Dateiverwaltung mit dem C16 in Maschinensprache

Um Ihre Basic-Programme zu beschleunigen, sind unterstützende Maschinenroutinen sehr hilfreich. Am Beispiel »Dateiverwaltung« zeigen wir Ihnen, wie man so etwas macht.

Dieser Artikel behandelt die Datenein- und -ausgabe in Maschinensprache mit Hilfe der im Betriebssystem des C16 vorhandenen Routinen. Eine dieser Routinen kennt wohl jeder Assemblerprogrammierer: Die Routine BSOUT ($FFD2), die häufig zur Ausgabe von Zeichen auf dem Bildschirm verwendet wird. Eine weitere bekannte Routine ist GETIN ($FFE4), die ein Zeichen von der Tastatur einliest.

Wenn Sie sich in Ihren Assemblerprogrammen auf Tastatur und Bildschirm als Peripheriegeräte beschränken, genügen Ihnen die genannten Routinen. Sollten Sie jedoch beabsichtigen, komplette Anwenderprogramme in Assembler zu schreiben, zum Beispiel eine Dateiverwaltung oder Textverarbeitung, müssen Sie zuvor folgende Probleme lösen:

1. Wie kann ich Daten statt auf den Bildschirm an beliebige Peripheriegeräte senden oder von diesen einlesen?
2. Wie erkenne ich bei Disketten- oder Datasettendateien das Dateiende, das in Basic mit der Statusvariablen ST abgefragt werden kann?
3. Woran erkenne ich, ob während der Datenübertragung Fehler auftraten?

Dieser Artikel wird - hoffentlich - alle derartigen Fragen beantworten. Zuvor will ich jedoch noch klären, warum es vorteilhaft ist, eine Dateiverwaltung in Maschinensprache zu programmieren, selbst wenn Sie vielleicht nie ein komplettes Anwenderprogramm in Maschinensprache schreiben werden.

Die meisten größeren Programme für den C16 werden Sie wahrscheinlich in Basic schreiben, da es außerordentlich aufwendig - und fehlerbehaftet - ist, ein größeres Programm vollständig in Maschinensprache zu erstellen. Am häufigsten wird Maschinensprache für kleinere Unterprogramme verwendet, um zeitkritische Basic-Programmteile zu ersetzen. Paradebeispiel für dieses Einsatzgebiet sind Sortierroutinen, die - in Basic programmiert - bei großen Datenmengen oftmals recht einschläfernd sein können.

Ein weiteres Einsatzgebiet sind Assemblerroutinen, die Mängel des eingebauten Basic-Interpreters beheben sollen, zum Beispiel eigene INPUT-Routinen.

Viele nützliche Assemblerroutinen können jedoch nur dann erstellt werden, wenn Sie die Datenein- und ausgabe auf der Maschinenspracheebene beherrschen. Vorstellbar wäre zum Beispiel ein sogenannter Drucker-Spooler, ein Assemblerprogramm, das in den Systeminterrupt eingebunden ist, der jede sechzigstel Sekunde erfolgt. Bei jedem Interrupt wird ein Zeichen des auszudruckenden Programmlistings oder Textes an den Drucker geschickt. Während der Text durch diese »im Hintergrund laufende« Interruptroutine ausgedruckt wird, können Sie ungehindert mit Ihrem C16 arbeiten, zum Beispiel ein Basic-Programm ablaufen lassen oder editieren.

Ein weiteres Beispiel wäre eine INPUT #-Routine, die einen String von Kassette oder Diskette einliest. Wozu, werden Sie sich fragen, da doch das C16-Basic den Befehl INPUT # bereits besitzt? Nun, jeder der sich bereits intensiver mit Dateiverwaltung beschäftigt hat, weiß, daß INPUT # den Programmierern des Basic-Interpreters gewiß keinen Ruhm einbrachte. Dieser Befehl arbeitet ebenso katastrophal wie der Befehl INPUT: Soll ein String eingelesen werden, der länger ist als 80 Zeichen ist, erscheint die Fehlermeldung »String too long error«. Enthält der String ein Komma oder ein Semikolon, geht das Einlesen schief, da beide Zeichen ebenso wie ein Return (chr$(13)) als Stringendemarke aufgefaßt werden.

Im folgenden Artikel werde ich die wichtigsten Betriebssystemroutinen des C16/C116 zur Datenein- beziehungsweise Datenausgabe vorstellen. Bei der Beschreibung dieser Routinen werden die Funktionen erläutert, was bei Verwendung der jeweiligen Routine zu beachten ist (Startadresse; Übergabeparameter), und welche Parameter - die vor allem zur Erkennung eventuell aufgetretener Fehler dienen - nach dem Aufruf der Routinen zurückübergeben werden.

Alle Beispielprogramme können problemlos mit dem Monitor des C16 eingegeben werden. Sowohl der Aufruf als auch die Startadressen der verwendeten Betriebssystemroutinen sind ohne jede Änderung auch auf den C64 und den C128 übertragbar, im Gegensatz zu den im letzten Teil des Artikels erläuterten Routinen des Basic-Interpreters, deren Einsprungadressen - und teilweise auch deren Funktion - leider von Computer zu Computer unterschiedlich ist.

Den Abschluß dieses Kapitels bildet die Programmierung einer INPUT #-Routine, die Sie in all Ihren Programmen als Ersatz für den ziemlich mangelhaften Basic-Befehl INPUT # einsetzen können, und die in der Lage ist, Strings bis zu einer Maximallänge von 255 Zeichen einzulesen, wobei der String im Gegensatz zum INPUT # des Basic-Interpreters beliebige Zeichen enthalten kann.

Öffnen einer Datei

Sie alle kennen den Basic-Befehl OPEN, mit dem eine Datei für Lese- oder Schreibzugriffe geöffnet werden kann. Erst nach dem Öffnen einer Datei kann die eigentliche Datenein- oder -ausgabe mit PRINT # oder INPUT # erfolgen. Die Programmierung auf der Maschinenspracheebene verläuft analog. Zuerst muß eine Datei geöffnet werden, wobei die gleichen Dateiparameter angegeben werden wie im OPEN-Befehl, dessen genaue Syntax bekanntlich lautet:

OPEN (LF), (GA), (SA), "DATEINAME, TYP, MODUS"

LF = Logische Filenummer
GA = Geräteadresse
SA = Sekundäradresse

Dieser für die Dateibehandlung grundlegende Basic-Befehl kann leider nur mit großem Aufwand in die entsprechenden Maschinenbefehle umgesetzt werden. Sollten Sie annehmen, daß hierzu der Aufruf einer einzigen Betriebssystemroutine ausreicht, muß ich Sie leider enttäuschen. Im Betriebssystem existiert zwar eine Routine mit dem Namen OPEN. Bevor diese Routine aufgerufen werden kann, müssen zuvor zwei Vorbereitungsroutinen aufgerufen werden, mit denen die benötigten Dateiparameter übergeben werden.

Mit der zuerst aufzurufenden Routine SETFLS ($FFBA) werden die Parameter logische Filenummer, Geräteadresse und Sekundäradresse übergeben. Anschließend wird die Routine SETNAM ($FFBD) aufgerufen, die für die Übergabe des Dateinamens zuständig ist (inklusive der Zusätze Typ und Modus). Erst nach dieser Vorbereitung kann die OPEN-Routine ($FFC0) aufgerufen werden.

SETFLS ($FFBA): Fileparameter setzen

Der erste Schritt zum Öffnen einer Datei - gleich ob zum Lesen oder Schreiben von Daten - ist die Übergabe der Parameter logische Filenummer, Geräteadresse und Sekundäradresse mit der Routine SETFLS.

Als logische Filenummer kann eine beliebige Zahl zwischen 1 und 255 angegeben werden.

Die Geräteadresse liegt üblicherweise zwischen Null und Acht (0=Tastatur; 1=Datasette; 3=Bildschirm; 4=Drucker; 8=Floppy). In den folgenden Programmbeispielen werde ich immer (!) die Geräteadresse Eins verwenden, da die Datasette wohl das am häufigsten in Verbindung mit dem C16 verwendete Peripheriegerät ist.

Die Bedeutung der Sekundäradresse ist je nach verwendetem Peripheriegerät völlig unterschiedlich. Soll eine Diskettendatei geöffnet werden, kann eine beliebige Sekundäradresse zwischen 2 und 14 angegeben werden.

Bei Druckern bestimmt die angegebene Sekundäradresse häufig den Druckmodus, ob zum Beispiel im Groß-/Grafik- oder im Klein-/Großmodus gedruckt werden soll. Mit welcher Sekundäradresse welcher Druckmodus angesprochen wird, ist von Drucker zu Drucker unterschiedlich und muß daher im zugehörigen Druckerhandbuch nachgelesen werden.

Beim Öffnen einer Datasettendatei kann eine Sekundäradresse zwischen Null und Zwei angegeben werden. Null bedeutet, daß anschließend Daten vom Band gelesen werden sollen, 1 geben Sie an, wenn Schreibzugriffe erfolgen sollen. Die Angabe einer 2 bedeutet ebenfalls, daß anschließend Schreibzugriffe erfolgen. Zusätzlich wird eine sogenannte »EOT«-Markierung auf das Band geschrieben, eine Markierung, die angibt, daß sich die Datei am Bandende befindet und keine weitere Datei folgt. Sinn und Zweck dieser Markierung blieb mir persönlich jedoch etwas schleierhaft, da ich sie in meinen Programmen noch nie benötigt habe. Auch in den folgenden Beispielprogrammen werde ich daher ausschließlich die Sekundäradressen Null und Eins verwenden.

Sie wissen nun, welche Parameter der Routine SETFLS übergeben werden müssen, nicht jedoch wie. Alle drei Parameter werden mit Hilfe der Prozessorregister übergeben:

Akku = logische Filenummer (1 bis 255)
X-Register = Geräteadresse (0, 1, 3, 4 oder 8)
Y-Register = Sekundäradresse (Datasette: 0 und 1; Floppy: 2 bis 14)

Wenn wir zum Beispiel Daten in eine Datasettendatei schreiben wollen, übergeben wir der Routine SETFLS folgende Parameter:

```
LDA #$01     ;LOGISCHE FILENUMMER 1
LDX #$01     ;GERÄTEADRESSE 1 = DATASETTE
LDY #$01     ;SEKUNDÄRADRESSE 1 = DATEN AUF BAND
             SCHREIBEN
JSR $FFBA    ;SETFLS AUFRUFEN
```

Wenn wir Daten vom Band lesen wollen, genügt es, den Befehl LDY #$01 abzuändern in LDY #$00.

SETNAM ($FFBD): Dateinamen (+Zusätze) übergeben

Der Dateiname darf aus maximal 16 Zeichen bestehen. Unter »Zusatz« sind die Angaben »Typ« und »Modus« zu verstehen. Beide Angaben werden nur bei Diskettendateien benötigt, um zum Beispiel mit dem Befehl

```
OPEN 2,8,2,"TEST,S,W"
```

die sequentielle Datei »test« zum Schreiben zu öffnen.

Typ = S (Sequentielle Datei), R (Relative Datei), P (Programmdatei) oder U (Userdatei).
Modus = R (»Read«=Daten lesen), W (»Write«=Daten schreiben) oder A (»Append«=Daten an bestehende Datei anhängen)

Die Angabe des Dateityps wird bei Datasettendateien nicht benötigt, da die Datasette nur eine einzige Form der Datendatei kennt, den sequentiellen Typ.

Die Angabe der Zugriffsart, des Modus, ist bei Verwendung der Datasette ebenfalls überflüssig, da diese Angabe bereits beim Aufruf von SETFLS erfolgte (Sekundäradresse 0 bis 2). Da wir im folgenden nur mit der Datasette arbeiten werden, können wir uns auf die Angabe des Dateinamens beschränken.

Bei Verwendung der Datasette muß übrigens nicht unbedingt ein Dateiname angegeben werden. Die angelegte Datei erhält dann natürlich keinen Namen. In den folgenden Beispielen werde ich dennoch immer Dateinamen verwenden. Sollten Sie jemals von der Datasette auf eine Floppy umsteigen, bleibt Ihnen dadurch die Umgewöhnung an Dateinamen erspart.

Die Parameter für SETNAM werden teilweise ebenfalls mit Hilfe der Prozessorregister übergeben. Zusätzlich muß jedoch in einem beliebigen Speicherbereich, im Programm selbst, in der Zeropage, oder an einem beliebigen anderen Ort der Dateiname in ASCII-Form abgelegt sein. Wenn Sie die RS232-Schnittstelle nicht benutzen (um zum Beispiel ein Modem zu betreiben), bietet sich der Bereich $03F7 bis $0436 (dezimal: 1015 bis 1078) an, der ausschließlich von dieser Schnittstelle verwendet wird. Wenn der Dateiname in diesem Bereich abgelegt wurde, wird der Akku mit der Länge des Namens geladen. Das X- beziehungsweise Y-Register enthält einen Zeiger auf die Adresse, an der sich der Name befindet (X-Register=Low-Byte der Adresse; Y-Register=High-Byte):

```
LDA #$04       ;LÄNGE DES NAMENS: 4 ZEICHEN
LDX #$F7       ;LOW-BYTE DER ADRESSE $03F7
LDY #$03       ;HIGH-BYTE DER ADRESSE $03F7
JSR $FFBD      ;SETNAM AUFRUFEN
```

Diese Routine ist nur dann lauffähig, wenn zuvor ein Dateiname mit einer Länge von vier Zeichen in ASCII-Form ab $03F7 abgelegt wurde.

OPEN ($FFC0): Datei öffnen

Nach diesen langwierigen Vorbereitungen kann endlich die OPEN-Routine des Betriebssystems aufgerufen werden. Der Aufruf mit JSR $FFC0 genügt, da ausnahmsweise keine weiteren Parameter übergeben werden müssen. Die logische Datei wird nun geöffnet. Dabei ist es jedoch möglich, daß ein Fehler auftritt, zum Beispiel wenn auf Band geschrieben werden soll und die Datasette noch im Schrank liegt. In diesem Fall tritt ein »device not present error« auf.

Wenn Sie wirklich professionell programmieren wollen, dürfen Sie eventuell auftretende Fehler selbstverständlich nicht einfach ignorieren, sondern benötigen eine sogenannte »Fehlerbehandlungsroutine«, zu der Ihr Programm in einem solchen Fall verzweigt (zum Beispiel, um den Benutzer aufzufordern, endlich die Floppy einzuschalten oder eine Diskette einzulegen). Übrigens: Da ich nicht von mir behaupte, professionell zu programmieren, habe ich es mir erlaubt, in der noch vorzustellenden INPUT#-Routine auf eine solche Fehlerbehandlung zu verzichten. Sollten Sie weniger bequem veranlagt sein, dürfen Sie das Programm jedoch gern entsprechend erweitern (und mir natürlich die erweiterte Version zuschicken).

Ob beim Öffnen einer Datei ein Fehler autrat, erkennen Sie am Carry-Flag. Bei fehlerfreiem Öffnen der Datei ist es nach der Rückkehr aus der OPEN-Routine gelöscht. Trat ein Fehler auf, ist das Carry-Flag gesetzt und im Akku wird die Nummer

des Fehlers übergeben. Bei gesetztem Carry-Flag sollte Ihr Programm daher zu einer Fehlerbehandlungsroutine verzweigen:

```
JSR $FFC0 ;OPEN AUFRUFEN
BCS $???? ;ADRESSE DER FEHLERBEHANDLUNGSROUTINE
```

Die Fehlernummern haben folgende Bedeutung:

0 = STOP-Taste gedrückt
1 = too many files
2 = file open
3 = file not open
4 = file not found
5 = device not present
6 = not input file
7 = not output file
8 = missing filename
9 = illegal device number

Die Fehler 3, 6 und 7 sind beim Öffnen einer Datei bedeutungslos. Sie können nur nach Lese- oder Schreibversuchen auftreten. Ein Fehler wie zum Beispiel »missing filename« ist dagegen gerätespezifisch. Bei Verwendung der Datasette müssen Sie nicht unbedingt einen Dateinamen angeben, daher kann auch dieser Fehler nicht auftreten.

CHKIN ($FFC6) und CKOUT ($FFC9): Ein-/Ausgaben umleiten

Ich habe behauptet, daß das Maschinenspracheäquivalent zum Basic-Befehl OPEN die Routinen SETFLS, SETNAM und der anschließende Aufruf der OPEN-Routine sei. Leider müssen wir noch zwei weitere Routinen besprechen, bevor wir zur eigentlichen Datenein- beziehungsweise -ausgabe kommen, die Routinen CHKIN und CKOUT.

Wie Sie wissen, ist die Tastatur beim C 16 das Standardeingabegerät und der Bildschirm das Standardausgabegerät. Um nach dem Öffnen einer Datei in diese Daten zu schreiben oder Daten daraus zu lesen, muß die Ein- beziehungsweise Ausgabe auf die geöffnete logische Datei umgelenkt werden, da sonst die Standardgeräte angesprochen werden.

Das Umlenken der Eingabe von der Tastatur auf die mit der OPEN-Routine geöffnete Datei erfolgt mit der Routine CHKIN, der im X-Register die beim Aufruf von SETFLS verwendete logische Filenummer übergeben wird, zum Beispiel die 1:

```
LDX #$01      ;LOGISCHE FILENUMMER
JSR $FFC6     ;AUFRUF VON CHKIN
```

Um in eine geöffnete Datei Daten zu schreiben, muß die Ausgabe vom Standardgerät Bildschirm zuvor auf die geöffnete logische Datei umgeleitet werden. Der zuständigen Routine CKOUT wird ebenfalls im X-Register die verwendete logische Filenummer übergeben:

```
LDX #$01      ;LOGISCHE FILENUMMER
JSR $FFC9     ;AUFRUF VON CKOUT
```

Beide Routinen zeigen auftretende Fehler ebenso wie die OPEN-Routine durch ein gesetztes Carry-Flag an. Trat ein Fehler auf, wird die Fehlernummer ebenfalls im Akku übergeben.

Demoprogramm 1: Datei in Maschinensprache öffnen

Wir sind nun endlich soweit, das theoretisch erworbene Wissen in ein Programm umzusetzen. Mit den besprochenen Routinen können wir ein Demonstrationsprogramm schreiben, das in Maschinensprache eine logische Datei zum Schreiben von Daten auf Band öffnet. Als Dateinamen verwenden wir »test«, als logische Filenummer eine Eins. Da eine Datasettendatei zum Schreiben geöffnet werden soll, müssen wir die Geräteadresse Eins (=Datasette) und die Sekundäradresse Eins (=Schreiben) verwenden.

Zum Schreiben der Daten verwenden wir ein kleines Basic-Programm, das zusätzlich die Aufgabe übernehmen soll, den Dateinamen »test« in ASCII-Form in den RS232-Puffer zu POKEn, bevor es unsere »Dateiöffnungsroutine« aufruft und anschließend Daten in die - durch die Maschinenroutine geöffnete - Datei schreibt:

```
100 REM *** DATEN SCHREIBEN ***
110 DN$="TEST":REM DATEINAME
120 FOR I=1 TO LEN(DN$)
130 : POKE 1014+I,ASC(MID$(DN$,I,1))
140 NEXT
150 SYS 1630:REM MASCHINENROUT.AUFRUFEN
160 PRINT#1,"DIES IST"
170 PRINT#1,"EIN TEST"
180 CLOSE 1
190 :
200 :
210 REM *** DATEN LESEN ***
220 PRINT"BAND BITTE ZURUECKSPULEN"
230 PRINT"DANACH 'RETURN' DRUECKEN"
240 GET A$:IF A$<>CHR$(13) THEN 240
250 SCNCLR:
260 :
270 OPEN 1,1,0,"TEST":REM DATEI OEFFNEN
280 INPUT#1,A$:PRINT A$
290 INPUT#1,B$:PRINT B$
300 CLOSE 1
```

Die Maschinenroutine geben Sie bitte mit dem im C 16 integrierten Monitor ab $065E (dezimal 1630) ein:

```
. 065e  LDA #$01    ;LOGISCHE FILENUMMER 1
. 0660  LDX #$01    ;GERAETEADRESSE 1 = DATASETTE
. 0662  LDY #$01    ;SEKUNDAERADRESSE 1 =
                    DATEN SCHREIBEN
. 0664  JSR $FFBA   ;SETFLS AUFRUFEN
. 0667  LDA #$04    ;LAENGE DES NAMENS: 4 ZEICHEN
. 0669  LDX #$F7    ;LOW-BYTE DER ADRESSE $03F7
. 066b  LDY #$03    ;HIGH-BYTE DER ADRESSE $03f7
. 066d  JSR $FFBD   ;SETNAM AUFRUFEN
. 0670  JSR $FFC0   ;OPEN AUFRUFEN
```

Tabelle der verwendeten Betriebssystem- und Interpreterroutinen

Name	Funktion	Parameter hin	Parameter zurück	Adresse
SETFLS	Fileparameter setzen	AKKU=LF; X=GA; Y=SA		$FFBA
SETNAM	Dateiname übertragen	AKKU=Länge; X/Y=Point. auf Name		$FFBD
OPEN	Log. Datei öffnen	Vorbereitung: SETFLS, SETNAM	Fehler: SEC, Fehlernr. im AKKU	$FFC0
CHKIN	Eingabe von Log. Datei	X=Log. Filenummer	Fehler: SEC, Fehlernr. im AKKU	$FFC6
CKOUT	Ausgabe auf Log. Datei	X=Log. Filenummer	Fehler: SEC, Fehlernr. im AKKU	$FFC9
READST	I/O-Status abfragen		Status im AKKU (BIT 6 = Dateiende)	$FFB7
BASIN	Zeichen einlesen		Zeichen im AKKU	$FFCF
BSOUT	Zeichen ausgeben	Zeichen im AKKU		$FFD2
CHKKOM	Komma lesen			$9491
GETBYT	Ein-Byte-Wert lesen		X=übergebenes Byte	$9D84
STRPOS	Variablenadresse holen		$47/$48=Point. auf Längendescr.	$96A5
STRRES	Stringplatz reservieren	AKKU=Stringlänge		$A906

```
. 0673  LDX #$01     ;LOGISCHE FILENUMMER
. 0675  JSR $FFC9    ;AUFRUF VON CKOUT
. 0678  RTS          ;ZURUECK ZUM BASIC
```

Das Maschinenprogramm besteht im Grunde nur aus einer Zusammenfassung der bereits beschriebenen Teile, das heißt aus dem Aufruf von SETFLS, SETNAM, OPEN und CKOUT. Beachten Sie bitte, daß eventuell auftretende Fehler ignoriert werden (Carry-Flag nach OPEN und CKOUT abfragen).

Geben Sie dieses Programm bitte mit dem Monitor ein (lassen Sie die Kommentare jedoch weg und versuchen Sie nicht, die einzelnen Programmteile wie abgebildet durch Leerzeilen optisch zu unterteilen!), geben Sie anschließend das Basic-Programm ein und starten Sie es mit

```
RUN
```

Zum Programmablauf: In den Zeilen 110 bis 140 werden mit einer Schleife die ASCII-Codes der einzelnen Zeichen des Dateinamens »test« nach 1015 bis 1018 gePOKEt. Anschließend wird unsere Maschinenroutine aufgerufen, die eine Datasettendatei für Schreibzugriffe mit der logischen Filenummer 1 öffnet (Zeile 150). Unter Angabe dieser Filenummer schreibt das Basic-Programm nun zwei Strings in die Datei, bevor sie wieder geschlossen wird (Zeilen 160 bis 180).

Vor dem Wiedereinlesen der Daten fordert Sie das Basic-Programm auf, zum Dateianfang zurückzuspulen und anschließend

```
RETURN
```

zu drücken. Nach RETURN wird die Warteschleife in Zeile 240 verlassen und der Bildschirm gelöscht, bevor die Datei »test« zum Lesen geöffnet wird (Zeilen 240 bis 270). Die beiden Strings werden eingelesen, auf dem Bildschirm ausgegeben und die Datei geschlossen.

Wie Sie sehen, ist es zweifellos bequemer, einen Basic-Befehl zum Öffnen einer Datei zu verwenden, als ein Maschinenprogramm, das an mehrere Routinen die verschiedensten Parameter übergeben muß, um die gleiche Funktion zu erfüllen. Die im Laufe dieses Kapitels vorgestellte INPUT#-Routine entschädigt Sie jedoch für diesen Aufwand. In Basic könnte diese Routine nur mit einem kleinen Unterprogramm unter Verwendung von GET# erstellt werden und wäre weitaus langsamer als die entsprechende Maschinenroutine. (Versuchen Sie einmal, 20 Strings mit einer Länge von jeweils 100 Zeichen mit GET# nach und nach einzulesen. Sie werden staunen, wie langsam Ihre Datasette plötzlich wird.)

BSOUT ($FFD2): Datenausgabe auf logische Datei

Wir können nun zwar in Maschinensprache beliebige Dateien öffnen, bisher jedoch weder Daten ausgeben noch einlesen. Sie alle wissen sicherlich, daß mit der Routine BSOUT Daten auf dem Bildschirm ausgegeben werden können. BSOUT gibt ein im Akku übergebenes Zeichen (ASCII-Code!) auf dem Bildschirm aus, und zwar an der aktuellen Cursorposition.

BSOUT ist jedoch keineswegs eine speziell zur Ausgabe auf dem Bildschirm gedachte Routine, sondern erfüllt die weitaus allgemeinere Funktion, Daten in eine beliebige logische Datei zu schreiben. Diese Datei ist normalerweise der Bildschirm, wir können mit BSOUT jedoch auch Zeichen in eine Datasetten-, Disketten- oder Druckerdatei schreiben.

Die benötigten Vorbereitungsroutinen kennen wir bereits. Zuerst wird eine logische Datei geöffnet, anschließend die Datenausgabe mit CKOUT auf diese Datei umgeleitet. Wenn nun BSOUT aufgerufen wird, werden die im Akku übergebenen Zeichen dank der Umleitung nicht auf dem Bildschirm ausgegeben, sondern in die angegebene Datei geschrieben.

```
...          ;DATASETTENDATEI ZUM
...          ;SCHREIBEN OEFFNEN
...          ;EINGABE AUF DIE DATEI LEGEN
LDA #$41     ;ASCII-CODE VON »A«
JSR $FFD2    ;»A« IN DIE DATEI SCHREIBEN
...
...
...
```

BASIN ($FFCF): Dateneingabe von logischer Datei

BASIN erfüllt die entgegengesetzte Funktion, das Einlesen von Daten aus einer beliebigen logischen Datei. Wenn die Zeichen nicht vom Standardgerät Tastatur gelesen werden sollen, muß ebenfalls eine logische Datei zum Lesen geöffnet und die Eingabe mit CHKIN auf diese Datei umgelenkt werden. Nun kann mit BASIN Zeichen für Zeichen aus dieser Datei gelesen werden. Die eingelesenen Zeichen werden im Akku übergeben.

```
...          ;DATASETTENDATEI ZUM
...          ;LESEN OEFFNEN
...          ;EINGABE AUF DIE DATEI LEGEN
JSR $FFCF    ;EIN ZEICHEN AUS DER DATEI LESEN
STA $xxx1    ;ZEICHEN SPEICHERN
JSR $FFCF    ;NAECHSTES ZEICHEN LESEN
STA $xxx2    ;EBENFALLS SPEICHERN
...
...
```

CLRCH ($FFCC): Standardein-/-ausgabegeräte setzen

Nach beendeter Ein- beziehungsweise Ausgabe von Daten sollte die Routine CLRCH aufgerufen werden, die die Eingabe wieder auf das Standardgerät Tastatur und die Ausgabe auf den Bildschirm umleitet. Es werden keinerlei Vorbereitungsroutinen oder Übergabeparameter benötigt.

```
...          ;LOGISCHE DATEI OEFFNEN
...          ;EIN- ODER AUSGABE UMLENKEN
...          ;DATEN LESEN ODER SCHREIBEN
JSR $FFCC    ;STANDARDEIN-/-AUSGABEGERAETE SETZEN
...          ;(TASTATUR/BILDSCHIRM)
...
...
```

CLOSE ($FFC3): Logische Datei schließen

CLRCH leitet weitere Ein- oder Ausgaben zwar wieder auf die Tastatur beziehungsweise den Bildschirm um, die zuvor geöffnete logische Datei wird jedoch nicht automatisch geschlossen! Zum Abschluß jeder Schreib- oder Leseroutine sollte daher die Routine CLOSE aufgerufen werden, die die angegebene logische Datei schließt, wobei die Filenummer im Akku übergeben werden muß.

```
...          ;DATEI MIT LOGISCHER FILENUMMER 1 OEFFNEN
...          ;EIN-/AUSGABE UMLEITEN
...          ;DATEN LESEN ODER SCHREIBEN
JSR $FFCC    ;EIN-/AUSGABE AUF STANDARDGERAETE
LDA #$01     ;LOGISCHE FILENUMMER
JSR $FFC3    ;LOGISCHE DATEI SCHLIESSEN
...
...
...
```

READST ($FFB7): Status abfragen

Ein Problem müssen wir noch lösen, bevor wir ausschließlich in Maschinensprache Dateiverwaltung betreiben können, und zwar die Abfrage des Dateiendes. Das Einlesen einer Datei muß beendet werden, wenn das Dateiende erreicht wurde. In Basic können wir dazu die Statusvariable ST abfragen, die den Wert 64 annimmt, wenn das Dateiende erreicht wurde.

In Maschinensprache verwenden wir die Routine READST, die die gleiche Funktion erfüllt (der Basic-Interpreter verwendet diese Routine selbst, um der Variablen ST den jeweiligen Status mitzuteilen).

Nach dem Aufruf von READST wird der aktuelle Status im Akku übergeben. Bei Erreichen des Dateiendes wird Bit 6 gesetzt (da dieses Bit den dezimalen Wert 64 besitzt, wird

die Analogie zur Basic-Variablen ST deutlich).

```
...
...
...
LABEL JSR $FFCF ;DATEN LESEN
      JSR $FFB7 ;READST AUFRUFEN
      AND #$40 ;ALLE BITS AUSSER BIT 6
               AUSMASKIEREN
      BEQ LABEL ;WEITERLESEN, WENN NICHT DATEIENDE
...
...
...
```

```
. 065e  a9 ff     lda #$ff
. 0660  20 06 a9  jsr $a906
. 0663  20 91 94  jsr $9491
. 0666  20 84 9d  jsr $9d84
. 0669  20 c6 ff  jsr $ffc6
. 066c  a0 00     ldy #$00
. 066e  20 cf ff  jsr $ffcf
. 0671  c9 0d     cmp #$0d
. 0673  f0 05     beq $067a
. 0675  91 33     sta ($33),y
. 0677  c8        iny
. 0678  d0 f4     bne $066e
. 067a  84 d0     sty $d0
. 067c  20 cc ff  jsr $ffcc
. 067f  a5 34     lda $34
. 0681  85 d2     sta $d2
. 0683  a9 ff     lda #$ff
. 0685  38        sec
. 0686  e5 d0     sbc $d0
. 0688  18        clc
. 0689  65 33     adc $33
. 068b  85 d1     sta $d1
. 068d  90 02     bcc $0691
. 068f  e6 d2     inc $d2
. 0691  a4 d0     ldy $d0
. 0693  88        dey
. 0694  b1 33     lda ($33),y
. 0696  91 d1     sta ($d1),y
. 0698  88        dey
. 0699  c0 ff     cpy #$ff
. 069b  d0 f7     bne $0694
. 069d  20 91 94  jsr $9491
. 06a0  20 a5 96  jsr $96a5
. 06a3  a0 02     ldy #$02
. 06a5  b1 47     lda ($47),y
. 06a7  99 d3 00  sta $00d3,y
. 06aa  88        dey
. 06ab  10 f8     bpl $06a5
. 06ad  a5 d3     lda $d3
. 06af  f0 08     beq $06b9
. 06b1  a8        tay
. 06b2  91 d4     sta ($d4),y
. 06b4  c8        iny
. 06b5  a9 ff     lda #$ff
. 06b7  91 d4     sta ($d4),y
. 06b9  a4 d0     ldy $d0
. 06bb  a5 47     lda $47
. 06bd  91 d1     sta ($d1),y
. 06bf  c8        iny
. 06c0  a5 48     lda $48
. 06c2  91 d1     sta ($d1),y
. 06c4  a0 00     ldy #$00
. 06c6  a5 d0     lda $d0
. 06c8  91 47     sta ($47),y
. 06ca  c8        iny
. 06cb  a5 d1     lda $d1
. 06cd  91 47     sta ($47),y
. 06cf  85 33     sta $33
. 06d1  c8        iny
. 06d2  a5 d2     lda $d2
. 06d4  91 47     sta ($47),y
. 06d6  85 34     sta $34
. 06d8  60        rts
```

Listing 1. »Input-Routine« in Maschinensprache. Bitte mit dem eingebauten Monitor eingeben.

Demoprogramm 2: Daten in Maschinensprache schreiben und lesen

Die Zeiten, in denen wir uns zum Schreiben und Lesen von Daten mit Basic behelfen mußten, sind nun endgültig vorbei. In diesem zweiten Programmbeispiel werden wir die Strings »DIES IST« und »EIN TEST« in Maschinensprache auf Band schreiben und wieder lesen.

Aus reiner Bequemlichkeit wird zusätzlich ein kleines Basic-Programm verwendet, das den Dateinamen »TEST« ab 1015 und die zu schreibenden Strings ab 1019 in den Speicher POKEt, die Routine zum Schreiben der Strings aufruft, danach die Meldung »ZURUECKSPULEN...« ausgibt, auf die Taste RETURN wartet und zuletzt die Leseroutine aufruft, die beide Strings aus der Datei einliest und auf dem Bildschirm ausgibt.

```
100 REM *** STRINGS VORBEREITEN ***
110 FOR I=1 TO 4:POKE 1014+I,ASC(MID$("TEST",I,1))
    :NEXT
120 A$="DIES IST"+CHR$(13)+"EIN TEST"+CHR$(13)
    +CHR$(0)
130 FOR I=1 TO LEN(A$):POKE 1018+I,ASC
    (MID$(A$,I,1)):NEXT
140 :
150 REM *** STRINGS SCHREIBEN/LESEN ***
160 SYS 1630:REM STRINGS SCHREIBEN
170 PRINT"ZURUECKSPULEN + 'RETURN' DRUECKEN"
180 GET A$:IF A$<>CHR$(13) THEN 180
190 SYS 1678:REM STRINGS LESEN/AUSGEBEN
```

Beachten Sie bitte, daß bei der zeichenweisen Ausgabe mit BSOUT im Gegensatz zur Basic-Ausgabe mit PRINT# nicht automatisch nach jedem String das Zeichen CHR$(13) (Carriage Return oder auch Zeilenvorschub) auf das Band geschrieben wird. Die Basic-Routine fügt dieses Zeichen daher an das Ende beider Zeichenketten an (Zeile 120). Als letztes Zeichen wird ein CHR$(0) als Endemarke angehängt, an dem unsere Maschinenroutine das Ende des auszugebenden Strings erkennen soll.

Teil 1: Daten schreiben

```
. 065e LDA #$01   ;LOGISCHE FILENUMMER
. 0660 LDX #$01   ;GERAETEADRESSE
. 0662 LDY #$01   ;SEKUNDAERADRESSE (1=WRITE)
. 0664 JSR $ffBA  ;SETLFS AUFRUFEN

. 0667 LDA #$04   ;LAENGE DES DATEINAMENS
. 0669 LDX #$f7   ;LOW-BYTE ADRESSE DATEINAME
. 066b LDY #$03   ;HIGH-BYTE ADRESSE DATEINAME
. 066d JSR $FFBD  ;SETNAM AUFRUFEN

. 0670 JSR $FFC0  ;OPEN AUFRUFEN
. 0673 LDX #$01   ;LOGISCHE FILENUMMER
. 0675 JSR $FFC9  ;CKOUT AUFRUFEN

. 0678 LDX #$00   ;ZEIGER INITIALISIEREN
. 067A LDA $03FB,X ;ERSTES ZEICHEN HOLEN
. 067D BEQ $0685  ;FERTIG, WENN ENDEMARKE
. 067f JSR $FFD2  ;ZEICHEN IN DATEI SCHREIBEN
. 0682 INX        ;ZEIGER AUF NAECHSTES ZEICHEN
. 0683 BNE $067A  ;ZUM SCHLEIFENANFANG (IMMER!)

. 0685 JSR $FFCC  ;STANDARDGERAETE SETZEN
. 0688 LDA #$01   ;LOGISCHE FILENUMMER
. 068a JSR $FFC3  ;LOGISCHE DATEI SCHLIESSEN
. 068d RTS        ;RUECKKEHR ZUM BASIC
```

Teil 2: Daten lesen/ausgeben

```
. 068e LDA #$01   ;LOGISCHE FILENUMMER
. 0690 LDX #$01   ;GERAETEADRESSE
```

```
. 0692 LDY #$00    ;SEKUNDAERADRESSE (0=READ)
. 0694 JSR $FFBA   ;SETLFS AUFRUFEN

. 0697 LDA #$04    ;LAENGE DES DATEINAMENS
. 0699 LDX #$F7    ;LOW-BYTE ADRESSE DATEINAME
. 069b LDY #$03    ;HIGH-BYTE ADRESSE DATEINAME
. 069D JSR $FFBD   ;SETNAM AUFRUFEN

. 06A0 JSR $FFC0   ;OPEN AUFRUFEN
. 06A3 LDX #$01    ;LOGISCHE FILENUMMER
. 06A5 JSR $FFC6   ;CHKIN AUFRUFEN

. 06A8 JSR $FFCF   ;BASIN AUFRUFEN
. 06AB JSR $FFD2   ;ZEICHEN AUSGEBEN
. 06AE JSR $FFB7   ;READST AUFRUFEN
. 06B1 AND #  $40 ;ALLE AUSSER BIT 6 AUSMASKIEREN
. 06B3 BEQ $06A8   ;WEITER, WENN NICHT DATEIENDE

. 06B5 JSR $FFCC   ;STANDARDGERAETE SETZEN
. 06B8 LDA #  $01 ;LOGISCHE FILENUMMER
. 06BA JSR $FFC3   ;LOGISCHE DATEI SCHLIESSEN
. 06BD RTS         ;RUECKKEHR ZUM BASIC
```

Dieses Programm dürfte gut verständlich sein, da es unter Verzicht auf Programmkürze in aller Ausführlichkeit programmiert wurde. Mit dem Öffnen und Schließen der Dateien im ersten und zweiten Programmteil sollten Sie nun vertraut sein.

Wichtig an diesem Programm ist, daß im ersten Teil Zeichen für Zeichen in die Datei geschrieben werden, bis das nächste Zeichen eine Null ist, jene Endemarke, die das Basic-Programm am Ende des Strings in den Speicher POKEte.

Beim Lesen der Zeichen könnte ebenfalls eine solche Endemarke benutzt werden, um das Dateiende festzustellen. Statt dessen wird die erläuterte Routine READST verwendet. Zeichen für Zeichen werden aus der Datei gelesen und auf dem Bildschirm ausgegeben, wobei nach jedem Lesevorgang READST aufgerufen und das sechste Bit des im Akku übergebenen Status-Bytes überprüft wird (AND #$40). Ist dieses Bit gesetzt, wurde das Dateiende erreicht und das Programm kehrt zum Basic zurück (zuvor werden noch CLRCH und CLOSE aufgerufen).

Die Interpreterroutinen CHKKOM, GETBYT, STRPOS und STRRES

Wie ich es Ihnen versprochen habe, will ich im letzten Teil dieses Artikels unsere frisch erworbenen Kentnisse dazu benutzen, eine verbesserte Version der INPUT#-Routine vorzustellen.

Vorher muß ich jedoch kurz auf verschiedene Routinen des Basic-Interpreters eingehen, die für unsere Routine unbedingt benötigt werden. Da der Aufruf dem normalen INPUT#-Befehl entsprechen soll (INPUT#(logische Filenummer), (Übergabestring)), muß unser Programm in der Lage sein, die Filenummer aus dem Basic-Text zu lesen und auf den angegebenen String zuzugreifen, in dem die eingelesenen Daten gespeichert werden sollen.

Unsere Routine wird im RAM-Bereich $065E bis $06EB (dezimal 1630 bis 1771) liegen und wie folgt aufgerufen:

```
SYS 1630,(LOGISCHE FILENUMMER),(ÜBERGABESTRING)
```

zum Beispiel:

```
SYS 1630,1,a$
```

oder

```
SYS 1630,2,X$(7)
```

Da die einzelnen Parameter durch Kommata getrennt sind, benötigen wir zuerst eine Routine, die in der Lage ist, Kommatas aus dem Basic-Programm einzulesen. Diese Routine wird CHKKOM ($9491) genannt. Der Basic-Interpreter führt ständig einen Zeiger auf die momentan bearbeitete Textstelle mit sich. Nach dem SYS-Aufruf weist dieser Zeiger auf das erste Zeichen hinter der angegebenen Adresse, das heißt auf das Komma.

```
10 DATA A9,FF,20,06,A9,20,91,94,20,84
20 DATA 9D,20,C6,FF,A0,00,20,CF,FF,C9
30 DATA 0D,F0,05,91,33,C8,D0,F4,84,D0
40 DATA 20,CC,FF,A5,34,85,D2,A9,FF,38
50 DATA E5,D0,18,65,33,85,D1,90,02,E6
60 DATA D2,A4,D0,88,B1,33,91,D1,88,C0
70 DATA FF,D0,F7,20,91,94,20,A5,96,A0
80 DATA 02,B1,47,99,D3,00,88,10,F8,A5
90 DATA D3,F0,08,A8,91,D4,C8,A9,FF,91
100 DATA D4,A4,D0,A5,47,91,D1,C8,A5,48
110 DATA 91,D1,A0,00,A5,D0,91,47,C8,A5
120 DATA D1,91,47,85,33,C8,A5,D2,91,47
130 DATA 85,34,60
140 FORI=1630TO1752
150 READA$
160 A=DEC(A$)
170 S=S+A
180 POKEI,A
190 NEXT
200 IFS<>17787THENPRINT"FEHLER!":END
210 PRINT"OK."

READY.
```

Listing 2. DATA-Lader zur »Input-Routine«

Der Aufruf von CHKKOM liest dieses Zeichen ein, setzt den »Textpointer« auf das nächste Zeichen im Basic-Programm und überprüft, ob tatsächlich ein Komma gelesen wurde. Wenn ja, endet CHKKOM mit einem RTS-Befehl, sonst wird ein »Syntax error in ...« ausgegeben.

Wenn mit CHKKOM das erste Trennzeichen eingelesen wurde, muß anschließend die angegebene Filenummer von unserem Programm gelesen werden. Die Routine GETBYT ($9D84) liest einen beliebigen Ein-Byte-Wert aus dem Basic-Text und übergibt ihn im X-Register. Der Wert muß übrigens nicht direkt als Zahl angegeben werden, da GETBYT auch Variable verarbeitet, so daß zum Beispiel folgender Aufruf möglich ist:

```
LF=1:SYS 1630,LF,A$
```

Bevor wir an die eigentliche Programmerstellung gehen können, muß ein weiteres Problem gelöst werden: Wo speichern wir die eingelesenen Zeichen? In welcher Form sollen die eingelesenen Daten im Speicher abgelegt werden? Am komfortabelsten ist die Benutzung unserer Routine, wenn sie die eingelesenen Zeichen als String anlegt, auf den das aufrufende Basic-Programm direkt zugreifen kann. Der Name des Übergabestrings sollte ebenso wie beim INPUT#-Befehl beim Aufruf angegeben werden können.

Ein Standardproblem bei der Verbindung von Basic und Maschinenroutinen ist der Zugriff auf Basic-Strings von Maschinensprache aus, das Lesen oder gar Anlegen von Strings. Beides ermöglicht uns die Routine STRPOS ($96A5). Diese Routine liest den beim Aufruf angegebenen Stringnamen aus dem Basic-Text und übergibt in $47 und $48 einen Pointer auf die sogenannten »Stringdescriptoren«.

Wie Sie vielleicht wissen, werden Strings vom Ende des verfügbaren Speichers aus abwärts angelegt und befinden sich im Gegensatz zu numerischen Variablen nicht direkt in der sogenannten »Variablentabelle«, die im Speicher immer unmittelbar dem Basic-Programm folgt. In der Variablentabelle befinden sich jedoch alle Daten, die zum Zugriff auf den eigentlichen String notwendig sind, der Stringname, die Stringlänge und die Adresse, an der sich der String selbst

befindet (Name: 2 Byte; Länge: 1 Byte; Adresse: 2 Byte in der Form Low/High).

STRPOS übergibt einen Zeiger auf diese Descriptoren des angegebenen Strings (und zwar nicht auf das erste Descriptoren-Byte (das erste Namens-Byte), sondern auf das Längen-Byte) und legt ihn an, wenn er bisher noch nicht existiert. Wenn der Aufruf zum Beispiel lautet:

```
SYS 1630,A$(7)
```

und unsere Routine zuerst CHKKOM und anschließend STRPOS aufruft, erhalten wir in $47 und $48 einen Pointer, der auf den Längendescriptor des Strings A$(7) zeigt.

Die letzte benötigte Routine heißt STRRES ($A906). Diese Routine reserviert Speicherplatz für den anzulegenden String. Ein Beispiel: Wir wollen einen String mit einer Länge von 10 Zeichen anlegen. Zuerst muß der Akku mit der Stringlänge geladen werden, anschließend wird STRRES aufgerufen:

```
LDA #$0A     ;STRINGLAENGE: 10 ZEICHEN
JSR $A906    ;AUFRUF VON STRRES
```

In $33 und $34 befindet sich ein Pointer auf den Anfang des sogenannten »Stringstacks«, das heißt auf das erste Zeichen des zuletzt angelegten Strings. Nach dem Aufruf von STRRES mit der Stringlänge Zehn im Akku wird dieser Pointer um den Wert Zehn (+2, wie wir noch sehen werden!) erniedrigt. Der dazukommende String kann nun vor (!) dem zuletzt angelegten gespeichert werden (denken Sie bitte daran, daß die Strings vom Speicherende ausgehend abwärts (!) angelegt werden).

Bevor STRRES den Pointer auf den Anfang der Strings um die übergebene Stringlänge (+2) vermindert, wird jedoch überprüft, ob überhaupt ausreichend Platz vorhanden ist, oder ob der nach unten wachsende Stringstack mit dem Basic-Programm kollidieren und dieses überschreiben würde. Wenn nicht ausreichend Platz vorhanden ist, wird eine Garbage Collection durchgeführt, das heißt nicht mehr benötigter Stringmüll beseitigt. Ist auch anschließend noch nicht ausreichend Platz für den anzulegenden String vorhanden, gibt STRRES die Fehlermeldung »out of memory error« aus.

Endlich: Die INPUT#-Routine

Nun können wir uns endlich an die eigentliche Programmierung wagen. Im Gegensatz zu den bisher verwendeten Monitorlistings werde ich die Entwicklung der INPUT#-Routine anhand von Assemblerlistings demonstrieren, da die Routine doch etwas komplexer als die bisherigen Programmbeispiele ist und Assemblerlistings einfacher zu durchschauen sind. Da Sie jedoch kaum über einen Assembler für den C 16 verfügen, finden Sie am Ende dieses Artikels wieder das gewohnte Monitorlisting, das Sie problemlos mit dem integrierten Monitor eingeben können.

```
;*************************
;* INPUT# FUER C16/C116 *
;* (C) SAID BALOUI, 1986 *
;*************************
        .BA $065E     ;PROGRAMMSTART
        .OS           ;OBJECTCODE GENERIEREN
;
LENGTH  .DE $D0       ;STRINGLAENGE
POINTR  .DE $D1       ;HILFSPOINTER
DESCR   .DE $D3       ;STRINGDESCRIPTOREN
DESPOI  .DE $47       ;POINTER AUF DESCRIPTOREN
CHKKOM  .DE $9491     ;KOMMA EINLESEN
GETBYT  .DE $9D84     ;BYTEWERT LESEN
CHKIN   .DE $FFC6     ;INPUT AUF LOGISCHE DATEI
CLRCH   .DE $FFCC     ;STANDARDGERAETE SETZEN
BASIN   .DE $FFCF     ;ZEICHEN AUS LOG.DATEI LESEN
STREND  .DE $33       ;POINTER AUF STRINGSTACK
STRPOS  .DE $96A5     ;STRINGVARIABLE LESEN
STRRES  .DE $A906     ;STRINGPLATZ RESERVIEREN
;
*** PLATZ RESERVIEREN ***
     LDA #255     ;255 BYTE
     JSR STRRES   ;RESERVIEREN
```

Diese beiden Befehle reservieren Platz für den einzulesenden String mit einer Maximallänge von 255 Zeichen. Dank der Verwendung von STRRES wird sich das Programm keinesfalls »aufhängen«. Sollte das Basic-Programm, in dem Sie die INPUT#-Routine verwenden, zu umfangreich und daher nicht ausreichend Platz vorhanden sein, wird es sich mit »out of memory error« verabschieden.

```
*** STRING EINLESEN ***
        JSR CHKKOM      ;LOGISCHE FILENUMMER
        JSR GETBYT      ;EINLESEN (X-REGISTER)
        JSR CHKIN       ;EINGABE AUF LOGISCHE DATEI
        LDY #0
INPUT   JSR BASIN
        CMP #13         ;LESEN BIS >>RETURN<<
        BEQ LIESEND     ;UND AB STREND/STREND+1
        STA (STREND),Y  ;IM STRINGSTACK ABLEGEN
        INY
        BNE INPUT       ;UNBEDINGTER SPRUNG (!)
LIESEND STY LENGTH      ;STRINGLAENGE MERKEN
        JSR CLRCH       ;STANDARDGERAETE SETZEN
```

Der Programmteil zum Einlesen des Strings liest zuerst das als Trennzeichen verwendete Komma und die angegebene Filenummer aus dem Basic-Text ein. GETBYT übergibt wie erwähnt die Filenummer im X-Register, in dem sich die Filenummer auch beim Aufruf von CHKIN befinden muß, so daß diese Routine ohne weitere Vorbereitung aufgerufen und die Eingabe auf die geöffnete Datei umgelenkt werden kann.

INPUT verbessert

Das Y-Register wird nun mit Null initialisiert und Zeichen für Zeichen aus der Datei eingelesen und ab dem neuen Anfang des Stringstacks gespeichert, bis das Zeichen RETURN gelesen wird, das das Stringende kennzeichnet. Die in Y enthaltene Stringlänge wird in LENGTH zur späteren Verwendung zwischengespeichert. Durch Aufruf von CLRCH werden wieder die Standardgeräte (Tastatur, Bildschirm) gesetzt.

```
*** POINTER AUF STRINGANFANG ***
     LDA STREND+1    ;POINTR(+1)=
     STA POINTR+1    ;ECHTER STRINGANFANG,
     LDA #255        ;D.H. MOMENTANER
     SEC             ;ANFANG (STREND)
     SBC LENGTH      ;+ DIE DIFFERENZ
     CLC             ;ZWISCHEN DER
     ADC STREND      ;RESERVIERTEN LAENGE
     STA POINTR      ;255 ZEICHEN UND
     BCC NOINC       ;DER TATSAECHLICHEN
     INC POINTR+1    ;LAENGE DES STRINGS
```

Die eingelesenen Zeichen wurden ab der Adresse gespeichert, auf die STREND weist. STREND weist durch die Stringreservierung 255 Zeichen (+2) vor den Beginn des nächsten Strings im Stringstack. Um die unglaubliche Platzverschwendung zu vermeiden, wenn zwar 255 Zeichen reserviert wurden, der eingelesene String tatsächlich jedoch zum Beispiel nur 20 oder 30 Zeichen lang ist, wird nach beendetem Einlesen der String nach oben verschoben, um den Leerraum zwischen dem eingelesenen und dem nächsten String im Stringstack wiederzugewinnen. Dazu wird zuerst anhand der tatsächlichen Stringlänge LENGTH und der daraus resultierenden Differenz zum reservierten Speicherplatz ein Pointer POINTR errechnet, der auf die korrekte Adresse weist, an die der String verschoben werden muß.

```
;*** STRING VERSCHIEBEN ***
NOINC   LDY LENGTH      ;STRING VON:
        DEY             ;STREND BIS STREND+LENGTH
COPY    LDA (STREND),Y  ;NACH:
        STA (POINTR),Y  ;POINTR BIS POINTR+LENGTH
        DEY             ;KOPIEREN
        CPY #255
        BNE COPY
```

Das zeichenweise Kopieren des Strings dürfte problemlos zu verstehen sein. Der String befindet sich an der korrekten Adresse und unsere Routine wäre beendet, doch leider kann das Basic-Programm noch nicht auf den angelegten String zugreifen. Zuvor müssen noch mehrere Pointer korrigiert werden.

```
;*** STRINGDESCRIPTOREN HOLEN ***
        JSR CHKKOM      ;POINTER AUF DIE
        JSR STRPOS      ;DESCRIPTOREN DES
        LDY #2          ;BEIM AUFRUF ANGEGEBENEN
DESCOPY LDA (DESPOI),Y  ;STRINGS HOLEN ($47 UND $48)
        STA DESCR,Y     ;UND DIE DESCRIPTOREN
        DEY             ;(LAENGE/ADRESSE(LOW/HIGH))
        BPL DESCOPY     ;NACH DESCR BIS DESCR+2
```

Mit CHKKOM wird das der Filenummer folgende Komma gelesen und STRPOS übergibt in DESPOI ($47 und $48) einen Pointer auf den beim Aufruf angegebenen String. Die Descriptoren dieses Strings werden in einer Schleife nach DESCR (Länge), DESCR+1 (Adresse Low) und DESCR+2 (Adresse high) kopiert.

```
;*** RUECKPOINTER BEHANDELN ***
        LDA DESCR       ;WENN ALTER STRING
                        EXISTIERT
        BEQ ANLEG       ;(DAS HEISST WENN LAENGE
                        <>0),
        TAY             ;WIRD ER UNGUELTIG GEMACHT
        STA (DESCR+1),Y ;(RUECKPOINTER=LAENGE,
        INY             ;RUECKPOINTER+1=$FF)
        LDA #$FF
        STA (DESCR+1),Y
;
ANLEG   LDY LENGTH      ;RUECKPOINTER/RUECKPOINTER
                        +1
        LDA DESPOI      ;WIRD DIE ADRESSE
        STA (POINTR),Y  ;DES LAENGENDESCRIPTORS
        INY             ;ZUGEWIESEN
        LDA DESPOI+1
        STA (POINTR),Y
```

Diesen Programmteil können Sie nicht verstehen, ohne zu wissen, daß der Basic-Interpreter des C16 am Ende jedes Strings einen Pointer anlegt, der auf den zugehörigen Stringdescriptor, genauer: auf den Längendescriptor des Strings, »zurückweist«. Das heißt, daß nicht nur ein Pointer in der Variablentabelle (die beiden Bytes des Adressendescriptors) auf den String im Stringstack weist, sondern ein weiterer Pointer von diesem String auf die Descriptoren dieses Strings zurückweist. Dieser zusätzliche Pointer hat nur für die Garbage Collection eine Bedeutung, die dadurch erheblich beschleunigt wird.

Entscheidend für uns ist jedoch, daß unsere Routine außer dem eigentlichen String ebenfalls diesen »Rückdescriptor« anlegen muß. Platz dafür ist am Stringende vorhanden, da STRRES den Pointer STREND nicht nur um die angegebene Stringlänge vermindert, sondern - wie bereits von mir angedeutet wurde - um zusätzliche zwei Byte für den ebenfalls anzulegenden Rückdescriptor.

Die Routine »Rückpointer behandeln« prüft zuerst, ob bereits ein String unter dem angegebenen Namen existierte, das heißt ob der Längendescriptor nicht gleich Null ist. Ist dies der Fall, muß der alte String ungültig gemacht werden, da der Basic-Interpreter sonst bei der nächsten Garbage Collection durcheinanderkommt, weil das Low-Byte des Rückpointers die alte Stringlänge und das High-Byte den Wert $FF als Kennzeichen für einen ungültigen String enthält.

Anschließend enthält der neu anzulegende Rückpointer die Adresse der zugehörigen Descriptoren. Er muß auf den Längendescriptor weisen.

```
;*** DESCRIPTOREN/STREND BEHANDELN ***
    LDY #0          ;DESCRIPTOREN AKTUALISIEREN
    LDA LENGTH      ;(LAENGENDESCRIPTOR MIT DER
    STA (DESPOI),Y  ;STRINGLAENGE UND ADRESSEN-
    INY             ;DESCRIPTOREN MIT DER STRING-
    LDA POINTR      ;ADRESSE VERSEHEN
    STA (DESPOI),Y  ;AUSSERDEM STREND/STREND+1
    STA STREND      ;ENTSPRECHEND DER GEAENDERTEN
    INY             ;STRINGADRESSE KORRIGIEREN
    LDA POINTR+1
    STA (DESPOI),Y
    STA STREND+1
    RTS             ;ZURUECK ZUM BASIC
```

Dem Längendescriptor des Strings wird nun die Länge LENGTH und den beiden Adressendescriptoren (Low/High) die geänderte Startadresse POINTR/POINTR+1 zugewiesen. Da der von den reservierten 255 Zeichen unbenutzte Bereich durch das Verschieben des Strings wieder freigegeben wurde, muß natürlich auch STREND/STREND+1 entsprechend der endgültigen Stringadresse korrigiert werden.

Alle Pointer und Descriptoren sind nun völlig korrekt gesetzt und das Basic-Programm kann auf den angelegten String zugreifen, wie das folgende Demoprogramm zeigt:

```
100 REM *** STRING ERZEUGEN ***
110 X$="123456789,"
120 FOR I=1 TO 25:Y$=Y$+X$:NEXT
130 Y$=Y$+"12345"
140 :
150 REM *** STRING SCHREIBEN ***
160 OPEN 1,1,1,"TEST"
170 PRINT#2,Y$
180 CLOSE 2
190 :
200 REM *** STRING LESEN ***
210 OPEN 1,1,0,"TEST"
220 SYS 1630,1,A$
230 CLOSE 2
240 PRINT A$
```

Dieses Demoprogramm schreibt einen String mit einer Maximallänge von 255 Zeichen inklusive dem mit INPUT# keinesfalls einlesbaren Zeichen »,« auf Band und liest ihn mit der vorgestellten Routine in der vollen Länge mit Kommata wieder ein, was für den INPUT#-Befehl völlig unmöglich wäre. Das Basic-Programm kann auf den eingelesenen String wie auf jeden anderen String mit PRINT A$ zugreifen.

Wir sind nun am Ende dieses Artikels angelangt. Nachfolgend finden Sie das Monitorlisting (Listing 1) der Routine, den entsprechenden Datalader (Listing 2) und eine Tabelle, in der alle verwendeten Routinen des Betriebssystems und des Basic-Interpreters aufgeführt sind.

Verzeihen Sie mir bitte, daß die Erläuterungen der Routinen zum Anlegen des Strings, zur Behandlung der Descriptoren und des Rückdescriptors ziemlich knapp ausfielen, aber der Schwerpunkt dieses Artikels war nun einmal nicht der Basic-Interpreter des C16 und dessen Variablenbehandlung, sondern es sollte gezeigt werden, wie Sie auch in Maschinensprache Daten an beliebige Peripheriegeräten senden oder von diesen empfangen können. (S. Baloui/ah)