MPBASIC

REIHE AUTOMATISIERUNGSTECHNIK Band 216

Programmieren mit BASIC Siegmar Müller

1. Auflage © VEB Verlag Technik, Berlin, 1985

S. 60 ff.

7. Tiny MPBASIC

7.1. Tiny BASIC

Tiny[1)] BASIC ist eine aus BASIC abgeleitete, sehr stark vereinfachte Programmiersprache. Die Vereinfachungen bestehen im wesentlichen in folgenden Punkten:

1. Es sind nur die Variablen A ... Z zugelassen.
2. Die einzige Datenart sind ganze Zahlen.
3. Es gibt keine Felder.
4. Es gibt keine Strings.
5. Tiny BASIC hat weniger Anweisungen.
6. Es fehlen eine Reihe von numerischen Funktionen.

Diese Vereinfachungen gestatten es, Interpreter für Tiny BASIC mit sehr geringem Speicherbedarf zu schreiben. Der Speicherplatz für einen „normalen„ BASIC-Interpreter liegt zwischen 8 bis über 20 KByte, während man Interpreter für Tiny BASIC in 2 ... 4 KByte unterbringt. Die ausschließliche Verwendung ganzzahliger Variablen und Konstanten ergibt dazu eine verhältnismäßig schnelle Arithmetik. Diese beiden Vorteile stehen dem Nachteil des geringeren Komforts gegenüber. Damit ergeben sich für Tiny BASIC andere Anwendungsbereiche als für ein „großes“ BASIC. Man setzt diese Sprache in sehr kleinen Rechnern mit wenig Speicherplatz zum Steuern und Regeln ein. Dabei ist man durchaus in der Lage, auch Echtzeit Probleme zu lösen.

Ein Echtzeitproblem besteht dann, wenn der Rechner auf ein externes Ereignis innerhalb einer vorgegebenen Zeit in einer diesem Ereignis entsprechenden Art und Weise reagieren muß. Ob eine Sprache mit einem gegebenen Rechner echtzeitfähig ist, hängt also auch von dem zu lösenden Problem ab. Wenn es um Zehntelsekunden geht, dann ist Tiny BASIC oft schon einsetzbar, während man im Mikrosekundenbereich nur mit Maschinenprogrammen Herr der Lage bleiben kann.

Ein generelles Problem bei der Anwendung höherer Programmiersprachen in Echtzeitproblemen besteht darin, daß es nicht möglich ist, wie bei der Assemblerprogrammierung, die Abarbeitungszeit exakt zu berechnen, In der Praxis genügt es aber fast immer, diese Zeit zu messen. Dabei sollte man jedoch genau überlegen, mit welchen Eingangsdaten man bei dem zu testenden Programmteil den ungünstigsten Fall erzeugt. Dadurch wird sichergestellt, daß sich nicht bei anderen Daten eine längere Abarbeitungszeit ergibt. Zum Messen genügt meist schon die Armbanduhr, wenn man das zu testende Programmstück in eine Schleife faßt. Zum Beispiel:

```
10 FOR I=1 TO 1000
```

```
programmstück
1000 NEXT I
```

Mißt man hier x Sekunden, dann dauert die Abarbeitung des Programmstücks etwas weniger als x Millisekunden. (Weniger, weil FOR und NEXT noch hinzu kommen)

In der Praxis sind jedoch meist nur wenige Programmstellen zeitkritisch, und Tiny BASIC läßt hier noch die Möglichkeit offen, auf die Maschinenebene auszuweichen (CALL, Interrupts im Hintergrund), wenn das eben Vorgeschlagene nicht zu befriedigenden Ergebnissen führt. Hat man nach eingehender Prüfung entschieden, Tiny BASIC einzusetzen, dann wird man wesentlich schneller zu einem fertigen Programm kommen als mit dem Assembler.

7.2. Konzept von Tiny MPBASIC

Wie bei BASIC gibt es auch bei Tiny BASIC eine kaum überschaubare Vielzahl von Versionen. Tiny MPBASIC (winziges Mikroprozessor-BASIC) wurde ursprünglich für den Einchipmikrorechner U883 geschrieben, kann aber prinzipiell auch für jeden anderen Prozessor implementiert werden. Zum Verständnis der folgenden Ausführungen sind dennoch einige Grundkenntnisse über den Aufbau und die Programmierung des Einchipmikrorechners U881 erforderlich. Hierzu sei z. B. auf [11] verwiesen. Wegen des begrenzten Programmspeichers (2 KByte) wurde im U883 lediglich ein lnterpreterprogramm untergebracht, das ein fertiges, sowohl syntaktisch als auch semantisch fehlerfreies Programm abarbeiten kann. Es fehlen die im Abschn. 5. behandelten Kommandos zum Editieren und Testen. Sie werden von einem externen Programm mit der Bezeichnung „Editor/Debugger„ realisiert. Es kann in dem. mit dem U883 realisierten System implementiert sein, oder auf einem Wirtsrechner laufen, Diese Trennung ermöglichte die Schaffung eines verhältnismäßig leistungsfähigen Interpreters, während der Editor/Debugger in fertigen Geräten (mit fertig entwickeltem und getestetem Programm) entfallen kann. Der Interpreter gestattet die Einbindung von Programmen in Maschinensprache. Wir sprechen hierbei von Prozeduren. Eine Prozedur in Tiny MPBASIC ist ein in Maschinensprache geschriebenes Programm, das Daten vom Interpreter übernimmt, verarbeitet, Daten an ihn zurückgibt und mit einem Namen aufgerufen werden kann. Eine Funktion ist demzufolge eine Prozedur, die genau einen Wert an den Interpreter zurückgibt. Die Datenübergabe wird vom Interpreter unterstützt. Die Zuordnung des Prozedurnamens zur Startadresse des Maschinenprogramms erfolgt vermittels einer Tabelle, der Prozedurtabelle. Sie ist vom Nutzer zu erstellen. Dadurch können mit diesem Tiny BASIC, trotz Einbeziehens der Maschinenebene, maschinenunabhängige Programme verfaßt werden. Beim Übergang von einem Rechner auf einen anderen genügt es, die benutzten Prozeduren neu oder umzuschreiben, während das BASIC- Programm unverändert übernommen werden kann. Neben der Möglichkeit, selbst Prozeduren (und Funktionen) zu erstellen, bietet der Interpreter dem Anwender eine Reihe bereits fest vorgegebener Prozeduren und Funktionen. Tiny MPBASIC gestattet darüber hinaus die Verarbeitung von Traps (Fallen). Das sind Programmunterbrechungen, die beim Erfülltsein einer gewissen Bedingung vom Interpreter softwaremäßig ausgelöst werden. Die Bedingung wird vom Programmierer festgelegt (s. Abschn. 7.3.2.). Ein weiteres Problem bei der Schaffung eines möglichst geräteunabhängigen Interpreters sind die Eingabe- und die Ausgabeschnittstellen für INPUT bzw. PRINT. Um hier alle Möglichkeiten offen zu lassen, ruft PRINT ein Unterprogramm PUTCHR auf, das ein Zeichen ausgibt und vom Anwender des U883 erstellt wird. Wenn PRINT eine Zeile ausdruckt, dann wird PUTCHR so oft aufgerufen, wie Zeichen in dieser Zeile sind. Zum Einlesen von Zahlen ruft INPUT ein, ebenfalls vom Anwender geschriebenes, Programm GETCHR auf, das bei jedem Aufruf ein Zeichen einliest.

7.3. Interpreter

7.3.1. Ausdrücke

Wir unterscheiden in Tiny MPBASIC arithmetische und logische Ausdrücke. Letztere haben immer die Gestalt

```
aausdruck vop aausdruck ,
```

wobei aausdruck ein arithmetischer Ausdruck und vop einer der Vergleichsoperatoren

```
>, <,>=, <=, = und < >
```

ist. Das Ergebnis eines logischen Ausdrucks ist eine logische Größe (wahr oder falsch) und kein Zahlenwert. Hier besteht ein grundsätzlicher Unterschied zu anderen BASIC- Interpretern (vgl. Abschn. 2.4.2.), der allerdings für die Praxis untergeordnete Bedeutung hat. Logische Ausdrücke dürfen nur in Anweisungen stehen, die logische Größen verarbeiten können. In arithmetischen Ausdrücken werden ganzzahlige Konstanten, Variablen und Funktionen verknüpft. Da es Gleitkommagrößen ohnehin nicht gibt, entfällt die Kennzeichnung mit %. Dieses Zeichen hat in Tiny MPBASIC eine andere Bedeutung. Vor einer Konstanten kennzeichnet es die hexadezimale Darstellung. Der zulässige Bereich liegt zwischen %0000 und %FFFF bei hexadezimaler und zwischen -32767 und 32767 bei dezimaler Schreibweise. (Die Zahl -32768 kann intern verarbeitet, aber nicht ein- bzw. ausgegeben werden.) Gerechnet wird im Zweierkomplement. Näheres zu dieser Darstellung und dem Rechnen damit findet der Leser z. B. in [9]. Es gibt die arithmetischen Operatoren +, -, *, / und $MOD. $MOD, modulo, liefert den Rest, den eine ganze Zahl bei der Division durch eine andere ganze Zahl gibt. Es gilt

```
| A $MOD B | = | A - (A/B*B) | .
```

Bei der Division wird der gebrochene Teil weggelassen. Das Dollarzeichen vor MOD hat nichts mit einem String zu tun; Strings sind nicht implementiert. Es kennzeichnet vielmehr $MOD als einen Operator. Das trifft auch auf die bitweisen logischen Operatoren

```
$AND, $OR und $XOR
```

zu. Diese arbeiten anders als die im Abschn. 2.4.2. beschriebenen logischen Operatoren. Sie führen für jedes Paar von gleichwertigen Binärstellen ihrer Operanden in der internen Zweierkomplementdarstellung die betreffende logische Operation aus. Ihre Benutzung setzt voraus, daß Zweierkomplement, binäre Darstellung und hexadezimale Darstellung für den Programmierer vertraute Begriffe sind. Beispielsweise kann man $AND zum „Maskieren“ von Bits in einem Datenwort benutzen. Mit

```
A $AND 1
```

erhält man den Zustand vom Bit 0 in A, indem mit der UND-Verknüpfung die restlichen Bits zurückgesetzt werden. Mit $OR können Bits auf Eins gesetzt werden. $XOR eignet sich u. a. zum bitweisen Vergleich von Daten. Der Ausdruck

```
A $XOR %70 $AND %F0
```

liefert z. B. genau dann den Wert Null, wenn in A die Bits 4, 5 und 6 Eins sind und Bit 7 Null ist. Alle bitweisen logischen und arithmetischen Operatoren haben dieselbe Priorität. Sie dürfen gemeinsam in arithmetischen Ausdrücken auftreten. Damit werden arithmetische Ausdrücke streng von links nach rechts abgearbeitet. Wird eine andere Reihenfolge der Berechnung gewünscht, so ist das mit Klammern zu regeln. Weiterhin stehen die in Tafel 8 aufgeführten allgemeinen, d. h. maschinenunabhängigen Funktionen zur Verfügung. Die Funktionsparameter werden grundsätzlich in eckige Klammern geschrieben. Beim Rotieren nach links werden alle 16 Bits um eine Stelle nach links geschoben, und das höchstwertige Bit wird zum niederwertigsten Bit. Rotiert man z. B. die Zahl -32767, die intern die binäre Darstellung 1000 0000 0000 0001 hat, nach links, so ergibt das 0000 0000 0000 0011 (binär) bzw. 3 in dezimaler Schreibweise. Das Rotieren nach rechts funktioniert prinzipiell genauso, nur eben in die andere Richtung. RR ist somit die inverse Funktion zu RL. Programmiert man RR[RL[x]], so ergibt das immer x. Man braucht diese Funktion z. B., wenn Eingabeports multiplex abgefragt werden sollen.

```
Tafel 8, Funktionen in Tiny MPBASIC
ABS[x]  absoluter Betrag von x
NOT[x]  bitweise logische Negation von x
GTC ein Zeichen von der Konsole holen (benutzt GETCHR)
INPUT   Zahl von der Konsole holen
RL[x]   x um ein Bit nach links rotieren
RR(x)   x um ein Bit nach rechts rotieren
```

```
Tafel 9. Maschinenorientierte Funktionen im U8S3
GETR[r] Inhalt des Registers r lesen
GETRR[r]    Inhalt des Registerpaares r, r+1 lesen
GETEB[a]    Inhalt des auf der Adresse a abgespeicherten Bytes im
Datenspeicher lesen
GETEW(a)    Inhalt der auf den Adressen a, a+1 abgespeicherten Bytes im
Daten-Speicher lesen
```

Darüber hinaus gibt es für den U883 einige Funktionen, mit denen man auf Register bzw. auf den Datenspeicher zugreifen kann (Tafel 9). Dadurch hat der Anwender sofort, d. h. ohne erst eigene Funktionen schreiben zu müssen, Zugriff auf fast alle Schaltkreisfunktionen. BASIC-Programme, die von diesen Funktionen Gebrauch machen, sind natürlich nicht unverändert auf einen anderen Rechner übertragbar. Deshalb sind diese vier Funktionen nicht Bestandteil der Sprache Tiny MPBASIC. Würde man etwa einen Interpreter für den U880 schreiben, so wären hier die Funktionen

```
GETB(a) Byte aus dem Speicher von Adresse a lesen
GETW[a] Bytes aus dem Speicher von den Adressen a und a+1 lesen
IN[p] Port p lesen
```

sinnvoll.

7.3.2. Anweisungen

Da uns die BASIC-Anweisungen i. allg. schon bekannt sind, genügt es, hier die davon in Tiny MPBASIC implementierten mit der Syntaxvorschrift aufzuführen, und nur die Besonderheiten zu erläutern (Tafel

10). Eine Anweisung besteht aus ihrem Namen, z. B. LET, und den Argumenten, z. B. A=0. Soll eine Anweisung mehrmals nacheinander ausgeführt werden, so genügt es, den Namen nur einmal und lediglich die Argumente mehrfach aufzuschreiben, wenn man letztere mit Komma voneinander trennt. Zum Beispiel

```
10 LET A=0, B=0, I=10
```

```
Tafel 10. Anweisungen in Tiny MPBASIC
LET variable = aausdruck GOTO aausdruck
IF lausdruck THEN anweisung
IF lausdruck THEN anweisungsblock
ELSE anweisung
ELSE anweisungsblock
GOSUB aausdruck
RETURN
INPUT {"text"} variable
PRINT {"text"} {aausdruck}
PRINTHEX {"text") {aausdruck}
STOP
END
REM {kommentar}
CALL aausdruck
PROC {variablenliste =} prozedurname {parameterliste}
TRAP lausdruck TO aausdruck
CLRTRP
WAIT aausdruck
```

Bei der Anweisung **GOTO** ist ein Ausdruck zur Angabe der Zeilennummer zugelassen. Gleiches trifft auf **GOSUB** zu. Die Anweisungen ON…GOTO und ON…GOSUB sind dadurch leicht zu umgehen. Ein Anweisungsblock besteht aus beliebig vielen, durch Semikolon getrennten Anweisungen, die auf einer Zeile stehen. Es ist möglich, mehrere Anweisungen auf eine Zeile zu schreiben, wenn man diese durch Semikolon voneinander trennt. Bei dem in Anführungsstriche gesetzten Text in den Anweisungen INPUT, PRINT und PRINTHEX handelt es sich im Grunde um eine Stringkonstante. Strings gibt es ansonsten nicht. **INPUT** druckt zunächst den programmierten Text aus und erwartet dann die Eingabe einer Zahl (dezimal oder hexadezimal). Wie wir bereits sahen (Tafel 8), gibt es **INPUT auch als Funktion**. Dort kann kein Text ausgedruckt werden. Es erscheint statt dessen ein Fragezeichen. INPUT hat als Funktion den Vorteil, daß es wie jede Funktion in Ausdrücken benutzt werden kann. Will man z. B. den Nutzer seines Programms eine Temperatur in Zehntel Grad Celsius eingeben lassen, die jedoch eigentlich in Zehntel Grad Kelvin benötigt wird, so kann man einfach

```
10 LET T = INPUT+2732
```

programmieren.

Die Anweisung **PRINTHEX** unterscheidet sich von PRINT nur dadurch, daß das Ergebnis des Ausdrucks hexadezimal ausgegeben wird. Will man mehrere Ausgaben auf eine Zeile bringen, so benutzt man bei Tiny MPBASIC anstelle des Semikolons (dieses wird schon zum Trennen von Anweisungen benutzt) das Komma. Das Drucken in Spalten ist nicht möglich.

Mit der Anweisung **PROC** werden Prozeduren vom BASIC-Interpreter aufgerufen. Die Variablenliste sind in eckige Klammern gesetzte und durch Komma getrennte Variablen. Bei der Parameterliste

handelt es sich um in eckige Klammern gesetzte und durch Komma getrennte Ausdrücke. Zum Beispiel

```
10 PROC [A, 3, X] = MPROG [3*A, X, 2] .
```

Es gibt bereits implementierte Prozeduren (Tafel 11). Der Sinn der Prozeduren besteht allerdings vor allem darin, daß der Anwender, wenn er selbst welche schreibt, Tiny MPBASIC einem gegebenen Anwendungsfall optimal anpassen kann. Mit Prozeduren realisiert man elementare Steuerfunktionen, die in BASIC nicht oder nur umständlich zu programmieren wären, oder deren Realisierung in BASIC ein zu langsames Programm ergeben würde.

```
Tafel 11. Prozeduren im U883
PTC[z]  Zeichen z an die Konsole ausgeben (benutzt PUTCHR)
SETR(r,w]   Register r mit dem Wert w belegen
SETRR[r,w]  Registerpaar r, r+1 mit dem Wert w belegen
SETEB[a,w]  Byte im Datenspeicher auf der Adresse a mit dem Wert w belegen
SETEW[a,w]  Bytes im Datenspeicher auf den Adressen a, a+1 mit dem Wert w
belegen
```

Die SET-Prozeduren in Tafel 11 sind, wie die GET-Funktionen, wiederum nicht Bestandteil der Sprache Tiny MPBASIC. Sie haben ebenfalls nur den Zweck, den Interpreter im U883 für den Anwender sofort, d. h. ohne daß er selbst erst eine Prozedur schreiben muß, nutzbar zu machen.

Eine weitere, speziell für Steuerungen gedachte Anweisung, ist **TRAP** (Falle). Nachdem TRAP abgearbeitet wurde, testet der Interpreter vor der Abarbeitung jeder neuen Programmzeile die gesetzte Trapbedingung (lausdruck). Sobald diese erfüllt ist, erfolgt automatisch ein GOSUB zu der durch aausdruck gegebenen Zeile. Die Bedingung wird vorher gelöscht. Danach wird die Traproutine abgearbeitet, bei der es sich um ein gewöhnliches Unterprogramm handelt, welches mit RETURN endet. RETURN bringt die Programmabarbeitung schließlich wieder zu der Stelle, an der sie unterbrochen wurde. Soll die Trapbedingung danach weiter überwacht werden, so muß in der Traproutine eine entsprechende TRAP-Anweisung gestanden haben. Es kann immer nur eine Bedingung aktiv sein. Das Setzen einer neuen führt zum Löschen der alten Bedingung. Oft ist es jedoch möglich, eine Bedingung so zu formulieren, daß sie mehrere in sich vereint, wenn das nötig sein sollte. Welche dann davon das Trap ausgelöst hat, kann in der Traproutine ermittelt werden. An einem Anwendersystem mögen z. B. zwei 8-Bit-A/D-Wandler angeschlossen sein, die mit den selbst geschriebenen Funktionen AD1 und AD2 abgefragt werden können. Wenn AD1>200 oder AD2>180 wird, soll ein Trap ausgelöst werden. Dann kann man

```
10 TRAP AD1/201 $OR (AD2/181)>0 TO ...
```

programmieren. Die Division AD1/201 ergibt so lange Null, wie AD1⇐200 ist, weil die Stellen nach dem Komma abgeschnitten werden. Genauso bleibt AD2/181 Null, wenn AD2⇐180 ist. Überschreitet einer der beiden, AD1 oder AD2, den vorgeschriebenen Maximalwert, so liefert die Verknüpfung $OR ein Ergebnis >0, und es kommt zum Trap.

Will man die Trapbedingung nur löschen, ohne dabei eine neue zu setzen, so gibt man die Anweisung **CLRTRP** (clear trap, lösche Trap).

Schließlich sei noch **WAIT** erläutert. Diese Anweisung berechnet zunächst den Ausdruck und ruft danach eine Software-Warteschleife auf, die so oft durchlaufen wird, wie das Ergebnis des Ausdrucks angibt. Ein Durchlauf dauert genau eine Millisekunde. (Beim U883 wird dabei ein 8-MHz-Quarz

vorausgesetzt.) Bei der Anwendung von WAIT beachte man, daß die Berechnung des Ausdrucks und die vor- und nachher abzuarbeitenden Anweisungen ebenfalls Zeit brauchen. Weiterhin hat man bei WAIT mit großen Zeiten zu beachten, daß während des Wartens keine Trapbedingung getestet wird.

7.4. Anwendungsbeispiele

Beispiel 1. Trotz des bescheidenen Zahlenbereiches lassen sich eine Reihe von mathematischen Berechnungen auch in Tiny BASIC mit hinreichender Genauigkeit ausführen, wiewohl etwas mehr Sorgfalt bei der Programmierung erforderlich ist, um Bereichsüberschreitungen zu vermeiden. Wir wollen mit dem Newton-Verfahren die Quadratwurzel aus einer Zahl berechnen. Dazu wird von einer Anfangsnäherung X0 ausgegangen und so lange eine bessere Näherung berechnet, bis sich die erreichbare Genauigkeit eingestellt hat. Die Formel zur Berechnung der verbesserten Näherung Xn+1 aus der Näherung Xn lautet

Xn+1 = Xn – (Xn^2 – Y) / (2Xn),

wobei Y die zu radizierende Zahl ist. Würde man nun

```
LET X = X-(X*X-Y)/(2*X)
```

programmieren, so brächte das keine brauchbaren Ergebnisse. Das liegt zum einen daran, daß bei der Berechnung von xn^2 es bereits für Xn>181 zu Bereichsüberschreitungen kommt, und zum anderen wird, was viel schlimmer ist, mit wachsendem n die Differenz zwischen Xn^2 und > immer kleiner und die Division durch 2*Xn deshalb und wegen der Verwendung von ganzen Zahlen viel zu ungenau. Das ist jedoch noch kein Grund, das Newton-Verfahren hier zu verwerfen. Statt dessen überlegt man, wie die Formel eventuell umzustellen ist, um die eben festgestellten Schwierigkeiten zu umgehen. Und tatsächlich ist das möglich. Es ist nämlich

Xn+1 = Xn – (Xn^2 – Y) / (2Xn) = Xn – Xn / 2 + Y / (2Xn)

Bei dieser Umformung sind sowohl das Quadrat als auch die kritische Differenz verschwunden. Wir können folgendes Programm aufschreiben:

```
10 INPUT "Y = " Y
20 LET X=1, N=10
30 LET X=X-(X/2)+(Y/X/2)
40 LET N=N-1; IF N>0 THEN GOTO 30
50 PRINT "SQR(Y) =" X
```

Als Anfangsnäherung benutzen wir die 1, und die zehnte Näherung ist bei unserem Zahlenbereich immer genau genug.

Beispiel 2. In dem nun folgenden Beispiel benutzen wir den Operator $MOD. Es handelt sich um eine Primfaktorzerlegung. Man benötigt so etwas z.B., um die Frequenzen zu ermitteln, die sich leicht aus einer gegebenen (Quarz-) Oszillatorfrequenz mittels Teilerstufen erzeugen lassen. Das benutzte Verfahren ist recht einfach. Es werden alle möglichen Faktoren ausprobiert, bis die Zahl schließlich vollständig zerlegt ist.

```
10 INPUT "ZAHL: " Z
20 LET F=2
```

```
30   IF Z $MOD F=0 THEN LET Z=Z/F; PRINT F
40     ELSE LET F=F+1
50 IF F*F<=Z THEN GOTO 30
60 PRINT Z
```

Die Zeile 30 lautet verbal umschrieben: Wenn Z durch den Faktor F teilbar ist, dann führe die Division aus und drucke F als ermittelten Primfaktor.

Beispiel 3. Wir wollen den Inhalt des Datenspeichers des U883 in einem gewissen Adreßbereich ausdrucken. Dabei soll das folgende Druckbild auf jeder Zeile entstehen:

adresse daten1 daten2 ... daten8 ascii

Die Daten geben wir zu je zwei Bytes aus. also kommen 16 Bytes auf jede Zeile. Rechts erscheinen die Bytes, die als druckbare ASCII-Zeichen interpretiert werden können, als solche ausgedruckt. Damit kann man leicht abgespeicherte Texte erkennen. Das Programm hierfür sieht folgendermaßen aus:

```
10 REM RAM DUMP
20 INPUT "ADRESSE: " A, "LAENGE: " I
25 IF I<=0 THEN END
27 REM AUF VIELFACHES VON 16 AUFRUNDEN
30 IF I $MOD 16<>0 THEN LET I=I/16+I*16
40 REM EINE ZEILE
45   REM ADRESSE
50   LET J=0; PRINTHEX A, " ",
55   REM DATEN
60     PRINTHEX " " GETEW[A],
70     LET I=I-2, J=J+2, A=A+2
80     IF J <16 THEN GOTO 60
90   LET A=A-16
95 REM ASCII
100 PRINT " ",
110 LET C=GETEB[A]
120 IF C>%1F THEN IF C<%7F THEN PROC PTC[C]
130   ELSE PROC PTC[%2E]; REM PUNKT
140 LET J=J-1, A=A+1
150 IF J>0 THEN GOTO 110
160 PRINT
170 IF I>0 THEN GOTO 50
```

In Zeile 30 wird die eingegebene Länge auf eine durch 16 teilbare Zahl aufgerundet, damit auch die letzte Zeile des RAM-Ausdrucks immer voll wird. Zeile 120 druckt alle Zeichen, deren Kode zwischen %1F und %7F liegt. Handelt es sich jedoch um kein druckbares Zeichen, so gibt Zeile 130 einen Punkt aus.

Beispiel 4. Wir wollen nun die Benutzung von TRAP mit einem formalen Beispiel illustrieren. Es bringt darüber hinaus einige typische Details bei der Anwendung von Tiny MPBASIC.

```
10 REM ZEICHENSATZ AUSDRUCKEN UND
20 REM IM HINTERGRUND TASTENEINGABEN LESEN
30   LET C=%20, A=%1300, I=0
```

```
40   GOSUB 250
50     PROC PTC[C]
60     LET W=100
70       LET W=W-1
80     IF W>0 THEN GOTO 70
90     LET C=C+1
100  IF C<%5F THEN GOTO 50
110 IF I=0 THEN GOTO 30
115 REM EINGABEN WIEDER AUSDRUCKEN
120 LET J=0
130 PROC PTC[%0D]
140   PROC PTC[GETEB[A+J]]
150   WAIT 100; LET J=J+1
160 IF I>J THEN GOTO 140
170 END
200 REM TRAPROUTINE
210 PROC SETR[%FA, GETR[%FA]$AND %F7]
220 PROC SETEB[A+I, GETR[%F0]]
230 LET I=I+1
240 IF I>%FF THEN LET I=0
250 TRAP GETR[%FA]$MOD 8<>0 TO 210
260 RETURN
```

Dieses Programm druckt fortwährend alle ASCII-Zeichen mit den Kodes %20 . .. %5E bis während des Druckens über die SIO Zeichen empfangen worden sind. Dann werden die empfangenen Zeichen ausgedruckt, und die Abarbeitung ist beendet. TRAP überwacht im Interrupt-Requestregister des U883 (Reg. %FA) das SIO-Empfängerbit, das in der Trapbedingung mit $AND 8 maskiert wird. Da die Trapbedingung am Ende der Traproutine ohnehin neu gesetzt werden muß, erfolgt das erste Setzen mit GOSUB 250 in Zeile 40. Das spart Speicherplatz, Mit LET A=%1300 in Zeile 30 wird die RAM-Adresse festgelegt, wo die empfangenen Zeichen abgespeichert werden sollen. Die Zeilen 60… 80 bilden eine Warteschleife, die zwischen dem Ausdrucken zweier Zeichen eine Pause erzeugt. Hier wäre es nicht günstig, WAIT einzusetzen, weil dann so lange die Trapbedingung nicht überwacht werden würde. In der Traproutine wird zuerst in Zeile 210 das SIO-Bit zurückgesetzt. Das geschieht dadurch, daß zuerst das gesamte Register gelesen, anschließend das Bit mittels $AND %F7 rückgesetzt und schließlich das Ergebnis mit SETR wieder in das Register eingeschrieben wird. Dadurch bleiben die anderen sieben Bits unverändert. In Zeile 220 wird das empfangene Zeichen gelesen (SIO-Register %F0) und abgespeichert. Wir sehen hier, wie man in Tiny MPBASIC mit einem eindimensionalen Feld arbeiten kann, obwohl Felder gar nicht implementiert sind. Statt DIM A… zu programmieren, legt man mit LET A=%1300 eine Adresse für dieses Feld fest und kann dann mit SETEB[A+I, …] oder GETEB[A+I] anstelle von LET A(I) = … bzw. … A (I) … auf dieses Feld zugreifen. Selbstverständlich hat man dabei darauf zu achten, daß der Index I keine unzulässigen Werte annimmt (Zeile 240).

7.5. Benutzen des Interpreters im U883

Der Interpreter im U883 ist als Unterprogramm ausgeführt, das ein fertiges, fehlerfreies BASIC-Programm abarbeiten kann. Der Eintrittspunkt ist %7FD, Vor dem Aufruf ist in die Register 6 und 7 die Startadresse des BASIC-Programms und in die Register 8 und 9 die Adresse der Prozedurtabelle, deren Aufbau wir im folgenden mit beschreiben werden, zu laden. Ist eine Prozedurtabelle nicht

vorhanden, dann müssen die Register 8 und 9 gelöscht werden. Der Registerzeiger ist auf %10 zu setzen.

Das BASIC-Programm wird ohne Leerzeichen im ASCII-Kode abgelegt. Die Zeilennummern werden binär abgespeichert, wobei jedoch das höchstwertige Bit immer gesetzt ist. Die Namen der Anweisungen und die Operatoren mit $ werden, wie in Tafel 12 aufgeführt, abgekürzt. Als Markierung für das Zeilenende dient Carriage Return (%0D), für das Programmende %00.

```
Tafel 12. Abkürzungen in Tiny MPBASIC

LET L
GOTO    G
IF ... THEN F ... ;
ELSE    >;
GOSUB   S
RETURN  R
PROC    O
TRAP ... TO ! ... ,
CLRTRP  /
INPUT   I
PRINT   P
PRINTHEX    H
STOP    T
END E
WAIT    W
CALL    C
REM M
SAND    $A
SOR $O
SXOR    $X
SMOD    $M
```

Das folgende kleine Programm, das 1 KByte Speicher ab Adresse %1000 löscht, wird so, wie in Tafel 13 aufgezeigt, abgespeichert. Zum Ausdrucken der internen Darstellung wurde das im letzten Abschnitt (Beispiel 2) aufgeführte Programm benutzt.

```
10 LET A=%1000, L=%400
20 PROC SETEB[A,0]
30 LET A=A+1, L=L-1
40 IF L>0 THEN GOTO 20
```

```
Tafel 13. Interne Darstellung eines Programms in Tiny MPBASIC

2000 80 0A 4C 41 3D 25 31 30 30 30 2C 4C 3D 25 34 30 ..LA=%1000,L=%40
2010 30 0D 80 14 4F 53 45 54 45 42 5B 41 2C 30 5D 0D 0...OSETEB[A,0].
2020 80 1E 4C 41 3D 41 2B 31 2C 4C 3D 4C 2D 31 0D 80 ..LA=A+1,L=L-1..
2030 28 46 4C 3E 30 3B 47 32 30 0D 00 00 00 00 00 00 (FL>0;G20.......
```

Es sei angenommen, daß dieses Programm auf der Adresse %2000 stehe. Dann kann es mit den folgenden Maschinenbefehlen aufgerufen werden:

```
...
LD 6, #%20
CLR 7
CLR 8
CLR 9
SRP #%10
CALL %07FD
...
```

Für die Programme **PUTCHR** und **GETCHR**, die das Benutzen von PRINT, PRINTHEX und INPUT ermöglichen, gelten nachstehende Konventionen: GETCHR hat die Eintrittsadresse %0815, PUTCHR %818. Das eingelesene Zeichen (GETCHR) wird im Arbeitsregister R3, das auszugebende Zeichen (PUTCHR) im Arbeitsregister R5 übergeben. Der Registerzeiger steht auf %10. Zur freien Verfügung hat der Anwender die Register ab %54.

Die **Prozedurtabelle**, die dem Interpreter die Zuordnung vom Prozedurnamen zur Eintrittsadresse ermöglicht, hat folgenden Aufbau:

```
Byte 0|1|2|3|4| . . . |N|N+1|N+2| ...
```

Byte 0 enthält die Länge N des Prozedurnamens, die binär abgespeichert wird. (0<N<%FE = 254). Die Bytes 1 ... N enthalten den Namen im ASCII-Kode, und die Bytes N+1 und N+2 enthalten die Eintrittsadresse der Prozedur. Danach darf, beginnend mit der Länge ihres Namens, die nächste Prozedur in die Tabelle eingetragen werden. Das Ende wird mit %FF gekennzeichnet. Beginnen z. B. die Prozeduren PROZ und FUNKT auf den Adressen %2010 bzw. %3000, dann müßte eine Prozedurtabelle damit so aussehen:

```
%04|%50|%52|%4F|%5A|%20|%10|%05|%46|%55|%4E|%4B|%54|%30|%00|%FF
```

Die **Übergabe der Parameter** vom Interpreter an die Prozedur und der Ergebnisse von der Prozedur an den Interpreter erfolgt über den Stack. Wenn die Prozedur m Ergebnisse an den Interpreter übergibt und n Parameter von diesem übernimmt, dann befindet sich der Stack beim Aufruf der Prozedur in dem in Tafel 14 aufgezeigten Zustand. Vor dem letzten RET in der Prozedur, das wieder zum Interpreter führt, muß der Stackpointer um 2n—2 erhöht worden sein (, wenn n>1) und natürlich auf die Rückkehradresse zum Interpreter zeigen.

```
Tafel 14. Datenübergabe bei Prozeduren im U883

Stack   Inhalt
SP+2n+2m+1  Platz für das m-1te bis erste Ergebnis,
... SP+2n+2 wenn m>1. Der Wert für das letzte (mte) Ergebnis ist in das
Arbeitsdoppelregister RR2 zu schreiben.
SP+2n+1 ... SP+2n   Wird vom Interpreter benutzt.
SP+2n-1 Erster bis n-1ter Parameter, wenn n>1.
... SP+2    Der Wert des letzten Parameters befindet sich im
Arbeitsdoppelregister RR4.
SP, SP+1    Rückkehradresse zum Interpreter
SP  Inhalt vom Stackpointer
```

Schließlich sei noch erwähnt, daß der U883 auf der Adresse %812 startet, falls dort ROM liegt und

sonst auf %E000.

7.6. Editor/Debugger

Vom Hersteller des U883 wird auch der zugehörige Editor/Debugger, den wir der Kürze halber nur „Editor„ nennen wollen, bereitgestellt. Er meldet sich mit # als Promptzeichen. Danach gibt man als erstes Kommando NEW, falls noch kein Programm im Speicher stand. Das ist notwendig, weil beim Einschalten der Speicher nicht automatisch gelöscht wird, um ein evtl. in einem CMOS-RAM abgelegtes Programm zu erhalten. Danach können Programme eingegeben werden, so wie wir das im Abschn. 5. kennengelernt haben. Der Editor beseitigt dabei die Leerzeichen, ersetzt die Anweisungsnamen durch deren Abkürzungen, setzt die Zeilennummer um und überprüft die syntaktische Richtigkeit. Er kennt außer NEW die folgenden Kommandos:
LIST {zeilennummer}
RUN
CONT {zeilennummer}
STEP {zeilennummer}
Mit LIST wird das gesamte Programm bzw. die angegebene Zeile aufgelistet. Drücken von Return nach LIST mit Zeilennummer führt zum Auflisten der Folgezeile. RUN startet das Programm an dessen Anfang, während man mit CONT von der eingegebenen Zeilennummer ab starten kann. Darüber hinaus kann mit diesem Kommando die Programmabarbeitung nach STOP fortgesetzt werden. STEP schließlich gestattet das Abarbeiten einer einzelnen Zeile. Anschließendes Drücken von Return bewirkt das Abarbeiten der Folgezeile.

Anhang B. Syntaxbeschreibung von Tiny MPBASIC

Metalinguistische Konstanten sind mit Großbuchstaben gedruckt. Das sind Zeichen bzw. Zeichenketten, die unverändert in die konkrete Anweisung übernommen werden. Die metalinguistischen Variablen sind klein gedruckt. Sie werden in der konkreten Anweisung durch andere Zeichen bzw. Zeichenketten gemäß der Syntaxdefinitionen ersetzt (vgl. Erläuterungen auf S. 17). Weiterhin bedeutet:

```
->   verbale Erläuterungen
=>   Definitionszeichen
|   oder
{xxx}   darf weggelassen werden
{xxx}*  darf beliebig oft dastehen oder auch weggelassen werden
{xxx}*n darf maximal n mal dastehen oder auch weggelassen werden

Syntaxbeschreibung

programm    => zeile {zeile}*
zeile       => zeilennummer anweisung {; anweisung}*
zeilennummer        => pkonst
anweisung       => anweisungsname anweisungsargumente {,
anweisungsargumente}
anweisungsname  => LET|PROC|GOTO|IF[ELSE|RETURN|GOSUB[WAIT|REM|
CALL|STOP|END|TRAP|CLRTRP|PRINT| PRINTHEX | INPUT
```

```
anweisung   => (s. Tafel 10, S. 64)
ausdruck    => aausdruck/lausdruck
aausdruck   => wert {alop wert}*
wert        => konst|var|fkt|(aausdruck)
alop        => +|-|*|/|$MOD|$AND[SOR|$XOR
konst       => pkonst | nkonst
var     => buchst
fkt     => funktionsname {parameterliste}
pkonst      -> ganze Zahl G mit 0<=G<=32767
nkonst      => -pconst
funktionsname   => name
name        => buchst buchst|ziffer {buchst|ziffer}*252
buchst      => A|B|C ... X|Y|Z
ziffer      => 1|2|3 ... 8|9|0
lausdruck   => aausdruck vop aausdruck
vop     => >|=|<|>=|<=|<>
variabienliste  {var{, var}*}
prozedurname    => name
parameterliste      => [aausdruck {, aausdruck}*]
text        -> bei. ASCII-Zeichenfolge ohne "
kommentar   -> bei. ASCII-Zeichenfolge ohne ; und ohne Return
```

Literatur

(11) Bennewitz, W.Podszuweit, H.: Programmierung von Einchipmikrorechnern. REIHE AUTOMATISIERUNGSTECHNIK, Bd. 215. Berlin: VEB Verlag Technik 1985
(12) Müller, S.: Einchipmikrorechner U883 interpretiert Tiny MPBASIC. radio fernsehen elektronik 34(1985) 3, S. 143 f.
(13) Technische Beschreibung Einchipmikrorechner U883. Erfurt: veb mikroelektronik „karl marx“ 1985

1)
Tiny bedeutet klein, winzig

From:
https://hc-ddr.hucki.net/wiki/ - **Homecomputer DDR**

Permanent link:
https://hc-ddr.hucki.net/wiki/doku.php/elektronik/u883/mpbasic?rev=1627915847

Last update: **2021/08/02 14:50**