste Dimension am häufigsten, die letzte am seltensten variiert. So findet man in den Elemente-Indices des obigen Beispiels die erste Dimension jedes zweite Mal, die zweite Dimension jedes dritte Mal und die dritte Dimension jedes siebte Mal variiert.

Die Nummer n eines Elementes A(i,j,k) nach dem Header eines Arrays, das durch DIM A(D_1,D_2,D_3) definiert und dann belegt worden ist, ergibt sich aus folgender Formel:

$n = (D_1 + 1) * (D_2 + 1) * k + (D_1 + 1) * j + i + 1$

Ein Beispiel (bezogen auf DIM A(1,2,3)): Die Nummer n des Elementes A(1,1,2) berechnet sich so:

$n = 2 * 3 * 2 + 2 * 1 + 1 + 1 = 16$

Das 16. Element heißt also A(1,1,2). Zählen Sie nach: Es stimmt! Damit ist es möglich, auch die Adresse des ersten Byte eines Elementes A(i,j,k) eines Feldes (das durch DIM A(D_1, D_2,D_3 definiert wurde) zu berechnen (m ist wieder die Länge eines Elementes):

Adresse = Header-Startadresse + $11 + m * ((D_1 + 1) * (D_2 + 1) * k + (D_1 + 1) * j + i)$

Höhere Dimensionen als 3 werden schon recht selten sein. Trotzdem: Es zeichnet sich — wenn man sich die Formeln für n der Reihe nach ansieht — eine gewisse Systematik ab. So liegt es nahe, daß die Nummer n eines Elementes A(i,j,k,l) des 4dimensionalen Feldes so zu berechnen ist:

$n = (D_1 + 1) * (D_2 + 1) * (D_3 + 1) * l + (D_1 + 1) * (D_2 + 1) * k + (D_1 + 1) * j + i + 1$

Wenn Sie Lust dazu haben, dann probieren Sie diese Formel doch einmal aus: Mit der Pointer-Funktion des C 128-Basic läßt sich das gut überprüfen.

Es gibt zwar eine Reihe von Basicinterpreter-Routinen, die speziell für den Umgang mit Arrays entwickelt wurden, sie sind aber meist nur sehr ungünstig

über ein Assemblerprogramm anzusteuern.

Ansteuern der Feld-Elemente

So übernehmen diese Routinen die Angaben aus dem Basic-Text und schieben anschließend die notwendigen Parameter in die verschiedensten Speicherstellen und auf den Stapel.

Oder, und dies ist eher noch komplizierter, die Routinen erfordern eine Unmenge teils sehr umständlicher Vorbereitungen. Parameter müssen hier an bestimmten Speicherstellen vorgegeben werden, eine Methode, die nicht nur Zeit kostet, sondern auch sehr fehleranfällig ist.

Es erscheint in fast jedem Fall sinnvoller, eine den eigenen Bedürfnissen angepaßte Routine selbst zu entwickeln, welche dann auch wesentlich komfortabler ausfallen kann als die vom Basic-Interpreter zur Verfügung gestellten.

Um Sie bei der Entwicklung einer solchen Routine zu unterstützen, möchten wir die Interpreter-Routinen an dieser Stelle kurz auflisten:

ISARY holt die Arrayparameter aus dem Basic-Text und legt sie auf den Stapel. Sie finden diese Routine beim C 64 ab $B1D1, beim C 128 ab $7CAB.

FNDARY sucht in der Arraytabelle nach dem Namen. Die Routine steht beim C 64 ab $B218, beim C 128 ab $7CF4. Der Name ist beim C 64 in $45/46, beim C 128 in $47/48 enthalten.

NOTFDD richtet ein neues Feld ein, wenn der Arrayname nicht gefunden wurde. Die Adressen der Routine: $B261 (C 64), $7D46 (C 128).

INLPN2 (C 64: $B30E, C 128: $7E00) sucht ein angegebenes Element und richtet den Zeiger VARPNT darauf. Diesen Zeiger finden Sie beim C 64 in $47/48, beim C 128 in $49/4A).

Eine sehr nützliche Routine ist aber UMULTD, eine 16-Bit-Multiplikationsroutine, die beim C 64 ab $B357 und beim C 128 ab $7E4B steht. UMULTD multipliziert eine Zahl in $28/29 (C 64) oder $72/73 (C 128). Das Ergebnis finden Sie in X/Y.

Entwickeln Sie einen Array-DUMP

Ein DUMP-Programm für die einfachen Variablen hatten wir bereits in der letzten Folge kennengelernt. Wir möchten Ihnen zum Abschluß dieser Folge — und damit gleichzeitig zum Abschluß der Serie — noch eine Aufgabe stellen, die Sie mit den bisher erworbenen Assembler-Kenntnissen selbst lösen können.

Wie wäre es, wenn Sie ein ähnliches Programm wie DUMP entwickeln würden, das die Felder und deren Inhalte auf dem Bildschirm ausgibt? Zu Ihrer Unterstützung hier einige Wegmarken, die Ihnen bei der Lösung des Problems sicherlich weiterhelfen:

Der Basicspeicher von der Adresse an, auf die ARYTAB zeigt (das ist der Vektor $2F/30 (beim C 64) oder $31/2 (beim C 128), der den Beginn der Arraytabellen markiert), bis zu der Adresse, auf die STREND weist (dieser Zeiger kennzeichnet das Ende der Arraytabellen und liegt bei $31/2 (C 64) oder $33/4 (C 128)) wird nach Arrayheadern durchforstet. Jeder gefundene Kopf ist dann zu untersuchen auf Name, Typ, Dimensionszahl und die Längen. Nun haben Sie mehrere Möglichkeiten: Sie können den Ausdruck der definierten Arrays — eventuell mit ihren Parametern — veranlassen. Sie können aber auch nach dieser ersten Option den Inhalt eines ausgesuchten Feldes ausdrucken oder aber die Ausgabe sämtlicher Arrayinhalte ermöglichen. Für die beiden zuletzt genannten Optionen dürfte das DUMP-Programm aus der letzten Folge einige Hilfsmittel bieten: Untersuchung des Typs und danach gesteuerte Bildschirmausgaben. Haben Sie dann eine funktionierende Problemlösung gefunden, wäre es interessant, diese im 64'er-Magazin wiederzufinden, denn solche Utilities sind noch recht rar. Die Lösung für den C 128 dürfte wegen der Bank-Probleme erheblich schwieriger zu finden sein als für den C 64.

Zukunftsperspektiven

Dieser Kurs ist hiermit beendet. Damit Ihre Assemblerfähigkeiten aber nicht einrosten, wird demnächst eine Serie starten, in der wir die Grafikprogrammierung in Assembler trainieren. Neben dem Training wird unser Ziel auch die Optimierung von Grafikoperationen sein. Möglichst schnelle Routinen und später auch einige interessante und sehr wirkungsvolle neue Basic-Befehle für Grafikoperationen werden wir gemeinsam entwickeln. (Heimo Ponnath/pd)