Der Basic-Interpreter des C 128

Um Ihnen Arbeit und schlaflose Nächte zu ersparen, zeigen wir, wie Sie beim C 128 die Basic-Interpreter-Routinen in eigene Maschinenprogramme einbauen können.

Zur Programmierung von Assemblerunterroutinen für Basic-Programme benötigt man oft die Basic-Interpreter-Routinen zur Parameterübergabe, zum Anlegen von Variablen und zum Rechnen mit Integer- und Realzahlen. In diesem Artikel wird gezeigt, wo sich diese Routinen beim C 128 befinden und worauf bei ihrem Einsatz zu achten ist.

Reine Maschinenprogramme kommen üblicherweise ohne jede Stringbehandlung aus. Bei einer Textverarbeitung zum Beispiel liegt der Text am Stück im Speicher und der Programmierer verwaltet selbst seine Zeiger auf die momentane Textstelle, auf die Seitenanfänge und so weiter.

Problematischer sind jedoch Unterprogramme in Maschinensprache, die zeitkritische Basic-Teile ersetzen sollen. In diesen Fällen soll oftmals das Ergebnis der Maschinenroutine in Form einer Variablen an das aufrufende Basic-Programm zurückübergeben werden. Wenn die Routine komfortabel aufzurufen sein soll, kann das Basic-Programm beim Aufruf mit »SYS...., (Variablenname)« angeben, in welcher Variablen das Ergebnis abgelegt werden soll.

In beiden Fällen, der Übergabe einer Variablen an das Maschinenprogramm und der Rückübergabe des Ergebnisses in Form einer Variablen, benötigen wir verschiedene Routinen des Basic-Interpreters. Es gibt eine Vielzahl von Anwendungen, in denen diese Routinen benötigt werden:

a) Sortierroutinen, die in der Lage sein sollten, beliebige Arraytypen zu sortieren und denen der Name des gewünschten Arrays übergeben werden muß. Weiterhin müssen Sortierroutinen über noch zu besprechende Routinen des Basic-Interpreters Anfang und Ende dieses Arrays erkennen können.

b) die beliebten INPUT-Routinen, die den eingebauten Befehl »INPUT X$« ersetzen sollen. Komfortablen INPUT-Routinen sollen zum Beispiel folgende Parameter übergeben werden können: Spalte und Zeile, an der die Eingabe beginnt; Länge des Eingabefeldes; als Eingabe zugelassene Zeichen; Zeichen, die die Eingabe beenden; String, in dem die Eingabe an das Basic übergeben werden soll. Eine INPUT-Routine darf die Eingabe selbst natürlich nicht an beliebiger Stelle im Speicher ablegen, sondern muß einen String anlegen, der die Eingabe enthält und den das Basic-Programm weiterverarbeiten kann.

c) Suchroutinen, die ein beliebiges Array (Strings oder numerische Variablen) nach einer vorgegebenen Zeichenkette oder Zahl durchsuchen sollen, zum Beispiel die im Speicher gehaltene Indexdatei in einer Dateiverwaltung. In diesem Fall muß die Assemblerroutine auf das entsprechende Array zugreifen (womöglich sogar erst ab einem vorgegebenen Index, der den Start der Routine bestimmen soll), jedes Arrayfeld mit dem Suchkriterium vergleichen und nach Abschluß der Suche eine numerische Variable anlegen, der das Ergebnis mitgeteilt wird, zum Beispiel »x% = 5« (das fünfte Arrayelement entspricht dem übergebenen Suchkriterium).

Da ich selbst kürzlich vor der Aufgabe stand, solche und andere Routinen vom C 64 auf den C 128 umzuschreiben und daher gezwungen war, die besprochenen Basic-Interpreter-Routinen des C 128 zu untersuchen, bietet es sich an, mein frisch erworbenes Wissen weiterzugeben. Es dürfte manchem Assemblerprogrammierer schlaflose Nächte ersparen, in denen er sich anhand des eingebauten Monitors durch das Basic-ROM des C 128 »hindurchwühlt«.

Bevor wir Strings oder Variablen anlegen können, ist es nötig, alle Arten von Parametern, die das aufrufende Basic-Programm übergibt, aus dem Basic-Text herauszulesen. Der C 128 bietet hier einiges mehr an Komfort als der C 64, da der SYS-Befehl bereits für die Übergabe von Parametern vorgesehen ist. Er besitzt folgendes Format:

SYS (Startadresse),(Akku),(X-Register),(Y-Register),(Status)

Beim Aufruf können daher bereits Ein-Byte-Werte übergeben werden, mit denen der Akku und die verschiedenen Register geladen werden. Im weiteren Verlauf dieses Artikels wird das Statusregister jedoch nicht verwendet werden. Bedenken Sie, daß die Abfrage des Statusregisters nur unter »Verrenkungen« möglich ist, indem die verschiedenen Flaggen getestet werden, wobei bei einigen Bits ein Test nicht möglich ist. Ich werde mich daher auf die Übergabe von maximal drei Werten im Akku, im X- und im Y-Register beschränken.

Ein Beispiel: Geben Sie mit dem eingebauten Monitor ein:

```
A 0B00 BRK
```

Geben Sie nun im Direktmodus ein:

```
SYS DEC("0B00"),1,2,3
```

Nach dem BRK wird der Monitor die übergebenen Werte in den Registern anzeigen:

Akku: 1; X-Register: 2; Y-Register: 3

Sie sind keineswegs gezwungen, beim Aufruf immer alle Parameter anzugeben. Wollen Sie zum Beispiel nur im Akku den Wert 20 und im X-Register eine 53 übergeben, genügt der Aufruf mit:

```
SYS DEC("0B00"),20,53
```

Soll ein Register »übergangen«, also nicht mit einem definierten Wert geladen werden, können Sie den entsprechenden Parameter einfach weglassen.

Beispiel: Akku mit 15 und Y-Register mit 87 laden:

```
SYS DEC("0B00"),15,,87
```

Für viele Anwendungen ist diese Form der Übergabe jedoch unzureichend, zum Beispiel für die Übergabe eines Zwei-Byte-Wertes oder einer Stringvariablen. Auch dann, wenn zwar ausschließlich Ein-Byte-Werte übergeben werden sollen, die Anzahl der Parameter jedoch größer drei ist, benötigen wir Routinen, die uns weitere Parameter aus dem Basic-Text holen. Die erste und einfachste dieser Routinen ist GETBYT, die einen Ein-Byte-Wert aus dem Basic-Text holt und im X-Register übergibt.

Bevor ich nun die Routine GETBYT beschreibe, geben Sie bitte die folgende Programmzeile ein, speichern Sie das Programm, und starten Sie es:

```
10 POKE DEC("2E"),DEC("20"):POKE
DEC("2000"),0:POKE DEC("30"),DEC
("20"):CLR:NEW
```

Gehen Sie nun in den eingebauten Monitor und geben Sie folgende Zeilen ein:

Initialisierung

```
a 00b00  ad 00 ff lda $ff00
a 00b03  48       pha
a 00b04  a9 00    lda #$00
a 00b06  8d 00 ff sta $ff00
a 00b09  ad 06 d5 lda $d506
a 00b0c  09 06    ora #$06
a 00b0e  8d 06 d5 sta $d506
a 00b11  68       pla
a 00b12  8d 00 ff sta $ff00
a 00b15  60       rts
```

Speichern Sie auch dieses Programm und rufen Sie es mit »SYS DEC("0B00")« auf.

Ein Problem bei der Programmierung des C 128 in Assembler ist das Bankswitching. Die Routinen des Basic-Interpreters schalten ständig zwischen Bank 0 (Basic-Text) und Bank 1 (Variablenbank) um. Unsere Programme legen wir in Bank 0. Wenn wir nun eine Routine des Basic-Interpreters aufrufen, nach deren Ausführung auf Bank 1 umgeschaltet ist, ist unser in Bank 0 liegendes Programm für den Prozessor nicht existent. Die Folge ist zwangsläufig ein Absturz.

Das Initialisierungsprogramm hat die Aufgabe, den Bereich $0000 - $1FFF als gemeinsamen Speicherbereich zu definieren. Egal, ob bei der Rückkehr aus einer Interpreter-Routine Bank 0 oder Bank 1 eingeschaltet ist. Unser Programm existiert immer in der jeweiligen Bank und kann daher weiter ausgeführt werden.

Gewöhnen Sie sich auf alle Fälle an, diese Initialisierung (sowohl das Basic- als auch das Maschinenprogramm) nach jedem Einschalten des Rechners oder Reset als erstes zu laden und aufzurufen. Sie vermeiden dadurch Abstürze, die bei Betrachtung der aufgerufenen Assemblerroutine selbst völlig unerklärlich sind und nur durch das Bankswitching zustande kamen.

CHKKOM ($795C)

Bevor wir, wie beabsichtigt, einen zusätzlichen Parameter aus dem Basic-Text holen, benötigen wir die Routine CHKKOM. Sie liest das nächste Zeichen des Basic-Textes (ausgehend von dem momentanen Stand des Programmzeigers der CHRGET/CHRGOT-Routine) und inkrementiert den Programmzeiger. Bei dem gelesenen Zeichen wird überprüft, ob es sich um ein Komma handelt. Wenn ja, ist alles in Ordnung und es erfolgt ein RTS. Handelte es sich um ein anderes Zeichen, wird in die Routine zur Fehlerausgabe gesprungen und ein »SYNTAX ERROR« ausgegeben. Da es üblich ist, zur Trennung mehrerer Übergabeparameter – ebenso wie die SYS-Routine – Kommata zu benutzen, bietet es sich an, diese Routine vor dem Aufruf von GETBYT zu verwenden.

1. Ein Komma einlesen:

```
a 0b20  20 5c 79 jmp $795c
```

Nach dem Aufruf mit »SYS DEC("0B20"),0,0,0,0,« oder »SYS DEC("0B20"),,,,,« erhalten wir die Meldung READY. Der SYS-Befehl hat alle vier Parameter gelesen, anschließend erfolgte der Sprung zu unserem eigenen Assemblerprogramm, das ein zusätzliches Komma las und danach die Rückkehr ins Basic. Ein zusätzlicher Befehl RTS entfällt, da sich dieser am Ende des Unterprogramms CHKKOM befindet. Beachten Sie bitte, daß bei der Verwendung der Interpreter-Routinen zur Übergabe von Parametern zuerst alle vier möglichen Parameter, die der SYS-Befehl einlesen kann, angegeben werden müssen. Würden wir unser Programm zum Beispiel mit »SYS DEC("0B20"),« aufrufen, wäre ein SYNTAX ERROR die Folge, da der SYS-Befehl nach dem Komma den ersten Standardparameter (Akku) lesen will und nicht findet.

2. Drei Kommas einlesen:

```
a 00b20  20 5c 79 jsr $795c
a 00b23  20 5c 79 jsr $795c
a 00b26  4c 5c 79 jmp $795c
```

Aufruf mit
»SYS DEC("0B20"),0,0,0,0,,,« oder
»SYS DEC("0B20"),,,,,,,«.

Jeder andere Aufruf, der von dieser Form (Übergabe der Standardparameter und der einzulesenden Anzahl Kommata) abweicht, führt zu einem SYNTAX ERROR. Die Routine CHKKOM bietet daher eine hervorragende Absicherung gegen Fehleingaben beim Aufruf einer Assemblerroutine.

GETBYT ($87F4 ($87F1))

GETBYT verwendet auch der SYS-Befehl zur Übergabe der Standardparameter. Diese Routine holt einen Ein-Byte-Wert aus dem Basic-Text und übergibt ihn im X-Register. Liegt der gelesene Wert außerhalb der zulässigen Grenzen (0-255), wird ein ILLEGAL QUANTITY ERROR ausgegeben.

Vier Ein-Byte-Werte übergeben:

```
a 00b30  8d 00 0d sta $0d00
a 00b33  8e 01 0d stx $0d01
a 00b36  8c 02 0d sty $0d02
a 00b39  20 5c 79 jsr $795c
a 00b3c  20 f4 87 jsr $87f4
a 00b3f  00       brk
```

Nach dem Aufruf mit »SYS DEC("0B30"),1,2,3,0,4« werden die vier Standardparameter übergeben. Bevor wir nun die zusätzlichen Werte einlesen können, müssen wir zuvor die bereits übergebenen Werte retten. Im Beispiel werden die Registerinhalte nach $0D00 bis $0D02 übertragen. Anschließend wird mit CHKKOM das folgende Komma und mit GETBYT der zusätzliche Ein-Byte-Wert gelesen. Nach dem BRK sehen wir, daß das X-Register den Wert vier besitzt. Wenn Sie sich den Bereich ab $0D00 mit dem Monitor anschauen, werden Sie feststellen, daß der Reihe nach die Werte eins, zwei und drei abgelegt wurden.

Die Routine GETBYT besitzt zwei Einsprungadressen, $87F4 und $87F1. Bisher wurde der Einsprung ab Adresse $87F4 benutzt. Im Unterschied hierzu wird zu Beginn von $87F1 zuerst ein Unterprogrammaufruf von CHRGET durchgeführt, das heißt, das Zeichen, auf das der Programmzeiger weist, wird gelesen und inkrementiert.

Diese Einsprungadresse ist für »faule« Programmierer geeignet. Der Aufruf von CHRGET liest das Trennzeichen zwischen dem vorhergehenden Wert und dem einzulesenden Wert, so daß sich der Aufruf von CHKKOM erübrigt. Da dieses Trennzeichen jedoch in keiner Weise überprüft wird, verzichte ich persönlich auf diese Art des Einsprungs, da ich davon ausgehe, daß ein Benutzer, der sich beim Trennzeichen vertippt hat, eventuell auch den nachfolgenden Ein-Byte-Wert falsch eingegeben hat.

Für alle »Bytesparer« stelle ich dennoch ein kleines Beispielprogramm vor, das diesen Einsprung zum Lesen von fünf Parametern verwendet:

```
a 00b40  8d 00 0d sta $0d00
a 00b43  8e 01 0d stx $0d01
a 00b46  8c 02 0d sty $0d02
a 00b49  20 f1 87 jsr $87f1
a 00b4c  8e 03 0d stx $0d03
a 00b4f  20 f1 87 jsr $87f1
a 00b52  00       brk
```

Nach dem Aufruf mit »SYS DEC("0B40"),1,2,3,0,4,5« befindet sich im X-Register der zuletzt übergebene Wert fünf und die zuvor übergebenen Parameter in $0D00 bis $0D03.

Dieses Spiel kann beliebig weiter fortgesetzt werden. Der Zahl der Ein-Byte-Parameter, die an ein Assemblerprogramm übergeben werden können, sind keine Grenzen gesetzt. Sie müssen nur darauf achten, nach jeder Übergabe den übergebenen Wert zu retten, bevor der nächste Aufruf von CHKKOM oder GETBYT erfolgt.

Der nächste Schritt besteht darin, einen Zwei-Byte-Wert einzulesen. Eine solche Übergabe ist zum Beispiel für Grafikprogramme erforderlich, da die Auflösung des Grafikbildschirms in X-Richtung höher als 255 ist, oder für die Arbeit mit Arrays, bei denen das Basic-Programm der Routine übergeben soll, ab welchem Element (dessen Index höher als 255 sein kann) zum Beispiel eine Suche beginnen soll.

Die Übergabe eines Zwei-Byte-Wertes könnte noch vollzogen werden, indem das Basic-Programm vor dem Aufruf eine Zerlegung in Low- und Highbyte vornimmt.

Beispiel:

```
10 X=1030
20 H=INT(X/256)
30 L=X-H*256
40 SYS DEC("0B00"),L,H
```

Wie Sie an diesem Beispiel sehen, verarbeitet die Parameterübergabe beim SYS-Befehl auch Variablen und nicht nur direkt eingesetzte Zahlen. Problematisch bei dieser Form des Aufrufes ist jedoch, daß eine Assemblerroutine in einem Basic-Programm üblicherweise an zeitkritischen Stellen eingesetzt wird. Soll diese Routine nun zum Beispiel innerhalb einer Schleife ständig (mit variierenden Übergabeparametern) aufgerufen werden, leidet die Geschwindigkeit sehr unter dieser umständlichen Vorbereitung des Aufrufs.

Es ist daher vorteilhaft, auch beim Einlesen eines Zwei-Byte-Wertes die entsprechenden Routinen des Basic-Interpreters zu verwenden. Daß diese vorhanden sein müssen, beweist bereits der POKE-Befehl (POKE XXXX,YY), der ja ebenfalls einen Zwei-Byte-Wert als Parameter verwendet.

FRNUM ($77D7)

Die Routine FRNUM holt einen beliebigen numerischen Ausdruck aus dem Basic-Text und wertet ihn aus. Das Ergebnis wird im Fließkommaformat im Fließkommaakku Nummer 1 hinterlegt (FAC #1).

ADRFOR ($8815)

ADRFOR wandelt eine im FAC #1 abgelegte Fließkommazahl in das Adreßformat um, das heißt, in eine 16-Bit- oder Zwei-Byte-Zahl. Diese Adresse wird sowohl in den Registern (Y=Low-Byte/Akku=High-Byte) als auch an die Adressen $16, $17 ($16=Low-Byte/$17=High-Byte) übergeben.

Wenn wir beide Routinen nacheinander aufrufen (nachdem mit CHKKOM das Trennzeichen zur vorherigen Variablen gelesen wurde), erhalten wir sowohl in den Registern als auch in den genannten Speicherzellen einen Zwei-Byte-Wert:

```
a 00b60  20 5c 79 jsr $795c
a 00b63  20 d7 77 jsr $77d7
a 00b66  20 15 88 jsr $8815
a 00b69  00       brk
```

Rufen Sie diese Routine mit »SYS DEC("0B60"),0,0,0,0,1026« oder mit »SYS DEC("0B60"),,,,,1026« auf. Zuerst wird das Komma eingelesen, danach mit FRNUM der numerische Ausdruck geholt und im FAC #1 abgelegt, anschließend mit ADRFOR der FAC #1-Inhalt ins Adreßformat gewandelt. Im Akku befindet sich nun eine vier und im Y-Register der Wert zwei (hex: $0402; dezimal:1026). Die gleichen Werte finden Sie in $16, $17 in der Form: Lowbyte, Highbyte.

GETADR ($880F)

Die Routinen FRNUM und ADRFOR können Sie selbstverständlich auch einzeln für andere Zwecke als das Einlesen eines Zwei-Byte-Wertes einsetzen. Für diesen Zweck ist es empfehlenswerter, die Routine $GETADR einzusetzen. Diese Routine ruft nacheinander CHKKOM, FRNUM und ADRFOR auf. Das Einlesen des Zwei-Byte-Wertes aus dem vorherigen Beispiel vereinfacht sich dadurch erheblich. Beachten Sie, daß das Einlesen des Trennkommas mit CHKKOM entfällt, da auch diese Routine von GETADR aufgerufen wird:

```
a 00b70  20 0f 88 jsr $880f
a 00b73  00       brk
```

Rufen Sie diese Routine mit »SYS DEC("0B70"),0,0,0,0,1026« auf. Sie erhalten die gleichen Werte wie zuvor sowohl in den Registern als auch in den Speicherzellen $16, $17.

ADRBYT ($8803)

ADRBYT ruft nacheinander die Routinen GETADR, CHKKOM und GETBYT auf. Diese Routine liest daher in einem Zuge sowohl einen Zwei-Byte-Wert als auch einen folgenden Ein-Byte-Wert ein und ist damit hervorragend geeignet, um zum Beispiel Grafikroutinen aufzurufen und ihnen sowohl eine X-Koordinate (16-Bit) als auch eine Y-Koordinate (8-Bit) zu übergeben:

```
a 00b80  20 5c 79 jsr $795c
a 00b83  20 03 88 jsr $8803
a 00b86  00       brk
```

Aufruf:
»SYS DEC("0B80"),0,0,0,0,513,15«

Nach diesem Aufruf befindet sich im X-Register der von GETBYT übergebene Ein-Byte-Wert 15 (=hex $0F). Den Zwei-Byte-Wert 513 suchen Sie leider vergeblich im Y-Register und im Akku. Durch den nachfolgenden Aufruf von CHKKOM und GETBYT werden diese Register überschrieben, so daß diese Adresse diesmal nur in den Speicherzellen $16, $17 (Low/High) zu finden ist.

Da ich es wirklich beeindruckend finde, wieviel Vorbereitungen erspart werden können, indem man die für die jeweilige Aufgabe optimal geeignete Routine aufruft, stelle ich nun noch ein kleines Programm vor, das die Übergabe einer Adresse (16-Bit) und eines Bytes nur mit Hilfe der grundlegenden Routinen CHKKOM, FRNUM, ADRFOR und GETBYT vollzieht:

```
a 0b30 jsr $795c    ;CHKKOM
a 0b33 jsr $77d7    ;FRNUM
a 0b36 jsr $8815    ;ADRFOR
a 0b39 jsr $795c    ;CHKKOM
a 0b3c jsr $87f4    ;GETBYT
a 0b3f brk
```

Wenn Sie diese Serie von Aufrufen mit einem einzigen Aufruf von ADRBYT vergleichen, stellen Sie fest, daß es sich wirklich lohnt, über die verschiedenen Routinen zur Übergabe von numerischen Variablen Bescheid zu wissen, um die jeweils geeignetste einsetzen zu können.

GETPOS ($7AAF)

Zum Abschluß der verschiedenen Übergaberoutinen stelle ich nun noch GETPOS vor, die ganz sicher eine der wichtigsten Routinen des Basic-Interpreters ist. Die bisher vorgestellten Routinen bezogen sich nur auf numerische Varia-

blen und übergaben im Basic-Text stehende Werte und Werte numerischer Variablen.

Im Unterschied zu diesen Routinen kann GETPOS sowohl in Verbindung mit numerischen Variablen (Integer oder Fließkomma) als auch in Verbindung mit Stringvariablen eingesetzt werden. GETPOS holt keinen Wert, sondern liefert einen Zeiger auf eine Variable, genauer einen Zeiger auf die Variablentabelle.

Jede Variable ist in der Variablentabelle beschrieben, die normalerweise beim C 128 ab $0400 beginnt und in Bank 1 liegt, ebenso wie die Variablen selbst. Jeder Eintrag in dieser Tabelle ist bei nicht dimensionierten Variablen sieben Byte lang. Die ersten zwei Byte sind die beiden Zeichen des Variablennamens, wobei bei Integerzahlen in beiden Byte Bit sieben, bei Stringvariablen Bit sieben nur im zweiten Byte des Namens gesetzt ist, und Realzahlen durch ein in beiden Bytes gelöschtes Bit sieben erkannt werden. Die restlichen fünf Byte (3 bis 7) haben folgende Bedeutung:
a) Real: (5 Byte Fließkommadarstellung)
b) Int: (High-Byte)(Low-Byte)(3 Byte ungenutzt)
c) String:(1 Byte Stringlänge)(2 Byte Stringadresse)(2 Byte ungenutzt)

Beachten Sie unbedingt, daß eine Integerzahl im – für Assemblerprogrammierer ungewohnten – Format: »High-Byte/Low-Byte« abgelegt wird.

Dimensionierte Arrays werden folgendermaßen verwaltet: Der Variablenname (2 Byte) befindet sich nur am Arraybeginn. Danach folgen zwei Byte, die den belegten Speicherplatz angeben, ein weiteres Byte, das die Dimensionierung enthält und zwei Byte mit der Anzahl der Arrayelemente. Erst anschließend folgt das Array:
a) Real: (5 Byte Fließkommadarstellung)
b) Int: (High-Byte)(Low-Byte)
c) String:(1 Byte Stringlänge)(2 Byte Stringadresse)

GETPOS liefert in jedem dieser Fälle einen Zeiger auf das erste Byte des Descriptors der jeweiligen Variablen, das heißt, bei nicht dimensionierten Variablen einen Zeiger auf das erste Byte des Variablennamens, bei dimensionierten Variablen entweder – numerische Variablen – einen Zeiger auf das erste Byte der Zahl selbst, oder aber – Stringvariablen – einen Zeiger auf die Länge der jeweiligen Stringvariablen.

Dieser Zeiger wird sowohl an die Register (Akku = Low/Y = High), als auch an die Speicherzellen $49, $4A (Low-/Highbyte) übergeben.

Da GETPOS alle Variablentypen verarbeitet, genügt ein winziges Beispielprogramm, um den Zugriff auf numerische und Stringvariablen, sowohl dimensioniert als auch nicht dimensioniert, zu demonstrieren:

```
a 00b90  20 5c 79 jsr $795c
a 00b93  20 af 7a jsr $7aaf
a 00b96  00       brk
```

Die folgenden Beispiele können Sie im Direktmodus eingeben:

a) nicht dimensionierte Variablen

1.NEW:A% = 10:SYS DEC("0B90"),0,0,0,0,A%
Gehen Sie in den Monitor und geben Sie ein:
M 49. Ergebnis: 00049 02 20 .....
Der Zeiger $49, $4A weist somit auf $2002 (in Bank 1, der Variablenbank). Geben Sie ein:
M 12002. Ergebnis: 12002 00 0A .....
In $12002 befindet sich das High-Byte (Null), in $12003 das Lowbyte ($0A = 10) der Integerzahl 10.
2.NEW:A = 10:SYS DEC("0B90"),0,0,0,0,A
M 49. Ergebnis: 00049 02 20 .....
M 12002. Ergebnis: 12002 84 20 00 00 00 (= Fließkommadarstellung von 10).
3.NEW:A$ = "BASICINTERPRETER":SYS DEC("0B90"),0,0,0,0,A$
M 49. Ergebnis: 00049 02 20 .....
M 12002. Ergebnis: 12002 10 EE FE .....

Im Unterschied zu numerischen Variablen zeigt der Zeiger $49, $4A bei Stringvariablen nicht unmittelbar auf die Variable selbst, sondern auf das erste Descriptorbyte (Stringlänge). Die beiden folgenden Bytes geben die Adresse des Strings an. Geben Sie daher ein:
M 1FEEE. Ergebnis: 1FEEE 42 41 53 ..... (= ASCII-Darstellung von 'BASICINTERPRETER').

b) dimensionierte Variablen

1.NEW:A%(2)= 10:SYS DEC("0B90"), 0,0,0,0,A%(2)
M 49. Ergebnis: 00049 0B 20.....
M 1200B. Ergebnis:
1200B 00 0A.....

In $12002 befindet sich das High-Byte (Null), in $12003 das Lowbyte ($0A = 10) der Integerzahl 10.

2.NEW:A(2) = 10:SYS DEC ("0B90"),0,0,0,0,A(2)
M 49. Ergebnis: 00049 11 20.....
M 12011. Ergebnis: 12011 84 20 00 00 00 (= Fließkommadarstellung von 10).

3.NEW:A$(2) = "C 128":SYS DEC ("0B90"),0,0,0,0,A$(2)
M 49. Ergebnis: 00049 0D 20.....
M 1200D. Ergebnis:
1200D 05 F9 FE.....
M 1FEF9. Ergebnis:
1FEF9 43 2D 31.....
(= ASCII-Darstellung von »C 128«)

Wie Sie sehen, ist diese Routine außerordentlich universell einsetzbar. Merken Sie sich folgende Punkte, die sowohl für einzelne als auch für Arrayvariablen gelten: Der Zeiger in den Speicherzellen $49, $4A zeigt bei numerischen Variablen auf das erste Byte der Zahl selbst, bei Stringvariablen auf den Längendescriptor des Strings, hinter dem in Form von Lowbyte/Highbyte die Stringadresse folgt.

Weiterhin ist eventuell von Interesse für Sie, daß beim Aufruf dieser Routine in den Speicherzellen $47, $48 der Variablenname übergeben wird und daß GETPOS eine Variable anlegt, wenn sie noch nicht definiert wurde. Daher besteht keine Veranlassung, darauf zu achten, daß beim Aufruf einer eigenen Assemblerroutine, die GETPOS verwendet wird. Ein Absturz ist dank des Anlegens der Variablen nicht möglich!

Einen Nachteil besitzt dagegen diese Routine gegenüber ihrem Äquivalent auf dem C 64: GETPOS liefert auf dem C 64 einen zweiten Zeiger auf den Anfang eines Arrays bei Verwendung dimensionierter Variablen. Dieser Zeiger kann zum Beispiel bei Suchroutinen sehr nützlich sein, die ein Arrayfeld nach einem bestimmten Wert oder String durchsuchen müssen. Eine solche Routine muß wissen, wann das Arrayende erreicht wird und die Suche beendet werden muß. Dieses Arrayende kann über den erwähnten Zeiger ermittelt werden, indem der Arraydescriptor (Speicherplatz, den das Array benötigt), der in der Arraybeschreibung steht, zum Arraybeginn addiert wird (= Ende des Arrays).

GETPOS auf dem C 128 besitzt diesen Zeiger zwar ebenfalls, er wird jedoch vor der Rückkehr aus GETPOS durch andere Werte überschrieben. Um dennoch auf die Descriptoren eines Arrays zuzugreifen, kann ein Trick verwendet werden. Beispiel: Eine Routine soll ein Stringarray ab Element A$(53) durchsuchen. Das Arrayende kann durch folgende Form des Aufrufs ermittelt werden:

SYS DEC("0BA0"),0,0,0,0,A$(0) ,A$(53)

Die aufgerufene Assemblerroutine lautet:

```
a 00ba0  20 5c 79 jsr $795c
;CHKKOM
a 00ba3  20 af 7a jsr $7aaf
;GETPOS von A$(0)
a 00ba6  38       sec
a 00ba7  e9 07    sbc #$07
;7 von Low-Byte subtrahieren
a 00ba9  85 fb    sta $fb
;Low-Byte speichern
a 00bab  b0 01    bcs $0bae
;Unterlauf?
a 00bad  88       dey
;High-Byte dekrementieren
a 00bae  84 fc    sty $fc
;High-Byte speichern
a 00bb0  20 5c 79 jsr $795c
;CHKKOM
a 00bb3  20 af 7a jsr $7aaf
;GETPOS von A$(53)
a 00bb6  00       brk
```

Diese Routine holt zuerst die Adresse der Descriptoren des ersten Arrayelementes, A$(0). Der Zeiger $49, $4A weist somit auf den Längendescriptor dieses ersten Elementes. Wird nun von diesem Zeiger – der sich zugleich in den Registern befindet (Akku = Low-byte/Y-Reg. = High-Byte) – der Wert sieben subtrahiert, weist der Zeiger auf das erste Byte der Arraybeschreibung, die wie erwähnt sieben Byte lang ist.

Dieser neue Zeiger wird in $FB, $FC gespeichert und anschließend GETPOS auf das eigentlich interessierende Startelement A$(53) angewendet, um dessen Länge und Adresse zu holen.

Da $FB, $FC nun auf das erste Element der Arraybeschreibung weist, kann durch Addition des vom Array benötigten Speicherplatzes zum Arrayanfang dessen Ende + 1, das heißt, der Beginn des nächsten Arrays, ermittelt werden. Da die Speicherbelegung in Byte zwei und drei der Arraybeschreibung enthalten ist (Low-Byte/High-Byte), kann mit indirekt indizierter Adressierung darauf zugegriffen werden.

Prinzip:

```
LDY #2
LDA ($FB),Y   ;Zugriff auf
    Low-Byte des benötigten
    Speicherplatzes
INY
LDA ($FB),Y   ;Zugriff auf High-
    Byte des benötigten
    Speicherplatzes
```

Wir kommen nun zum Anlegen von Variablen aller Art. Zum Anlegen müssen wir jedoch auf die Variablentabelle und – bei Stringvariablen – auf den Stringstack zugreifen. Sollten Sie anderer Meinung sein, da die Variablentabelle normalerweise ab $0400 beginnt und der Bereich $0000 bis $1FFF von unserem Initialisierungsprogramm als gemeinsamer Bereich von Bank 0 und Bank 1 definiert wurde, so schauen Sie sich bitte mit dem Monitor die Speicherzellen $2F,$2E an, den Zeiger auf den Anfang der Variablentabelle. Dieser Zeiger weist auf $2000, ein Ergebnis des Basic-Initialisierungs-Programms. Damit liegt der Variablenbereich außerhalb des gemeinsamen Bereichs. Beim Zugriff auf die Tabelle muß zuvor auf Bank 1 umgeschaltet werden.

Mit dem eingebauten Monitor ist ein solcher Zugriff sehr einfach, da vor der eigentlichen Adresse nur eine Eins (Bank 1) angegeben werden muß und der Monitor dann selbsttätig auf die gewünschte Bank schaltet. In unseren eigenen Programmen müssen wir jedoch »per Hand« zwischen Bank 0 und Bank 1 umschalten. Ich stelle nun ein erweitertes Initialisierungs-Programm vor, in dem zwei Umschaltroutinen vorhanden sind:

```
a 00b00  ad 00 ff lda $ff00
a 00b03  48       pha
a 00b04  a9 00    lda #$00
a 00b06  8d 00 ff sta $ff00
a 00b09  ad 06 d5 lda $d506
a 00b0c  09 06    ora #$06
a 00b0e  8d 06 d5 sta $d506
a 00b11  68       pla
a 00b12  8d 00 ff sta $ff00
a 00b15  60       rts
a 00b16  08       php
a 00b17  48       pha
a 00b18  a9 00    lda #$00
a 00b1a  8d 00 ff sta $ff00
a 00b1d  68       pla
a 00b1e  28       plp
a 00b1f  60       rts
a 00b20  08       php
a 00b21  48       pha
a 00b22  a9 7f    lda #$7f
a 00b24  8d 00 ff sta $ff00
a 00b27  68       pla
a 00b28  28       plp
a 00b29  60       rts
```

Dieses erweiterte Initialisierungs-Programm enthält eine Routine zum Umschalten auf Bank 0 ab $0B16 und eine zweite zum Umschalten auf Bank 1 ab $0B20. In beiden Fällen wird der Inhalt der beiden Register, die beim Ablauf verändert werden, Akku und Status, gerettet und nach beendetem Umschalten zurückgeholt. Beide Routinen können Sie daher in Ihren eigenen Programmen an beliebigen Stellen aufrufen, ohne sich Gedanken darüber zu machen, welche momentan benötigten Registerinhalte verändert werden könnten und daher vor dem Aufruf gerettet werden müssen.

Wichtig ist dieses Retten der Register dann, wenn Sie zum Beispiel einen Wert aus der Variablentabelle holen wollen:

```
jsr bank 1
lda vartab,x
cmp vartab,y
jsr bank 0
```

Nach dem Zurückschalten auf Bank 0 sind alle Flaggen, die der CMP-Befehl gesetzt hat, unverändert vorhanden.

Integervariable anlegen

Zum Anlegen einer Integervariablen genügen die bereits besprochenen Routinen des Basic-Interpreters. Wir gehen davon aus, daß unser Programm zum Beispiel nach dem Durchsuchen eines Arrays den Index eines Elementes, das dem von Basic übergebenen Suchkriterium entspricht, in einer Variablen an das Basic zurückübergeben soll:

```
a 00b30  20 5c 79 jsr $795c
a 00b33  20 af 7a jsr $7aaf
a 00b36  20 20 0b jsr $0b20
a 00b39  a0 00    ldy #$00
a 00b3b  a9 02    lda #$02
a 00b3d  91 49    sta ($49),y
a 00b3f  c8       iny
a 00b40  a9 04    lda #$04
a 00b42  91 49    sta ($49),y
a 00b44  20 16 0b jsr $0b16
a 00b47  60       rts
```

Rufen Sie dieses Programm mit »SYS DEC("0B30"),0,0,0,0,I%« auf. Denken Sie jedoch daran, zuvor das erweiterte Initialisierungs-Programm einzugeben!

Unsere Routine liest nun das Komma ein und ruft GETPOS auf, die in den Speicherzellen $49, $4A einen Zeiger auf die – neu angelegte – Variable liefert, das heißt, auf das erste Byte der Integerzahl selbst, die momentan natürlich Null ist. Nachdem auf Bank 1, die Variablenbank, umgeschaltet wurde, schreiben wir in das erste Byte dieser Zahl eine $02, in das zweite Byte den Wert $04. Zum Abschluß schalten wir wieder auf Bank 0 um.

Geben Sie bitte »PRINT I%« ein, nachdem die Routine bearbeitet wurde. Jedem, der nun erwartet, daß der Wert 1026 ausgegeben wird, empfehle ich, den kleinen Exkurs über die Variablenabspeicherung nachzulesen. Integervariablen werden in der ungewohnten Form »High/Low« abgelegt. Die als Wert eingetragenen Zahlen $02, $04 entsprechen somit dem Wert

$0204 beziehungsweise der ausgegebenen Dezimalzahl 516.

Anlegen einer Realvariablen

Wenn wir einem aufrufenden Basic-Programm ein Realergebnis mitteilen wollen, legen wir auch diese Realvariable analog dem eben geschilderten Ablauf an: Wir holen die Variablenadresse, schalten auf Bank 1 um, legen die Realzahl im Fließkommaformat ab der angezeigten Adresse ab und schalten wieder auf Bank 0:

```
a 00b50  20 5c 79 jsr $795c
a 00b53  20 af 7a jsr $7aaf
a 00b56  20 20 0b jsr $0b20
a 00b59  a0 00    ldy #$00
a 00b5b  a9 84    lda #$84
a 00b5d  91 49    sta ($49),y
a 00b5f  c8       iny
a 00b60  a9 20    lda #$20
a 00b62  91 49    sta ($49),y
a 00b64  a9 00    lda #$00
a 00b66  c8       iny
a 00b67  91 49    sta ($49),y
a 00b69  c8       iny
a 00b6a  91 49    sta ($49),y
a 00b6c  c8       iny
a 00b6d  91 49    sta ($49),y
a 00b6f  20 16 0b jsr $0b16
a 00b72  60       rts
```

Starten Sie dieses Programm mit »SYS DEC("0B50"),0,0,0,0,R«. Geben Sie anschließend ein: »PRINT R«. Wir erhalten den Dezimalwert 10 oder in Fließkommadarstellung: »84 20 00 00 00«.

Wie Sie sehen, ist es völlig unproblematisch, numerische Variablen auf dem C128 anzulegen. Der entscheidende Punkt ist das Umschalten auf Bank 1, bevor auf die Variablentabelle zugegriffen wird. Im Unterschied zum C64 sollten Sie bei der Fehlersuche in Ihren Programmen daher zuerst nachforschen, ob diese Umschaltung an den nötigen Stellen vorgenommen wird, da ansonsten auch das durchdachteste Programm auf dem C128 nicht laufen wird.

Die Datentyp-Flags ($0F/$10)

Bevor wir uns nun dem Anlegen von Strings zuwenden, will ich zuvor zwei wichtige Zeropageadressen erwähnen, $0F und $10. Wenn wir die Routine GETPOS aufrufen, erhalten wir in diesen beiden Speicherzellen Informationen über die Art der gelesenen Variablen.

Wie es bereits im Handbuch steht, enthält $0F den Wert $00 bei numerischen Variablen und $80 bei Stringvariablen.

$10 enthält im Falle einer Integervariablen den Wert $80 und $00 im Fall einer Realvariablen.

Wie wir beide Speicherzellen sinnvoll verwenden können, will ich an einem Beispiel demonstrieren, an einer kleinen Routine, die die gleiche Funktion erfüllt wie der Befehl SWAP in manch anderem Basic-Interpreter, nämlich die Vertauschung zweier Variablen.

Ein solcher Befehl ist vor allem in Sortierroutinen, in denen ständig Variablen miteinander vertauscht werden müssen, äußerst nützlich. Der Befehl:

SWAP A$(2),A$(7) ist zum einen schneller als die herkömmliche Methode:

S$=A$(2):A$(2)=A$(7):A$(7)=S$

und der zweite, fast noch wesentlichere Vorteil besteht darin, daß bei der Vertauschung von Stringvariablen die Strings nicht am unteren Ende des Stringstacks neu angelegt werden müssen, sondern daß die Vertauschung der Stringdescriptoren genügt.

Die Arbeit einer Sortierroutine, die einen derartigen Befehl verwendet, wird daher weitaus seltener durch eine nötige Garbage Collection unterbrochen, so daß sich die Sortierzeit erheblich verkürzt.

Bevor Sie die folgenden Programmzeilen eingeben, vergewissern Sie sich bitte, daß sich das neuere Initialisierungs-Programm im Speicher befindet. Unsere Routine wird die darin enthaltenen Unterprogramme zum Umschalten der Speicherbänke benötigen.

Zeiger auf die Variablen holen

```
a 00b30  20 5c 79 jsr $795c
a 00b33  20 af 7a jsr $7aaf
a 00b36  a5 49    lda $49
a 00b38  a6 4a    ldx $4a
a 00b3a  85 fb    sta $fb
a 00b3c  86 fc    stx $fc
a 00b3e  20 5c 79 jsr $795c
a 00b41  20 af 7a jsr $7aaf
```

Der erste Programmteil bietet nur bereits Bekanntes. Nach dem Aufruf mit zum Beispiel »SYS DEC("0B30"),0,0,0,0,A$,B$« wird ein Zeiger auf A$ mit Hilfe von GETPOS geholt. Dieser Zeiger wird nach $FB, $FC kopiert, da anschließend ein Zeiger auf B$ geholt wird und sich in den Speicherzellen $49, $4A nun der Zeiger auf die Descriptoren dieser Variablen befinden.

Typflags auswerten

Die Auswertung verläuft nach dem abgebildeten Schema: Wenn $0F gleich Null ist, handelt es sich um eine numerische Variable und es wird geprüft, ob $10 ebenfalls den Wert Null enthält. Wenn ja, wurde eine Realvariable gelesen und wir laden das Y-Register mit $04, dem Startwert für eine Schleife, die von Y bis inklusive Null läuft und die einzelnen Bytes beider Variablen (beziehungsweise bei Strings deren Descriptoren) vertauscht.

War $0F gleich Null, $10 jedoch ungleich Null, wird das Y-Register mit $01 geladen (Integervariable).

Wurde hingegen eine Stringvariable gelesen, laden wir das Y-Register mit dem Wert $02.

```
        lda $0f
        bne string
        lda $10
        bne integer
        ldy #$04
        bne weiter
integer ldy #$01
        bne weiter
string  ldy #$02
weiter  jsr $0b20
```

Soweit der prinzipielle Ablauf, den ich jedoch in ein kürzeres Programm umgesetzt habe:

```
a 00b44  a5 0f    lda $0f
a 00b46  d0 0a    bne $0b52
a 00b48  a5 10    lda $10
a 00b4a  d0 03    bne $0b4f
a 00b4c  a0 04    ldy #$04
a 00b4e  2c a0 01 bit $01a0
a 00b51  2c a0 02 bit $02a0
a 00b54  20 20 0b jsr $0b20
```

Dieser Programmteil wird für erfahrene 6502/8502 - Programmierer auf Anhieb verständlich sein, für weniger erfahrene jedoch wahrscheinlich ein unlösbares Rätsel darstellen.

Zuerst schauen wir uns den Inhalt des Flags $0F an. Wenn es gleich Null ist, das heißt eine numerische Variable übergeben wird, wird $10 getestet. Handelte es sich um eine Realvariable, wird das Y-Register unmittelbar mit $04 geladen.

Das auf den Befehl LDY #04 folgende Byte $2C ist der Prozessorcode für den BIT-Befehl mit 16-Bit-Adressierung. Die nachfolgenden beiden Bytes werden somit als Adresse angesehen.

Der BIT-Befehl hat keinerlei Auswirkungen auf das Y-Register. Da anschließend wiederum Byte $2C folgt, werden auch die folgenden beiden Bytes als Adresse eines BIT-Befehls interpretiert.

Seltsamerweise springt unser Programm mitten in den BIT-Befehl »hinein«, wenn eine Integervariable übergeben wurde (a 00b4a d0 03 bne $0b4f). Geben Sie bitte mit dem Monitor ein:

d 0b4f. Er interpretiert die auf den BIT-Code folgenden Bytes als: »LDY #$01«, wodurch das Y-Register - wie im Falle einer Integervariablen gewünscht - mit einer Eins geladen wird.

Soviel zu den Einsatzmöglichkeiten des BIT-Befehls. Daß diese

Anwendung üblich ist, zeigt der ständige Einsatz dieser Methode im ROM des C 128.

Vertauschen der Variablen (-zeiger)

```
a 00b57  b1 49    lda ($49),y
a 00b59  aa       tax
a 00b5a  b1 fb    lda ($fb),y
a 00b5c  91 49    sta ($49),y
a 00b5e  8a       txa
a 00b5f  91 fb    sta ($fb),y
a 00b61  88       dey
a 00b62  10 f3    bpl $0b57
a 00b64  20 16 0b jsr $0b16
a 00b67  60       rts
```

Das Vertauschen der beiden Variablen geschieht analog zu den erwähnten Basic-Befehlen, jedoch byteweise. Als Zwischenspeicher dient das X-Register.

Beachten Sie, daß die Routine nicht überprüft, ob beide übergebenen Variablen vom gleichen Typ sind, was Grundbedingung für eine sinnvolle Vertauschung ist.

Komplettes Listing von »SWAP«

```
a 00b30  20 5c 79 jsr $795c
a 00b33  20 af 7a jsr $7aaf
a 00b36  a5 49    lda $49
a 00b38  a6 4a    ldx $4a
a 00b3a  85 fb    sta $fb
a 00b3c  86 fc    stx $fc
a 00b3e  20 5c 79 jsr $795c
a 00b41  20 af 7a jsr $7aaf
a 00b44  a5 0f    lda $0f
a 00b46  d0 0a    bne $0b52
a 00b48  a5 10    lda $10
a 00b4a  d0 03    bne $0b4f
a 00b4c  a0 04    ldy #$04
a 00b4e  2c a0 01 bit $01a0
a 00b51  2c a0 02 bit $02a0
a 00b54  20 20 0b jsr $0b20
a 00b57  b1 49    lda ($49),y
a 00b59  aa       tax
a 00b5a  b1 fb    lda ($fb),y
a 00b5c  91 49    sta ($49),y
a 00b5e  8a       txa
a 00b5f  91 fb    sta ($fb),y
a 00b61  88       dey
a 00b62  10 f3    bpl $0b57
a 00b64  20 16 0b jsr $0b16
a 00b67  60       rts
```

Zum Testen des Programms geben Sie bitte ein:

```
A$="C128":B$="C64"
SYS DEC("0B30"),,,,,A$,B$
PRINT A$,B$
```

und Sie erhalten auf dem Bildschirm die vertauschten Stringinhalte:

C64 C128

Anlegen einer Stringvariablen

Die bisherigen Programmbeispiele dieses Artikels wurden so einfach wie möglich gehalten und dürften keine Verständnisschwierigkeiten hervorrufen. Das Anlegen einer Stringvariablen werde ich nun an einem komplexeren Beispielprogramm erläutern, einer INPUT-Routine.

Bei der Entwicklung dieser Routine werden viele der beschriebenen Interpreter-Routinen benötigt. Außerdem werden weitere Routinen und spezielle Zeropageadressen beschrieben, ohne die das Anlegen eines Strings nicht möglich ist.

Das Niveau dieses Artikels wird durch das komplexere Programm zwar deutlich höher werden, dafür haben Sie jedoch am Ende dieses Abschnitts eine lauffähige Eingaberoutine in Ihren Händen, die Sie jederzeit ausbauen können und kennen das Anlegen von Strings auf dem C 128, das in den verschiedensten Routinen von Bedeutung ist. Einsatzbeispiel sind wie erwähnt Sortierroutinen, INPUT-Routinen; eine Vielzahl weiterer Routinen, die Strings bearbeiten, sind denkbar. Eine nette Aufgabe könnte zum Beispiel eine INPUT#-Routine sein, die Strings mit bis zu 255 beliebigen Zeichen einliest und damit die Beschränkungen des INPUT#-Befehls aufhebt. Die INPUT-Routine soll folgendes leisten:

1. Vor Beginn der Eingabe den Cursor auf eine angegebene Position setzen.

2. Während der Eingabe nur Zeichen zulassen, die in einem als Parameter übergebenen String enthalten sind.

3. Prinzipiell - soweit nicht durch Punkt 2 verhindert - alle Zeichen zulassen, auch solche, die beim INPUT-Befehl zu einem »REDO FROM START«-Error führen.

4. Die Eingabe vom Bildschirm übernehmen und in einem vom Basic-Programm frei wählbaren String an dieses übergeben.

Da als Eingabezeichen nur die in dem übergebenen String enthaltenen zugelassen werden sollen, ist es nötig, diesen String um:

CHR$(29) = Cusor-right,
CHR$(157) = Cursor-left,
CHR$(20) = Delete und
CHR$(148) = Insert zu erweitern, um die gleichen Editiermöglichkeiten wie beim INPUT-Befehl zu erhalten.

Beispiel:

```
Z$="1234567890"+CHR$(29)+
CHR$(157)+CHR$(20)+CHR$(148)
```

läßt Ziffern und Editiertasten als Eingabe zu.

Da dieses Programm deutlich umfangreicher und damit schwieriger zu verstehen ist als die bisherigen Programmbeispiele, werde ich den Programmablauf diesmal anhand eines Assemblerlistings erklären. Zum Abtippen mit dem eingebauten Monitor finden Sie im Anschluß an die Erläuterungen das Programm in der gewohnten disassemblierten Form.

Zeropageadressen

```
**ZEROPAGEADRESSEN*
STREND = $35    ;ZEIGER AUF ENDE
                 DES STRINGSTACKS
ZEIGER = $49    ;ZEIGER AUF
                 DESCRIPTOREN
SPALTE = $EC    ;CURSORSPALTE
ZEILPOI = $E0   ;ZEIGER:SPALTE 0
                 DER MOMENTANEN
                 CURSORZEILE
```

STREND zeigt immer das momentane Ende des Stringstacks, der bekanntlich von »oben« ($FFFF) nach unten wächst. ZEIGER ist der von GETPOS übergebene Zeiger auf die Descriptoren einer aus dem Basic-Text gelesenen Variablen. In SPALTE ist – bei Benutzung der Betriebssystem-Routinen zur Zeichenausgabe – die momentane Cursorspalte enthalten, und ZEIGER weist auf den Anfang der aktuellen Zeile, in der sich der Cursor befindet. Die aktuelle Cursorposition ergibt sich somit aus ZEIGER+SPALTE.

Eigene Variablen

```
**EIGENE VARIABLEN**
BANK0    = $0B16
BANK1    = $0B20
ZLAENGE  = $FB  ;LAENGE DES
                 STRINGS MIT DEN
                 ZUGEL.ZEICHEN
POSIT    = $FC  ;ZEIGER:POS. DES
                 STRINGS MIT DEN
                 ZUGEL.ZEICHEN
STARTPOS = $A3  ;ZEIGER:
                 STARTPOSITION DER
                 EINGABE
LAENGE   = $A5  ;EINGABEL.MAX.
```

BANK0 und BANK1 sind die Einsprungadressen für unsere Unterprogramme zur Bankumschaltung. In ZLAENGE und in POSIT, POSIT+1 werden die Descriptoren des Strings gespeichert, in dem die als Eingabe zulässigen Zeichen abgelegt wurden. STARTPOS, STARTPOS+1 ist die beim Aufruf übergebene Bildschirmposition, bei der die Eingabe beginnen soll. In LAENGE legen wir die ebenfalls vom Basic-Programm übergebene maximale Eingabelänge ab.

Verwendete Kernelroutinen

Zu diesen Routinen dürfte keine weitere Erläuterung nötig sein, da sie den entsprechenden C64-Rou-

tinen äquivalent sind und sich dank der Sprungtabelle auch die Einsprungadressen nicht verändern.

```
**KERNELROUTINEN**
BSOUT = $FFD2     ;ZEICHEN IM AKKU
                  AUSGEBEN
BASIN = $FFCF     ;ZEICHEN AUS
LOG. DATEI HOLEN
CHKKOM = $795C    ;AUF KOMMA
                  PRUEFEN
GETPOS = $7AAF    ;DESCRIPTOREN
                  POS. EINER
                  VARIABLEN
                  HOLEN
STRRES = $9299    ;SPEICHERPLATZ
                  RESERVIEREN
GETIN = $FFE4     ;ZEICHEN HOLEN
SETCRS = $FFF0    ;CURSOR SETZEN
OPEN = $FFC0      ;DATEI OEFFNEN
CLOSE = $FFC3     ;DATEI
                  SCHLIESSEN
CHKIN = $FFC6     ;EINGABE AUF
                  LOG. DATEI LEGEN
CLRCH = $FFCC     ;EINGABE AUF
                  TASTATUR LEGEN
SETFLS = $FFBA    ;FILEPARAMETER
                  SETZEN
SETNAM = $FFBD    ;FILENAME SETZEN
 .BA $0D00        ;PROGRAMM-
                  START
```

Parameter retten/Cursor setzen

Aufgerufen werden soll unsere Routine mit:

```
SYS DEC("0D00"),Länge,Zeile,
Spalte,0,Zeichenstring,Übergabe-
string
```

Beispiel:

```
SYS DEC("0D00"),20,10,5,0,Z$,A$
```

bedeutet: die maximale Eingabelänge beträgt 20 Zeichen, die Eingabe soll ab Spalte 5 in Zeile 10 beginnen, nur die in Z$ enthaltenen Zeichen werden als Eingabe zugelassen, und die Eingabe selbst soll dem Basic-Programm in A$ übergeben werden (vergessen Sie bei der Definition von Z$ die Editiertasten nicht).

Die in den Registern übergebenen Parameter Länge/Zeile/Spalte müssen für eine spätere Verwendung gerettet werden:

```
**PARAMETER RETTEN*
STA LAENGE
TXA
PHA
TYA
PHA
```

Der Cursor wird nun auf die Zeile und Spalte des Eingabebeginns gesetzt (X = Zeile/Y = Spalte):

```
**CURSOR SETZEN*
CLC
JSR SETCRS ;CURSOR SETZEN
```

Descriptoren holen

Der nächste Schritt besteht darin, die Descriptoren für den String mit den als Eingabe zulässigen Zeichen zu holen:

```
**ZEIGER:DESCR.ZEICHEN$ HOLEN**
       JSR CHKKOM
       JSR GETPOS
       JSR BANK1
       LDY #2
HOLDES LDA (ZEIGER),Y
       STA ZLAENGE,Y
       DEY
       BPL HOLDES
       JSR BANK0
```

Zuerst wird das Komma hinter den Standardparametern gelesen, anschließend GETPOS aufgerufen, die uns in den Speicherzellen $49, $4A (=ZEIGER) einen Zeiger auf den nachfolgenden String (im Beispielaufruf Z$) liefert beziehungsweise diesen String zugleich anlegt, wenn er noch nicht existiert.

Wir lesen nacheinander alle drei Descriptor-Bytes (Stringlänge; Low- und High-Byte der Stringposition) und übertragen sie nach ZLAENGE und ZLAENGE, ZLAENGE+1 beziehungsweise POSIT und POSIT+1 (ZLAENGE+1=POSIT und ZLAENGE+2=POSIT+1, siehe Deklaration eigener Variablen).

Achten Sie bitte darauf, daß diese Routine unbedingt die C 128-spezifische Eigenschaft des Bankswitching berücksichtigen muß! Beim Lesen der Descriptoren greifen wir auf die Variablentabelle in Bank 1 und müssen daher unbedingt BANK1 aufrufen, unser Unterprogramm zum Umschalten auf diese Bank. Da nach der Rückkehr aus GETPOS Bank 0 eingeschaltet ist, würden wir sonst anstelle der Variablentabelle die Bytes an den entsprechenden Adressen in Bank 0 lesen. Nach dem Beenden der Routine schalten wir wieder auf die Standardbank 0 zurück.

Eingabeschleife/Zeichen unter Cursor invertieren

Die Aufgabe der Eingabeschleife ist schnell erklärt: Sie ruft GETIN auf, eine Routine, die analog dem Basic-Befehl GET ist, das heißt, ein Zeichen von der Tastatur liest, jedoch nicht wartet, wenn keine Taste gedrückt ist. Das Zeichen wird im Akku übergeben. Die Eingabeschleife wird erst verlassen, wenn der Akkuinhalt ungleich Null ist, das heißt, wenn eine Taste gedrückt wurde.

```
**EINGABESCHLEIFE**
GET  JSR INVERT  ;ZEICHEN INVERT.
GET1 JSR GETIN
     BEQ  GET
     JSR  INVERT
```

INVERT stellt ein Unterprogramm dar, das das Zeichen an der momentanen Cursorposition invertiert. Vor einer Eingabe wird das jeweilige Zeichen durch den Aufruf von INVERT daher revers dargestellt, nach einer Eingabe wieder invertiert, das heißt normal dargestellt. Diese Routine ist nötig, da GETIN ebenso wie GET keinen blinkenden Cursor erzeugt. Durch INVERT ergibt sich zwar ebenfalls kein blinkender, jedoch zumindest ein stehender Cursor, der dem Benutzer die momentane Eingabeposition angibt.

```
**Z.UNTER CRS.INVERTIEREN**
INVERT PHA
       LDY SPALTE
       LDA (ZEILPOI),Y
       EOR #128
       STA (ZEILPOI),Y
       PLA
       RTS
```

Beachten Sie bitte, daß das im Akku übergebene Zeichen durch INVERT nicht verändert werden darf, und daher auf den Stack gerettet und vor dem Rücksprung wieder vom Stack geholt wird. INVERT folgt im Programm übrigens nicht der Eingabeschleife, sondern befindet sich am Programmende.

Zeichenüberprüfung

Nach der Eingabe eines Zeichens muß überprüft werden, ob es sich um Carriage Return (RETURN = Code $0D) handelt. Wenn ja, ist die Eingabe beendet und der String mit den eingegebenen Zeichen wird angelegt (Routine RETURN).

```
**RETURN?**
CMP #$0D
BEQ RETURN
```

Wurde nicht RETURN gedrückt, muß überprüft werden, ob ein im Zeichenstring (im Beispiel Z$) enthaltenes Zeichen eingegeben wurde.

```
**ZULAESSIGES ZEICHEN?**
       LDY ZLAENGE
       DEY
COMPARE JSR BANK1
       CMP (ZPOSIT),Y
       JSR BANK0
       BEQ AUSGABE
       DEY
       BPL COMPARE
       BMI GET      ;ABS.SPRUNG
```

Erinnern Sie sich: in ZLAENGE wurde die Länge des Zeichenstrings und in ZPOSIT, ZPOSIT+1 der Descriptorzeiger auf den String selbst übertragen. Zur Überprüfung greifen wir in einer Schleife auf jedes einzelne Zeichen dieses Strings zu und vergleichen es mit dem im Akku enthaltenen Eingabe-

zeichen. Vor jedem Vergleich schalten wir auf Bank 1, also auf die Bank, in der Variablentabelle und Stringstack enthalten sind, und anschließend auf Bank 0. Diese Lösung ist sicher nicht die schnellste, die Geschwindigkeit der vorgestellten INPUT-Routine ist jedoch auch für »Schnelltipper« bei weitem ausreichend.

Diese bereits einmal angewendete Lösung für das Bankswitching, nach jeder Routine, die auf Bank 1 zugreift, prinzipiell wieder auf die Standardbank 0 zurückzuschalten, hat den Vorteil eines immer exakt definierten Ausgangszustands für weitere Routinen. Es ist dadurch unnötig, sich bei jedem Zugriff auf Bank 0 oder Bank 1 zu überlegen, welche Bank momentan gerade eingeschaltet ist, da beim Eintritt in eine neue Teilroutine des Gesamtprogramms immer Bank 0 eingeschaltet ist. Von dieser »sturen« Lösung abzugehen und damit zusätzliche Fehlerquellen in Kauf zu nehmen, empfehle ich Ihnen jedoch bei besonders zeitkritischen Programmen wie zum Beispiel Sortierroutinen.

Nach dieser kleinen Abschweifung zurück zur Zeichenüberprüfung: wurde auch das letzte Zeichen des Zeichenstrings mit dem eingegebenen Zeichen verglichen, ohne daß eine Übereinstimmung festgestellt wurde, wird zur Eingabeschleife zurückgesprungen, da die gedrückte Taste unzulässig war.

Ist das eingegebene Zeichen jedoch im Zeichenstring enthalten, erfolgt ein Sprung zur Zeichenausgabe:

```
**ZEICHENAUSGABE**
AUSGABE JSR BSOUT
        JMP GET
```

BSOUT gibt ein im Akku enthaltenes Zeichen aus. Das auszugebende Zeichen ist immer noch unverändert im Akku enthalten. Nach der Ausgabe erfolgt die Rückkehr zur Eingabeschleife.

Übernahme der eingegebenen Zeichen:

Nun folgen die für Sie wohl interessantesten Programmteile. Bisher wurden nur bereits erläuterte Routinen des Basic-Interpreters benutzt. Neu war nur die praktische Anwendung des Umschaltens zwischen Bank 0 und Bank 1. Im folgenden lernen Sie jedoch, wie ein String auf dem C 128 angelegt wird und welche Routinen und Zeropageadressen hierfür benötigt werden. Die Übernahme der eingegebenen Zeichen erfolgt in mehreren Schritten:

1. Cursor auf Startposition der Eingabe setzen und einen Pointer auf diese Startposition errechnen
2. Die echte Eingabelänge ermitteln, das heißt alle nachfolgenden Spaces abschneiden, als echte Länge jedoch nur einen Wert akzeptieren, der maximal so groß ist wie die als Parameter übergebene maximale Eingabelänge
3. CHKKOM und GETPOS aufrufen, das heißt einen Zeiger auf die Stringdescriptoren jenes Strings holen, in dem die Eingabe an das Basic übergeben werden soll
4. Speicherplatz für den anzulegenden String reservieren beziehungsweise prüfen, ob ausreichend Platz vorhanden ist
5. Den Bildschirm als logische Datei eröffnen und die Eingabe auf diese Datei legen
6. Die Eingabe vom Bildschirm lesen und in den Übergabestring übertragen
7. Datei schließen und die Eingabe wieder auf Tastatur legen
8. Die Descriptoren für den angelegten String und das Ende des Stringstacks aktualisieren

Cursor auf Startposition setzen/Pointer errechnen

Spalte und Zeile des Eingabebeginns wurden am Programmanfang auf den Stack gerettet. Diese Werte werden nun zurückgeholt und der Cursor auf die dadurch definierte Position gesetzt:

```
*CURSOR AUF STARTPOSITION*
RETURN PLA          ;GERETTETE
TAY                 ;STARTPOSITION
PLA                 ;HOLEN
TAX
CLC                 ;UND CURSOR
JSR SETCRS          ;DARAUF SETZEN
```

Ein Pointer auf diese Startposition wird nun errechnet, indem, wie bereits erwähnt, die momentane Cursorspalte zum Zeiger auf den Beginn der momentanen Cursorzeile addiert wird:

```
*POINTER AUF EINGABESTARTPOS.*
LDA ZEILPOI+1       ;POINTER AUF
STA STARTPOS+1      ;EINGABEBEGINN
LDA ZEILPOI         ;ERZEUGEN
CLC
ADC SPALTE
STA STARTPOS
BCC STARTP
INC STARTPOS+1
```

Nach dieser Addition befindet sich in STARTPOS, STARTPOS+1 ein Pointer, der exakt auf den Beginn der Eingabe zeigt.

Ermitteln der echten Eingabelänge

Um zu demonstrieren, was ich unter »echter« Eingabelänge verstehe, sehen Sie sich folgende möglichen Eingaben an (S = Space). Gehen Sie davon aus, daß das Basic-Programm als maximale Länge einer Eingabe zehn Zeichen vorgab:

C128SSSSSS.....
Commodore128SSSSSS.....

Im ersten Fall wurden nur vier Zeichen eingegeben, die von unserem Programm alle übernommen werden sollen. Die bis zur maximalen Länge folgenden Spaces auf dem Bildschirm sollen jedoch nicht mit in den anzulegenden String übernommen werden (ebensowenig wie beim Basic-Befehl INPUT). Als echte Eingabelänge muß daher eine Vier ermittelt werden. Im zweiten Beispiel wurden zwölf Zeichen eingegeben. Da die maximale Eingabelänge jedoch zehn beträgt und weitere Zeichen nicht übernommen werden sollen, muß unsere Routine als echte Länge eine Zehn ermitteln.

```
*ECHTE EINGABELAENGE ERMITTELN*
STARTP LDY LAENGE
       DEY
LECHT  LDA (STARTPOS),Y
       CMP #32
       BNE OKAY      ;NACHFOLGENDE
       DEY           ;SPACES
       CPY #255      ;ABSCHNEIDEN
       BNE LECHT
```

Die Schleife beginnt mit einem Y-Wert von Länge-1 (DEY), das heißt im Beispiel ist Y=9. Zuerst wird daher auf das neunte Zeichen hinter der Startposition zugegriffen, also auf das zehnte Zeichen der Eingabe. Nun wird dieses Zeichen mit SPACE (=Code 32) verglichen. Ist es ungleich SPACE, wird das vorangehende Zeichen mit SPACE verglichen und so weiter.

Bei dem ersten »Nicht-SPACE« wird der Vergleich abgebrochen und zu OKAY gesprungen. Im Y-Register ist die echte Eingabelänge minus eins enthalten.

Stringdescriptoren des Übernahmestring holen

Da das Y-Register die Länge minus eins enthält, erfolgt zuerst eine Korrektur (INY), bevor die echte Länge in LAENGE gespeichert wird. Anschließend werden CHKKOM und GETPOS aufgerufen, wir erhalten also in den Speicherzellen $49, $4A einen Zeiger auf die Descriptoren des Strings, in dem die Eingabe an das Basic übergeben werden soll.

```
*UEBERNAHMEVORBEREITUNGEN*
OKAY INY
     STY LAENGE    ;LECHT RETTEN
     JSR CHKKOM    ;=) ZEIGER AUF
     JSR GETPOS    ;UEBERN.STRING
```

Platz für den anzulegenden String reservieren (STRRS = $9299)

```
     LDA LAENGE
     JSR STRRES    ;PLATZ RESERV.
```

Nun wird der Akku mit der Länge der Eingabe geladen und die Routine STRRES ($9299) aufgerufen. Diese Routine muß vor dem Anlegen eines Strings unbedingt aufgerufen werden. Sie erfüllt mehrere Aufgaben:

1. Sie prüft, ob ausreichend Speicherplatz für einen anzulegenden String mit der im Akku übergebenen Länge vorhanden ist. Wenn nicht, wird eine Garbage Collection durchgeführt. Ist auch anschließend nicht ausreichend Platz vorhanden, wird ein OUT OF MEMORY ERROR ausgegeben.

2. STREND ($35) stellt einen Zeiger auf die Untergrenze des Stringstacks dar. Ein neuer String wird bis zu dieser momentanen Untergrenze angelegt, die nun um die Länge des anzulegenden Strings verringert werden muß. STRRES erledigt auch dieses Herabsetzen des Zeigers STREND um die angegebene Stringlänge. Der neue Wert von STREND ist unser Zeiger auf den Beginn unseres anzulegenden Strings, ab dem dieser Zeichen für Zeichen anlegt.

Bildschirm als logische Datei öffnen/Eingabe auf logische Datei legen

Es erscheint ziemlich umständlich, zum Einlesen der eingegebenen Zeichen den Bildschirm als logische Datei zu definieren und die Eingabe auf diese Datei zu legen. Würde jedoch die beim C 64 von den meisten INPUT-Routinen her verwendete Methode benutzt, die Zeichen ohne Verwendung von Betriebssystem-Routinen direkt vom Bildschirm zu lesen, wäre der Aufwand beim C 128 noch um ein Vielfaches höher: die vom Bildschirm gelesenen Zeichen müssen vor dem Abspeichern im Stringstack in das ASCII-Format konvertiert werden.

Beim C 64 genügt hierzu eine sehr einfache Wandlungsroutine. Der C 128 besitzt jedoch mehrere Zeichensätze – denken Sie an die Umlaute, Akzente, mathematischen Sonderzeichen etc. -, durch die die Wandlung ganz außerordentlich kompliziert und umständlich wird. Es ist daher auf dem C 128 einfacher, wenn möglich die Betriebssystem-Routinen zu verwenden, die diese Wandlung automatisch vornehmen.

```
*BILDSCHIRM: LOG.DATEI EROEFF.*
     LDA #3
     TAX
     TAY
     JSR SETFLS
     LDA #0
     JSR SETNAM
     JSR OPEN
     LDX #3
     JSR CHKIN
```

Der Bildschirm wird als Datei mit der logischen Filenummer drei und der Sekundäradresse definiert, die Länge des Filenamens mit Null angegeben, die Datei geöffnet und die Eingabe auf diese Datei gelegt. Jeder Aufruf der zum Lesen verwendeten Routine BASIN wird sich daher auf den Bildschirm beziehen.

Eingabe lesen/String anlegen

BASIN liest die Zeichen ab der momentanen Cursorposition vom Bildschirm und übergibt sie im Akku. In einer Schleife wird BASIN entsprechend der echten Eingabelänge aufgerufen und das gelesene Zeichen ab der aktualisierten neuen Untergrenze des Stringstacks (STREND) gespeichert. Vor dem Speichern eines gelesenen Zeichens wird auf Bank 1 und anschließend auf Bank 0 geschaltet. Beachten Sie bitte, daß das einmalige und daher zweifellos schnellere Umschalten auf Bank 1 vor der Schleife nicht genügt, da BASIN auf den Bildschirm zugreift, der in Bank 0 liegt und daher auf diese Bank umschaltet.

```
*LESEN/STRING ANLEGEN*
     LDY #0
LIES JSR BASIN
     JSR BANK1
     STA (STREND),Y
     JSR BANK0
     INY
     CPY LAENGE
     BNE LIES
```

Datei schließen/Standardeingabe setzen

Durch CLRCH wird die Eingabe wieder auf das Standardgerät, die Tastatur, gelegt und die geöffnete Datei anschließend mit CLOSE geschlossen.

```
*DATEI SCHLIESSEN/STAND.EING.*
      JSR CLRCH
      LDA #3
      JSR CLOSE
```

Descriptoren aktualisieren

Angenommen, als Übergabestring wurde vom Basic-Programm A$ angegeben. Die Stringdescriptoren von A$ weisen unverändert auf dessen alte Position und beinhalten dessen alte Länge. Wir müssen daher diese Descriptoren noch aktualisieren, also den Längendescriptor mit der Länge und den Positionszeiger mit der Position unseres Strings versehen. Die Länge ist in LAENGE enthalten und die Position entspricht der geänderten Untergrenze STREND des Stringstack, ab der wir unseren String anlegten.

```
*DESCRIPTOREN AKTUALISIEREN*
       JSR BANK1
DESAKT LDY #0
       LDA LAENGE
       STA (ZEIGER),Y
DAKT   LDA STREND,Y
       INY
       STA (ZEIGER),Y
       CPY #2
       BNE DAKT
       JSR BANK0
       RTS    ;ZURUECK INS BASIC
```

Die Eingabe wurde nun komplett übernommen und das Programm kehrt ins Basic zurück. Zur Erinnerung noch einmal das Format des Aufrufs:

```
SYS DEC("0D00"),Länge,Zeile,
Spalte,0,Zeichen$,Rückgabe$
```

Wie Sie wissen, kann GETPOS beliebige Variablentypen verarbeiten. Sie können daher als Zeichensatz- und als Rückgabestring auch Strings aus einem Stringarray angeben.

Die Beschreibung dieser INPUT-Routine ist nun beendet. Sie können auf Ihrem C 128 nun sowohl beliebige Parameter aus dem Basic-Text holen, als auch beliebige Variablen von Maschinensprache aus anlegen. Als Test zum Umgang mit den Basic-Interpreter-Routinen könnten Sie nun zum Beispiel das vorgestellte Programm komfortabler gestalten.

Eine vernünftige INPUT-Routine läßt zum Beispiel – im Gegensatz zu dieser Routine, die zeigen soll, wie mit den entsprechenden Interpreter-Routinen umzugehen ist – keine Cursorbewegungen zu, die aus der vorgegebenen Eingabezone herausführen. Wenn die maximale Eingabelänge, zum Beispiel zehn Zeichen beträgt, dürften nur Cursorbewegungen zwischen der Startposition und den nächsten neun Spalten zugelassen werden.

Zudem sollten INSERT und DELETE nur die Zeichen in der Eingabezone verschieben, ohne die restlichen Zeichen in der Zeile zu

beeinflussen und ein Endezeichenstring von Basic übergeben. Nicht nur, wenn RETURN, sondern eine beliebige in diesem String enthaltene Taste gedrückt wird, soll die Eingabe übernommen werden. Wenn das betreffende Zeichen, zum Beispiel CURSOR UP oder CURSOR DOWN dem Basic-Programm in einem String oder einer Integervariablen (ASCII-Code des Zeichens) übergeben wird, lassen sich sehr komfortable Eingabemasken gestalten.

INPUT-Routine

```
a 00d00 85 a5    sta $a5
a 00d02 8a       txa
a 00d03 48       pha
a 00d04 98       tya
a 00d05 48       pha
a 00d06 18       clc
a 00d07 20 f0 ff jsr $fff0
a 00d0a 20 5c 79 jsr $795c
a 00d0d 20 af 7a jsr $7aaf
a 00d10 20 20 0b jsr $0b20
a 00d13 a0 02    ldy #$02
a 00d15 b1 49    lda ($49),y
a 00d17 99 fb 00 sta $00fb,y
a 00d1a 88       dey
a 00d1b 10 f8    bpl $0d15
a 00d1d 20 16 0b jsr $0b16
a 00d20 20 bf 0d jsr $0dbf
a 00d23 20 e4 ff jsr $ffe4
a 00d26 f0 fb    beq $0d23
a 00d28 20 bf 0d jsr $0dbf
a 00d2b c9 0d    cmp #$0d
a 00d2d f0 18    beq $0d47
a 00d2f a4 fb    ldy $fb
a 00d31 88       dey
a 00d32 20 20 0b jsr $0b20
a 00d35 d1 fc    cmp ($fc),y
a 00d37 20 16 0b jsr $0b16
a 00d3a f0 05    beq $0d41
a 00d3c 88       dey
a 00d3d 10 f3    bpl $0d32
a 00d3f 30 df    bmi $0d20
a 00d41 20 d2 ff jsr $ffd2
a 00d44 4c 20 0d jmp $0d20
a 00d47 68       pla
a 00d48 a8       tay
a 00d49 68       pla
a 00d4a aa       tax
a 00d4b 18       clc
a 00d4c 20 f0 ff jsr $fff0
a 00d4f a5 e1    lda $e1
a 00d51 85 a4    sta $a4
a 00d53 a5 e0    lda $e0
a 00d55 18       clc
a 00d56 65 ec    adc $ec
a 00d58 85 a3    sta $a3
a 00d5a 90 02    bcc $0d5e
a 00d5c e6 a4    inc $a4
a 00d5e a4 a5    ldy $a5
a 00d60 88       dey
a 00d61 b1 a3    lda ($a3),y
a 00d63 c9 20    cmp #$20
a 00d65 d0 05    bne $0d6c
a 00d67 88       dey
a 00d68 c0 ff    cpy #$ff
a 00d6a d0 f5    bne $0d61
a 00d6c c8       iny
a 00d6d 84 a5    sty $a5
a 00d6f 20 5c 79 jsr $795c
a 00d72 20 af 7a jsr $7aaf
a 00d75 a5 a5    lda $a5
a 00d77 20 99 92 jsr $9299
a 00d7a a9 03    lda #$03
a 00d7c aa       tax
a 00d7d a8       tay ,
a 00d7e 20 ba ff jsr $ffba
a 00d81 a9 00    lda #$00
a 00d83 20 bd ff jsr $ffbd
a 00d86 20 c0 ff jsr $ffc0
a 00d89 a2 03    ldx #$03
a 00d8b 20 c6 ff jsr $ffc6
a 00d8e a0 00    ldy #$00
a 00d90 20 cf ff jsr $ffcf
a 00d93 20 20 0b jsr $0b20
a 00d96 91 35    sta ($35),y
a 00d98 20 16 0b jsr $0b16
a 00d9b c8       iny
a 00d9c c4 a5    cpy $a5
a 00d9e d0 f0    bne $0d90
a 00da0 20 cc ff jsr $ffcc
a 00da3 a9 03    lda #$03
a 00da5 20 c3 ff jsr $ffc3
a 00da8 20 20 0b jsr $0b20
a 00dab a0 00    ldy #$00
a 00dad a5 a5    lda $a5
a 00daf 91 49    sta ($49),y
a 00db1 b9 35 00 lda $0035,y
a 00db4 c8       iny
a 00db5 91 49    sta ($49),y
a 00db7 c0 02    cpy #$02
a 00db9 d0 f6    bne $0db1
a 00dbb 20 16 0b jsr $0b16
a 00dbe 60       rts
a 00dbf 48       pha
a 00dc0 a4 ec    ldy $ec
a 00dc2 b1 e0    lda ($e0),y
a 00dc4 49 80    eor #$80
a 00dc6 91 e0    sta ($e0),y
a 00dc8 68       pla
a 00dc9 60       rts
```

Zum Testen dieser Routine geben Sie bitte ein:

```
Z$="1234567890"+CHR$(29)+
CHR$(157)+CHR$(20)+CHR$(148)
SYS DEC("0D00"),10,20,5,0,Z$,A$
```

Der Cursor wird auf Spalte fünf der Zeile 20 gesetzt; Sie können beliebige Ziffern eingeben und die Eingabe mit CURSOR-RIGHT, CURSOR-LEFT, DEL und INST editieren. Nach RETURN werden maximal zehn Ziffern in die Variable A$ übernommen.

Außer der Variablen- und Stringbehandlung sind die Rechenroutinen des Basic-Interpreters oftmals interessant. So ist es für mich persönlich außerordentlich unangenehm, in Maschinensprache zu multiplizieren und zu dividieren. Da die entsprechenden Interpreter-Routinen nicht sehr schnell arbeiten, ist es für zeitkritische Programmteile notwendig, sich seine eigenen, möglichst optimal an das jeweilige Problem angepaßten Routinen zu schreiben.

In weniger zeitkritischen Programmteilen bietet es sich jedoch an, die bereits vorhandenen Routinen zu nutzen. Die Rechenroutinen des Interpreters arbeiten üblicherweise im Fließkommaformat. In reinen Assemblerprogrammen genügen oftmals Berechnungen, die ausschließlich mit Integerwerten durchgeführt werden. Auch in diesen Fällen können die Interpreter-Routinen genutzt werden, da wir im folgenden Routinen kennenlernen, die Integerwerte ins Fließkommaformat wandeln und weitere, mit denen nach beendeter Berechnung das Ergebnis wieder in einen Integerwert gewandelt wird.

Außer den reinen Rechenroutinen ist für viele Anwendungen eine weitere Routine sehr hilfreich, die einen Fließkommawert auf dem Bildschirm ausgibt. Überlegen Sie sich bitte, welchen Aufwand es erfordert, zum Beispiel die Spalte (Ein-Byte-Integer) und die Zeile (Zwei-Byte-Integer) des Cursors in einer Textverarbeitung ständig »per Hand« in einer Informationszeile auszugeben. Eine weitere Anwendung ist die ständige Ausgabe der momentan erreichten Punktzahl in einem Videospiel.

Da ich beide Anwendungsbeispiele selbst programmiert habe, bevor ich die entsprechenden Interpreter-Routinen kannte, weiß ich deren Vorhandensein nun umso mehr zu schätzen, und auch Ihnen könnte eines Tages ein entsprechendes Problem begegnen.

Alle Rechenroutinen laufen in den sogenannten »Fließkommaakkumulatoren« ab (FAC #1 und FAC #2). FAC #1 befindet sich beim C128 an Adresse $63 bis $69 und FAC #2 an Adresse $6A bis $71. Oftmals wird FAC #1 als FAC und FAC #2 als ARG bezeichnet, da letzterer bei Rechenoperationen mit zwei Fließkommawerten mit dem Argument versehen wird. Im folgenden werde ich diese Bezeichnungen übernehmen.

Auf die Fließkommadarstellung von Zahlen möchte ich in diesem Artikel nicht weiter eingehen, da deren exakte Erläuterung für die Anwendung der Rechenroutinen nicht unbedingt nötig ist und ich selbst zugegebenermaßen mit der

Fließkommaarithmetik nicht sonderlich vertraut bin.

Wichtig für uns ist, daß jeder Fließkomma-Akkumulator gleich aufgebaut ist. Für unsere Anwendungsbeispiele benötigen wir folgende Adressen:

```
$63      = FAC, Exponent
$64-$67 = FAC, Mantisse
$68      = FAC, Vorzeichen

$6A      = ARG, Exponent
$6B-$6E = ARG, Mantisse
$6F      = ARG, Vorzeichen
```

In den folgenden Beispielen werde ich drei Fließkommazahlen verwenden, mit denen die verschiedenen Programmbeispiele arbeiten werden:

```
dez.     hex      Fließkommaformat
  8      $08      84 80 00 00 00
 10      $A0      84 A0 00 00 00
4098     $1002    8D 80 00 00 00
```

Zur Erläuterung der verschiedenen Rechenroutinen würde es genügen, FAC und ARG mit den jeweiligen Werten zu laden, indem der eingebaute Monitor benutzt wird, die Rechenroutine aufzurufen und das Ergebnis wiederum mit dem Monitor zu betrachten. Nachdem ich diese Methode anwandte und völlig unsinnige Ergebnisse erhielt, mußte ich feststellen, daß der Monitor selbst FAC verwendet (für seinen Zeiger auf die momentane Adresse, zum Beispiel beim Assemblieren/Disassemblieren).

Im folgenden werden wir daher zwangsläufig unsere Beispielwerte mit einem Programm in die Fließkommaakkumulatoren schreiben und auch das Ergebnis vom Programm retten lassen, bevor wir den Monitor wieder benutzen.

Führen Sie bitte auch diesmal wieder die notwendige Initialisierung durch, wenn Sie nach den vorigen Beispielen den Rechner ausgeschaltet oder einen Reset durchgeführt haben. Als Initialisierungs-Programm genügt die Routine ohne die Unterprogramme zum Umschalten der Speicherbänke, da wir nicht auf Bank 1 zugreifen müssen.

FAC-Inhalt retten

Die folgende Maschinenroutine rettet den Inhalt des Fließkommaakkus Nummer 1, der beim Aufruf des Monitors durch diesen verändert wird:

```
a 00b30  a2 04     ldx #$04
a 00b32  b5 63     lda $63,x
a 00b34  9d 00 0d  sta $0d00,x
a 00b37  ca        dex
a 00b38  10 f8     bpl $0b32
a 00b3a  60        rts
```

Fließkomma nach Integer wandeln ($8CC7)

Diese Routine wandelt einen Fließkommawert, der sich im FAC befindet, in eine Zwei-Byte-Integerzahl, die wiederum im FAC abgelegt wird (in $66/$67 = High/Low).

```
a 00b40  a2 04     ldx #$04
a 00b42  bd 50 0b  lda $0b50,x
a 00b45  95 63     sta $63,x
a 00b47  ca        dex
a 00b48  10 f8     bpl $0b42
a 00b4a  20 c7 8c  jsr $8cc7
a 00b4d  4c 30 0b  jmp $0b30
a 00b50  84 a0     sty $a0
a 00b52  00        brk
a 00b53  00        brk
a 00b54  00        brk
```

Da das Ende dieses Listings ein wenig ungewöhnlich erscheinen mag, will ich die letzten Befehle genauer erläutern: Um eine Integerzahl in eine Fließkommazahl oder eine Fließkommazahl in eine Integerzahl zu wandeln, muß erstere im FAC #1 abgelegt sein. In diesem Beispiel verwende ich die Dezimalzahl zehn, die sich, im Fließkommaformat abgelegt, am Programmende befindet:

```
m 0b50 84 a0 00 00 00
```

Das Programm überträgt diese Werte nach FAC #1. Nach dem Aufruf der Routine $8CC7 befindet sich die entsprechende Integerzahl in den Adressen $66, $67 (High/Low) und wird durch das Unterprogramm $0B30 nach $0D00 bis $0D04 übertragen und vor dem Überschreiben durch den Monitor gerettet.

Rufen Sie die Routine mit »SYS DEC("0B40")« auf und schauen Sie sich $0D00... mit dem Monitor an:

```
m 0d00 84 00 00 00 A0
```

In $0D03 und $0D04 befindet sich die Integerzahl $00A0.

2-Byte-Integer (vorzeichenbehaftet) nach Fließkomma wandeln ($8C70)

Es gibt zwei Arten der Darstellung von Integerzahlen: vorzeichenbehaftet und positiv. Als Assemblerprogrammierer benötigen Sie üblicherweise nur die positive Darstellung, bei der alle 16 Bits für den Absolutwert einer Zahl verwendet werden, die damit im Bereich zwischen $0000 bis $FFFF liegen kann.

Bei der vorzeichenbehafteten Darstellung wird Bit-16 als Vorzeichen betrachtet (gesetzt = negativ). Eine Vorzeichenbehaftete Zwei-Byte-Integerzahl besitzt daher einen Wert zwischen -$7FFF und +$7FFF. Diese Routine wird zum Beispiel beim C64 zur Ausgabe von FRE(0) verwendet, in diesem Fall leider fälschlicherweise, da jeder C64-Programmierer das Problem kennt, daß bei Werten, die größer sind als $7FFF der freie Speicherplatz angeblich negativ ist.

Die Integerzahl wird dieser Routine in den Speicherzellen $64, $65 (High/Low) übergeben. Das X-Register muß vor dem Aufruf unmittelbar mit $90 geladen werden:

```
a 00b60  a9 01     lda #$01
a 00b62  a2 00     ldx #$00
a 00b64  85 64     sta $64
a 00b66  86 65     stx $65
a 00b68  a2 90     ldx #$90
a 00b6a  20 70 8c  jsr $8c70
a 00b6d  4c 30 0b  jmp $0b30
```

Nach dem Aufruf mit »SYS DEC("0B60")« befindet sich die übergebene Zahl +$0100 im Fließkommaformat im FAC #1. Die Routine $0B30 überträgt sie nach $0D00 bis $0D04, wovon Sie sich wiederum mit dem Monitor überzeugen können.

2-Byte-Integer (positiv) nach Fließkommaformat wandeln ($8C75)

Wahrscheinlich werden Sie diese Routine, die die üblichen positiven Integerwerte ins Fließkommaformat wandelt, häufiger in Ihren Programmen verwenden. Auch dieser Routine wird die zu wandelnde Integerzahl in den Speicherzellen $64, $65 übergeben und auch das X-Register vor dem Aufruf unmittelbar mit $90 geladen. Weiterhin muß das Carry-Flag vor dem Einsprung gesetzt werden.

```
a 00b80  a9 ff     lda #$ff
a 00b82  a2 01     ldx #$01
a 00b84  85 64     sta $64
a 00b86  86 65     stx $65
a 00b88  a2 90     ldx #$90
a 00b8a  38        sec
a 00b8b  20 75 8c  jsr $8c75
a 00b8e  4c 30 0b  jmp $0b30
```

Dieses Demoprogramm übergibt den Wert $FF01. Nach dem Aufruf mit »SYS DEC("0B80")« befindet sich die gerettete Fließkommazahl in $0D00 bis $0D04:

```
m 0d00  90 ff 01 00 00
```

Wenn Sie in Ihren Programmen nur mit Integerzahlen rechnen, werden Sie sich fragen, was Sie mit Routinen zur Umwandlung von Integer- in Fließkommazahlen anfangen sollen. Zum Beispiel, die Integerzahl auf dem Bildschirm ausgeben. Eine solche Ausgabe ist unter Verwendung von zwei weiteren Routinen möglich, jedoch erst, nachdem die Integerzahl ins Fließkommaformat gewandelt wurde:

FAC #1 nach ASCII wandeln ($8E42)

Diese Routine legt eine Fließkom-

mazahl, die sich im FAC #1 befindet, als String (ASCII-Format) ab, der sich wiederum im FAC #1 befindet. Interessant wird diese Wandlung in Verbindung mit einer weiteren Routine:

String in FAC #1 ausgeben ($55E2)

$55E2 gibt einen String, der sich im FAC #1 befindet, auf dem Bildschirm ab der momentanen Cursorposition aus.

Beide Routinen benötigen zur Vorbereitung nur die Fließkommazahl in FAC #1 ($8E42) beziehungsweise den String in FAC #1 ($55E2) und können ohne Übergabe weiterer Parameter aufgerufen werden. Wir besitzen nun alle erforderlichen Kenntnisse, um eine beliebige Integerzahl auf dem Bildschirm auszugeben, wie es für jede Textverarbeitung erforderlich ist.

Zuerst übergeben wir eine positive Integerzahl an die Speicherzellen $64, $65 und rufen $8C75 auf, die die Wandlung ins Fließkommaformat vornimmt. Anschließend wird $8E42 aufgerufen und die Zahl dadurch in einen String umgewandelt, der mit $55E2 ab der momentanen Cursorposition ausgegeben wird:

```
a 00ba0  a9 ff     lda #$ff
a 00ba2  a2 01     ldx #$01
a 00ba4  85 64     sta $64
a 00ba6  86 65     stx $65
a 00ba8  a2 90     ldx #$90
a 00baa  38        sec
a 00bab  20 75 8c  jsr $8c75
a 00bae  20 42 8e  jsr $8e42
a 00bb1  4c e2 55  jmp $55e2
```

Das Demoprogramm verwendet die Zahl $FF01, also dezimal 65281. Wenn Sie es mit »SYS DEC("0BA0")« starten, wird eben diese Zahl auf dem Bildschirm ausgegeben. Äußerst angenehm an diesen Routinen ist, daß Sie zwar nicht ohne das Fließkommaformat auskommen, der Programmierer selbst mit diesem jedoch nicht im geringsten in Berührung kommt, sondern nur eine Integerzahl übergibt.

2-Byte-Integer (positiv) ausgeben ($8E32)

Die Wandlungs- und Ausgaberoutinen, die zur Ausgabe einer Integerzahl nacheinander aufgerufen werden, können selbstverständlich unabhängig voneinander auch für andere Zwecke eingesetzt werden. Zur Ausgabe einer Integerzahl auf dem Bildschirm gibt es eine weitere, besser geeignete Routine, die ab der Adresse $8E32 liegt. Dieser Routine wird die betreffende Integerzahl im Akku und im X-Register übergeben (Akku = Low-, X = High-Byte). Sie legt diese Werte in den Speicherzellen $64, $65 ab und ruft die beschriebenen Routinen in der zur Bildschirmausgabe notwendigen Reihenfolge auf. Die Ausgabevorbereitungen beschränken sich daher auf das Laden dieser Register. Danach kann die Routine aufgerufen werden:

```
a 00bc0  a9 ff     lda #$ff
a 00bc2  a2 01     ldx #$01
a 00bc4  4c 00 00  jmp $8e32
```

Diese drei Programmzeilen besitzen die gleiche Funktion wie das vorige Demoprogramm. Sie geben ebenfalls den Wert $FF01, also die Dezimalzahl 65281 auf dem Bildschirm aus. Der Aufruf erfolgt mit »SYS DEC("0BC0")«.

Alle im folgenden beschriebenen Routinen führen Berechnungen mit zwei Fließkommazahlen durch (Addition, Division etc.), die sich in FAC #1 und FAC #2 befinden und speichern das Ergebnis im FAC #1. Da uns nun bekannt ist, auf welche Weise Integerzahlen ins Fließkommaformat gewandelt werden, können wir sie ebenso für Berechnungen mit zwei Integerzahlen verwenden, indem wir diese mit den entsprechenden Routinen ins Fließkommaformat wandeln.

Diese Anwendung kann vor allem in Maschinenprogrammen, in denen häufig multipliziert und dividiert werden muß, das Schreiben eigener Routinen ersparen. Bei zeitkritischen Programmteilen sollte die Verwendung der Fließkommaarithmetik jedoch unbedingt vermieden werden, da die Integerzahlen zuerst aufwendig gewandelt werden müssen und anschließend die noch weit aufwendigeren Fließkommarechenoperationen erfolgen. In solchen Fällen ist es angebrachter, eigene Integer-Rechenroutinen zu schreiben.

Gerade unsinnig wird die Verwendung der vorgestellten Routinen in speziellen Fällen wie zum Beispiel der Multiplikation oder Division einer (Zwei-Byte-) Integerzahl mit zwei, vier etc., da diese mit wenigen Rotations- und Verschiebebefehlen erledigt werden kann. Merken Sie sich daher bitte: für Rechenoperationen mit den in Assemblerprogrammen zumeist gebrauchten Integerzahlen sollte die eingebaute Fließkommaarithmetik nur in zeitunkritischen Fällen verwendet werden!

FAC = FAC + ARG ($8848)

Diese Routine addiert zwei in FAC #1 (= FAC) und FAC #2 (= ARG) hinterlegte Fließkommazahlen.

```
a 00bd0  a2 04     ldx #$04
a 00bd2  bd e8 0b  lda $0be8,x
a 00bd5  95 63     sta $63,x
a 00bd7  bd ed 0b  lda $0bed,x
a 00bda  95 6a     sta $6a,x
a 00bdc  ca        dex
a 00bdd  10 f3     bpl $0bd2
a 00bdf  20 48 88  jsr $8848
;FAC = FAC + ARG
a 00be2  20 42 8e  jsr $8e42
a 00be5  4c e2 55  jmp $55e2
a 00be8  84 80     sty $80
a 00bea  00        brk
a 00beb  00        brk
a 00bec  00        brk
a 00bed  8d 80 10  sta $1080
a 00bf0  00        brk
a 00bf1  00        brk
```

Rufen Sie die Routine mit »SYS DEC("0BD0")« auf. Die Fließkommazahlen ($08 = 8 und $1002 = 4098) befinden sich am Ende des Programms und werden in einer Schleife an den FAC und ARG übergeben. Anschließend wird die Additions-Routine, die ASCII-Wandlungs- und die Ausgabe-Routine aufgerufen. Wir erhalten die Ausgabe der Zahl 4106 = 8 + 4098.

FAC = ARG – FAC ($8831)

Auch diese Routine benötigt zwei in FAC und ARG hinterlegte Fließkommazahlen. Achten Sie bitte darauf, daß FAC von ARG subtrahiert wird und nicht umgekehrt. Um die Anwendung zu demonstrieren, ersetzen Sie bitte einfach im letzten Programm den Befehl »jsr 8848« durch »jsr 8831«:

```
...
...
...
a 00bdc  ca        dex
a 00bdd  10 f3     bpl $0bd2
a 00bdf  20 48 88  jsr $8848
;FAC = FAC - ARG
a 00be2  20 42 8e  jsr $8e42
a 00be5  4c e2 55  jmp $55e2
...
...
...
```

Als Ergebnis erhalten wir 4090 = 4098 – 8.

FAC = ARG * FAC ($8A27)

Der Aufruf unterscheidet sich in keiner Weise von den vorangehenden Arithmetik-Routinen. Ersetzen Sie daher »jsr 8848« durch »jsr 8a27«:

```
...
...
...
a 00bdc  ca        dex
a 00bdd  10 f3     bpl $0bd2
a 00bdf  20 48 88  jsr $8a27
;FAC = FAC * ARG
a 00be2  20 42 8e  jsr $8e42
a 00be5  4c e2 55  jmp $55e2
...
...
...
```

Als Ausgabe erhalten wir 32784 = 8 * 4098.

FAC = ARG / FAC ($8B4C)

Ändern Sie nun »jsr 8a27« ab in »jsr 8b4c«. Sie erhalten die Ausgabe von 512.25 = 4098 / 8. Achten Sie bitte wie bei der Subtraktion auch hier auf die Reihenfolge von Dividend und Divisor:

```
 ...
 ...
 ...
a 00bdc  ca       dex
a 00bdd  10 f3    bpl $0bd2
a 00bdf  20 48 88 jsr $8b4c
;FAC = ARG / FAC
a 00be2  20 42 8e jsr $8e42
a 00be5  4c e2 55 jmp $55e2
 ...
 ...
 ...
```

Zum Abschluß möchte ich noch zwei Spezialroutinen vorstellen, die in den entsprechenden Fällen zum einen weniger Vorbereitungen benötigen und zum anderen erheblich schneller arbeiten als die Benutzung von FAC=FAC*ARG oder FAC=ARG/FAC.

FAC = FAC * 10 ($8B17)

Sollte in Ihren Programmen eine entsprechende Anwendung gegeben sein, so ersparen Sie sich mit dieser Routine die Wandlung von Zehn ins Fließkommaformat.

```
a 00bd0  a2 04    ldx #$04
a 00bd2  bd e8 0b lda $0be3,x
a 00bd5  95 63    sta $63,x
a 00bd7  ca       dex
a 00bd8  10 f3    bpl $0bd2
a 00bda  20 48 88 jsr $8b17
;FAC = FAC * 10
a 00bdd  20 42 8e jsr $8e42
a 00be0  4c e2 55 jmp $55e2
a 00be3  84 80    sty $80
a 00be5  00       brk
a 00be6  00       brk
a 00be7  00       brk
```

Der Aufruf mit »SYS DEC("0BD0")« führt zur Ausgabe von 80 = 8 * 10.

FAC = FAC / 10 ($8B38)

```
 ...
 ...
 ...
a 00bd7  ca       dex
a 00bd8  10 f3    bpl $0bd2
a 00bda  20 48 88 jsr $8b17
;FAC = FAC * 10
a 00bdd  20 42 8e jsr $8e42
a 00be0  4c e2 55 jmp $55e2
 ...
 ...
 ...
```

Nach dem Aufruf erhalten wir die Ausgabe von 0,8 = 8 / 10.

Wollen Sie sich die Fließkommaarithmetik auch für das Rechnen mit Integerzahlen zunutze machen, sieht der Ablauf prinzipiell wie folgt aus:

1. Akku und X-Register mit dem ersten Wert laden
2. Eine eigene Unterroutine aufrufen, die die Registerinhalte benutzt, um sie der Wandlungsroutine INTEGER nach FLIESSKOMMA zu übergeben und diese aufruft
3. Eine weitere eigene Unter-Routine aufrufen, die die FAC nach ARG kopiert
4. Akku und X-Register mit dem zweiten Wert laden
5. Die Unter-Routine von 2. aufrufen
6. Die entsprechende Arithmetik-Routine aufrufen
7. Ein eigenes Unterprogramm aufrufen, das die Wandlungsroutine FLIESSKOMMA nach INTEGER aufruft und den Inhalt der Speicherzellen $66, $67 dem Hauptprogramm im Akku und im X-Register übergibt

Nachdem Sie diese kleinen Unterprogramme geschrieben haben, genügen zur Multiplikation von $00A5 mit $02B2 folgende Befehle:

```
LDA #$B2
LDX #$02
JSR ROUTINE1    ;$02b2 nach
                Fließkomma wandeln
JSR ROUTINE2    ;FAC in ARG
                kopieren
LDA #$A5
LDX #$00
JSR ROUTINE1    ;$00a5 nach
                Fließkomma
                wandeln
JSR ARITHMETIKROUTINE
                ;Operation
                durchführen
JSR ROUTINE3    ;Fließkomma nach
                Integer wandeln
                und nach A,X
```

(S. Baloui / ah)

Die wichtigsten Interpreterroutinen des C128

Routine	Parameter hin	Parameter zurück	C128
Parameter aus dem Basic-Text holen:			
CHKKOM	...(Komma)	keine	$795C
GETBYT	..,(Byte)	Byte im X-Register	$87F4
GETBYT MIT CHRGET	..,(Byte)	Byte im X-Register	$87F1
FRNUM	..,(num.Ausdruck)	Ergebnis in FAC #1	$77D7
ADRFOR	FAC #1: num. Ausdruck	Adresse in Y/Akku und in $16,$17	$8815
GETADR	..,(Adresse)	Adresse in Y/Akku und in $16,$17	$880F
ADRBYT	..,(Adresse),(Byte)	Byte im X-Register Adresse in $16,$17	$8803
Anlegen von Variablen:			
GETPOS	..,(belieb.Variable)	Zeiger auf die Var. in Akku/Y und $49,$4A Var. Name in $47,$48	$7AAF
STRRES	Stringlänge im Akku	Ev. OUT OF-MEMORY STREND ($35,$36) um Länge verringern	$9299
TYPFLAGS	keine	$0F:$00=num.,$FF=Str. $10:$00=Real,$80=Int$10	$0F
Routinen zur Fließkommaarithmetik:			
FLIESSK.NACH INTEGR	Fließk. Zahl in FAC	Integerz. in $66,$67	$8CC7
INT. MIT VORZ. NACH FLIESSK.	Int.Zahl in $64, $65 X=$90	Fließk. Zahl in FAC	$8C70
INT. NACH FLIESSKOMMA	Int. Zahl in $64, $65 X=90, SEC	Fließk. Zahl in FAC	$8C75
FAC NACH ASCII	Fließkommazahl in FAC	Zahlenstring in FAC	$8E42
AUSGABE VON STRING IM FAC	Zahlenstring in FAC	Ausgabe auf Screen	$55E2
AUSGABE VON POS. INTEGERZAHL	Integerzahl in Akku/X	Ausgabe auf Screen	$8E32
FAC = FAC + ARG	Fließkommaz. in FAC/ARG	Ergebnis in FAC	$8848
FAC = ARG - FAC	Fließkommaz. in FAC/ARG	Ergebnis in FAC	$8831
FAC = ARG * FAC	Fließkommaz. in FAC/ARG	Ergebnis in FAC	$8A27
FAC = ARG / FAC	Fließkommaz. in FAC/ARG	Ergebnis in FAC	$8B4C
FAC = FAC * 10	Fließkommazahl in FAC	Ergebnis in FAC	$8B17
FAC = FAC / 10	Fließkommazahl in FAC	Ergebnis in FAC	$8B38