Basic/Debug-Handbuch

ZILOG REFERENCE MANUAL Z8671 SINGLE-CHIP INTERPRETER BASIC/DEBUG SOFTWARE

Basic/Debug meldet sich mit dem Prompt ':' und wartet auf Eingabe.

Alle Kommandos im Direktmodus und im Programmmodus möglich Zeilennummern 1 bis 32767

Korrektur der Eingabe nur durch DEL, nicht mit Kursor links !!!!

Mehrere Anweisungen können auf eine einzelne Zeilennummer folgen, wenn sie durch Doppelpunkte getrennt werden. Mehrere Befehle in eine Zeile packen spart Speicherplatz. Die Anzahl der Befehle in der Zeile ist nicht beschränkt, aber die Zeile darf nicht mehr als 130 Zeichen enthalten.

Basic/Debug ignoriert die Unterscheidung zwischen Groß- und Kleinbuchstaben. Daher sind PRINT, PrInT und print alles gültige Basic/Debug-Anweisungen. Zur guten Lesbarkeit wird hier alles in Großbuchstaben geschrieben.

Beispiele:

```
PRINT "Hallo"
IF C <> USR(A) %500
@%1020 = 100
"DIE ANTWORT IST";X
```

Basic/Debug kennt fünfzehn Schlüsselwörter (Beschreibung s. unten).

Leerzeichen diesen nur der Lesbarkeit. In der obigen Beispiel-Programmzeile trennt ein Leerzeichen das Schlüsselwort PRINT vom Argument „HALLO“. Obwohl es die Aussage einfacher macht, ist der Leerraum für Basic unnötig. Innerhalb einer Anweisung in einer Zeile ignoriert Basic/Debug alle Leerzeichen. Eingegebene Leerzeichen bleiben jedoch im Programm und belegen Speicherplatz. (Außer zwischen Zeilennummer und erstem Zeichen, hier werden Leerzeichen entfernt)

```
print a
P RI NT A
PRINTA
```

sind alles gültige Anweisungen.

Wenn vor der Eingabe eine ganze Zeile gelöscht werden muss, ist es schneller, die Escape-Taste zu drücken, als mit Backspace durch den Zeilenpuffer zu gehen. Ein Escape-Tastendruck leert den Inhalt des Zeilenpuffers.

Zahlen

Alle Berechnungen werden in zwei Acht-Bit-Registern durchgeführt, erfordern Sechzehn-Bit-Werte und geben Sechzehn-Bit-Ergebnisse zurück.

Wenn ein Ergebnis sechzehn Bit überschreitet, wird abgeschnitten und der Überlauf wird verworfen.

Alle numerischen Werte werden intern in dargestellt 16-Bit binäre Zweierkomplementform.

Numerische Werte reichen von -32768 bis +32767. Wenn eine Berechnung ergibt einen Wert außerhalb des negativen Bereichs, der Antwort wird als positive Zahl gedruckt. Wenn eine Berechnung Ergebnis ist höher als der positive Bereich, wird eine negative Zahl gedruckt.

Hexadezimalwerte werden häufig zur Adressierung verwendet da Hardwaregrenzen oft bei geraden Hex-Adressen auftreten. Ganzzahlen ohne Vorzeichen zwischen 0 und 65536 können in die Adresse eingegeben werden.

Normalerweise werden nur Werte im Bereich von +32767 bis -32768 ausgegeben. Das Drucken von Werten außerhalb des Bereichs ist mit '\' möglich, s.u.

Basic/Debug kennt nur ganze Zahlen. Brüche kann nicht eingegeben werden, und der Bruchteil jedes Ergebnisses ist verworfen. Im Folgenden sind Beispiele für gültige Dezimal- und hexadezimale Konstanten:

```
Dez:    Hex:
123     %7B
256     %100
32766   %7FFE
32768   %8000
```

Variablen

Basic/Debug unterstützt 26 Variablen. Jeder Buchstabe des Alphabets wird als Variablenname verwendet.

Operatoren

Basic/Debug unterstützt zwei Sätze von Operatoren: arithmetische Operatoren und relationale Operatoren. Basic/Debug erkennt die folgenden traditionellen Operatoren für arithmetische Funktionen:

```
+
-
*
/
```

Die Operationen werden von links nach rechts ausgeführt. Dabei werden Multiplikation und Division zuerst durchgeführt, gefolgt von Addition und Subtraktion. Dies kann durch die Verwendung von Klammern geändert werden. Zum Beispiel:

```
3*24-18/3+10 = 76
3*(24-18)/(3+10) = 1
```

Byte-Befehl verwendet, um Bytes des Basic-Programms in den ersten 32 KB zu ändern.

Basic/Debug unterstützt keine Bruchzahlen, daher wird Rest der Division in der zweiten Zeile verworfen.

Ein spezieller Divisionsoperator ist der Backslash „\“ für vorzeichenlose Division. Der Dividend wird als vorzeichenlose ganze Zahl im Bereich von 0 - 65535 behandelt. Beispiel: Die Anweisung PRINT 40000/3 gibt den korrekten Wert 13333 zurück. Dagegen gibt PRINT 40000/3 -8512 zurück, weil die Ganzzahl mit Vorzeichen für 40000 ist -25536.

Weil hier der Dividende als positive sechzehn Bit-Zahl behandelt wird, kann der Backslash eine logische Rechtsverschiebung auf Bitmuster durchführen:

```
(-2)/2 = -1
 (-2)\2 = 32767
```

Ein Versuch, mit dem Backslash durch eine negative Zahl zu dividieren, liefert undefinierte Ergebnisse.

Der Backslash-Operator kann auch zum Drucken von Werten größer als +32767 verwendet werden. Angenommen, N ist ein Wert außerhalb des normalen Druckbereichs, so wird dieser mit folgender Anweisung korrekt ausgegeben:

```
PRINT N\10; N-N\10*10
```

AND führt ein logisches UND aus. Es kann verwendet werden, um zu maskieren, zu drehen, Bits ausschalten oder Bits zu isolieren.

AND (Ausdruck, Ausdruck)

Die beiden Ausdrücke werden ausgewertet, dann ihre Bitmuster UND-verknüpft. Zum Beispiel gibt AND (3,6) den Wert 2 zurück. Wenn nur ein Wert in Klammern steht, wird er mit sich selbst verknüpft.

Für logisches ODER ergänzen Sie die AND-Funktion um das Subtrahieren jedes Elements von -1. Zum Beispiel entspricht folgendes dem ODER von A und B:

```
    -1-AND(-1-A, -1-B)
```

Die arithmetische Summe kann auch für das logische ODER verwendet werden wenn bekannt ist, dass die hinzuzufügenden Bits vorher Null sind.

Basic/Debug kann die internen Z8671-Register direkt ansprechen sowie den externen Speicher (> 100h). Der Inhalt einer beliebigen Adresse kann untersucht und RAM geändert werden.

Mit @ erfolgt ein byteweiser Zugriff, mit ^ der adressweise Zugriff:

Zahlen > 256 werden als RAM-Adresse gesehen, Zahlen < 256 als Register-Nummer.

@%1000 Byte auf Adresse 1000h

Basic/Debug kann eine Maschinensprache-Subroutine aufrufen, die einen Wert für die weitere Berechnung durch die USR-Funktion zurück gibt. Um ein Unterprogramm aufzurufen, das keinen Wert zurückgibt, verwenden Sie den GO@ Befehl.

Nachdem das Unterprogramm für die Maschinensprache zusammengestellt wurde, speichern Sie es im Speicher, der sonst nicht von Basic/Debug belegt wird (außerhalb Programm oder Stack).

Verwenden Sie die Adresse der ersten Anweisung des Unterprogramms als erstes Argument der USR-Funktion wie folgt:

```
USR(%2000)
```

Basic/Debug führt alles aus, was es an dieser Adresse findet. Wenn es keine Maschinensprachroutine am Standort gibt, ist das Ergebnis undefiniert. Auf die Adresse können ein oder zwei Werte folgen, die vom Unterprogramm verarbeitet werden können. Zum Beispiel:

```
USR(%2000,256,C)
```

Die Adresse und die Argumente sind Ausdrücke, die durch Kommas getrennt sind. Basic/Debug übergibt die Werte an das Unterprogramm in den Registern 18-19 und 20-21 und erwartet den resultierenden Wert in 18-19 zurück. Dieser resultierende Wert wird verwendet, um die Auswertung des Ausdrucks zu beenden.

Die Register, in denen die Argumente übergeben werden, hängen von der Anzahl der Argumente innerhalb der Klammern ab. Zum Beispiel ruft die Funktion USR(%700,A) das Unterprogramm bei %700 auf und übergibt ihm die Variable A im Register 18-19. USR(%700,A,B) jedoch übergibt A in 20-21 und B in 18-19. In beiden Fällen muss das Maschinensprachen-Unterprogramm den Rückgabewert in 18-19 übergeben:

USR Arguments and Registers

```
call            R18-19 contains    R20-21 contains
USR (%700, A, B)    B        A
USR (%700, A)       A        A
```

Das Unterprogramm für die Maschinensprache muss den folgende Anforderungen genügen: es muss mit einem RET (hex AF) enden, der zurückzugebende Wert muss in 18-19 belassen werden, es dürfen nur freie Register verwendet werden, die im Anhang aufgeführt sind. Der Registerzeiger ist so eingestellt, dass er auf 16-31 zeigt. Damit können die Argumente direkt aus den Arbeitsregistern r2-r3 und r4-r5 abgerufen werden. Der Registerzeiger darf verändert werden. Er muss am Ende auch nicht restauriert werden.

Die einzelnen Anweisungen von Basic/Debug

GO@ GO '@' address [',' arg_l [',' arg_2]]

```
GO@%E000, A, B
GO@%700
```

Der GO@-Befehl verzweigt bedingungslos zu einem Maschinensprachen-Unterprogramm. Er darf nur verwendet werden, wenn das Unterprogramm keinen Wert zurückgibt. Das erste Argument ist die Adresse des ersten Bytes des Subroutine. Die letzten beiden optionalen Argumente werden verwendet, um Werte an das Unterprogramm zu übergeben. Im Gegensatz zur USR-Funktion wird der Inhalt von R18-19 verworfen und es wird kein Wert zurückgegeben. Ansonsten übergibt GO@ Argumente an die Unterprogramm auf die gleiche Weise wie USR (d.h. in Registier 18-19 und 20-21, s.o.).

GOSUB GOSUB expression

```
GOSUB 50
GOSUB C
GOSUB B*100
```

Im Gegensatz zu Dartmouth Basic steht nach dem Schlüsselwort GOSUB entweder die Nummer der ersten Zeile des Unterprogramms oder ein Ausdruck, der die Subroutine-Zeilennummer ergibt. Das Unterprogramm muss mit RETURN beendet werden.

GOSUB speichert die Nummer der nächsten zu auszuführenden Zeile. RETURN führt das Programm an dieser Zeile fort. GOSUB muss der letzte Befehl in seiner Zeile sein. Ein Unterprogramm kann ein anderes aufrufen. Die RETURN-Anweisung am Ende des zweiten Unterprogramms kehrt nach Ausführung zum ersten Unterprogramm zurück. Auf diese Weise können Unterprogramme verschachtelt werden.

GOTO GOTO expression

```
GOTO 100
GOTO %FF
GOTO B*100
```

GOTO ändert bedingungslos den Programmablauf. Im Gegensatz zum Dartmouth Basic akzeptiert Basic/Debug Ausdrücke, die dem Schlüsselwort GOTO folgen. Diese Funktion ermöglicht eine Variable zur Auswahl einer Zeilennummer. Zum Beispiel, wenn die Variable G 1, 2 oder 3 entspricht, und Zeile 100, 200 oder 300 jeweils ausgeführt werden soll, verwenden Sie die folgende Anweisung:

```
GOTO G*100
```

GOTO wird oft im Direktmodus für interaktives Debuggen verwendet, weil GOTO in den Ausführungsmodus wechselt. Im Gegensatz zum RUN-Befehl kann GOTO die Zeilennummer angeben, in der die Ausführung erfolgen soll.

GOTO muss immer die letzte Anweisung in einer Zeile sein.

IF/THEN IF expression relational_op expression [THEN] statement

```
IF A>B THEN PRINT "A>B"
IF A>B "A>B"            das gleiche, s. PRINT
IF X=Y IF Y=Z PRINT "X=Z"
IF A<>B I=0:J=K+2:GOTO 100
IF 1=2 THEN this part never matters
```

Der IF/THEN-Befehl wird für bedingte Operationen und Verzweigungen verwendet. statement kann eine andere Anweisung sein oder eine Zeilennummer, oder eine Liste von durch Doppelpunkten getrennte Anweisungen. Jede dieser Anweisungen kann ein anderes IF sein. Das Schlüsselwort THEN kann weggelassen werden, um Speicherplatz zu sparen. Ebenso darf GOTO vor einer Zeilennummer entfallen.

IF vergleicht den Wert des ersten Ausdrucks mit dem Wert des zweiten. Wenn die durch den relationale Operator angegebene Beziehung wahr ist, dann werden die folgenden Anweisungen

ausgeführt. Wenn die Beziehung nicht wahr ist, dann wird die nächste Zeile ausgeführt.

Es gibt nur zwei Situationen, in denen das Schlüsselwort THEN nicht weggelassen werden darf: Zum einen darf es nicht weggelassen werden, wenn der zweite Ausdruck mit einer dezimalen oder hexadezimalen Konstante endet und eine Zeilennummer statt einer Anweisung verwendet wird. Zum Beispiel:

```
IF X <1 THEN 1000
```

Die obige Anweisung erfordert ein THEN, um die Zahl vom zweiten Ausdruck von der Zeilennummer zu trennen. THEN kann jedoch durch Umordnen der Ausdrücke aus der Anweisung entfernt werden:

```
IF 1 > X 1000
```

Die zweite Situation, bei der THEN nicht weggelassen werden darf, ist, wenn der zweite Ausdruck mit einer hexadezimalen Konstante endet, und der Anweisungsteil eine LET-Anweisung ist, in der das Schlüsselwort weggelassen wurde und die Variable zwischen A und F liegt. Beispiel:

```
IF Z > %100 THEN A = Z
```

Leerzeichen anstelle des THEN verhindern die Interpretation des Variablen-Buchstabens als Hexadezimalwert nicht, weil Leerzeichen ignoriert werden. THEN muss daher zum Trennen enthalten sein.

INPUT/IN INPUT|IN variable (',' variable)*

```
IN C, E, G
INPUT A
```

Diese Anweisungen geben eine Eingabeaufforderung „?“ aus, und lesen dann die Eingabewerte von der der Tastatur und speichern sie in den angegebenen Variablen. Das sind zwei der drei Anweisungen, die einen Ausdruck einer Variable zuweisen.

Jeder Befehl akzeptiert Werte für eine Liste von einem oder mehreren Variablen. Wenn der Benutzer nicht so viele Werte eingibt, wie benötigt werden, wiederholen beide Befehle die Eingabeaufforderung, bis die erforderliche Anzahl der Werte eingegeben werden. Die Befehle unterscheiden sich in der Art und Weise, wie sie zusätzliche Werte verarbeiten, die vom Bediener eingegeben wurden.

INPUT verwirft alle im Puffer verbleibenden Werte von vorherige IN-, INPUT- oder RUN-Anweisungen und fordert neue Daten vom Nutzer an. IN verwendet erst alle Werte, die im Puffer verblieben sind, und fordert dann neue Daten an.

Im Gegensatz zu Dartmouth Basic akzeptiert Basic/Debug allgemeine Ausdrücke als Eingabe. Es akzeptiert auch Variablen, die bereits ein Wert zugewiesen wurde. Eine Variable, die einen Wert am Anfang der Liste zugewiesen bekam, kann verwendet werden, um später eine weitere Variable in der Liste zu definieren. Beispielsweise kann die Anweisung INPUT C,A als gültige Eingabe 10,C*5 verarbeiten.

Wenn ein Programm vom Bediener die Eingabe einer Liste von Werten anfordert, muss er möglicherweise jedes Element durch ein Komma trennen. Kommas können weggelassen werden,

wenn sie nicht zur direkten Auslegung erforderlich sind. Leerzeichen werden ignoriert. Die folgenden Beispiele zeigen, wie Trennzeichen verwendet werden, um die Interpretation von Eingabewerten zu ändern:

```
    ? %123,A,ND(56)        (hex 123, Variablen A,N,D, dezimal 56)
    ? %12 3AND(56)        (hex 123A, Variablen N, D, dezimal 56)
    ? %123,AND(56)        (hex 123, Wert von 56 mit sich selbst UND-
verknüpft)
```

Da Basic/Debug nur einen Eingabezeilenpuffer hat, werden INPUT und IN im Direkt- und im Run-Modus unterschiedlich ausgeführt. Im Direktmodus überlagert und zerstört die Benutzerantwort den INPUT- oder IN-Befehl, der es angefordert hat. Folglich ist es egal, wie viele Variablen nach dem Schlüsselwort INPUT aufgelistet sind, nur die erste wird den Eingabedaten zugewiesen.

IN kann jedoch im Sofortmodus Listen mit Variablen und Ausdrücken zuweisen, wenn beide Listen abwechselnd in die Befehlszeile. Zum Beispiel:

```
IN A, 10, B, 1 5, C, 20
```

Wenn die obige Zeile im Sofortmodus ausgeführt wird, holt Basic/Debug die erste Variable, A, vom Tastatur-Puffer und rückt den Pufferzeiger vor. INPUT würde an dieser Stelle eine neue Eingabezeile von der Tastatur anfordern, aber IN verwendet erst alle Werte im Puffer, bevor das „?“ ausgegeben wird, kehrt zum Puffer zurück und weist A den Wert 10 zu. Der Vorgang wird fortgesetzt, bis alle Variablen und Werte aufgebraucht sind. Wenn die Befehlszeile mit einer Variablen geschlossen wird, wird das „?“ ausgegeben.

Im Allgemeinen ist es einfacher, LET zu verwenden, um Variablen im Sofortmodus Werte zuzuweisen.

Um dem Nutzer zu helfen, die richtige Anzahl und Art von Werten einzugeben, wird IN und INPUT normalerweise eine PRINT-Anweisung vorangestellt, um die Anforderungen zu beschreiben. Wenn die PRINT-Anweisung mit einem Semikolon abgeschlossen ist, wird der INPUT-Prompt „?“ in derselben Zeile ausgegeben.

Obwohl Basic/Debug keine Zeichenketten unterstützt, kann der INPUT-Befehl verwendet werden, um ein einzelnes Zeichen als Benutzerantwort einzugeben:

```
100 PRINT "BITTE JA ODER NEIN EINGEBEN"
110 LET N=J-1
120 PRINT "VERSTEHEN SIE";
130 INPUT N
140 IF N=J THEN PRINT "GUT!"
```

In diesem Beispiel spielt der Wert von J keine Rolle. Wenn der Benutzer J oder JA eingibt, dann ist die Variable N gleich J. Wenn der Operator N, NEIN oder NOCH NICHT eingibt, dann ist die Variable N unverändert und ungleich J. Um nach anderen Buchstaben als J oder N zu suchen, verwenden Sie einen ungewöhnlichen Wert für J, z. B. -32323, und überprüfen Sie sowohl J als auch J+1 nach der Eingabe.

LET [LET] left_part '=' expression

```
LET A = A+1
```

```
@ 1020 = 100
^8 = %100*C
```

LET weist einer Variablen oder einem Speicherort den Wert eines Ausdrucks zu. Der linke Teil der Anweisung kann ein beliebiges alphabetisches Zeichen A-Z sein, eine Speicherreferenz oder eine Register-Referenz. Der Wert des Ausdrucks wird entweder im Speicherort oder in den Speicherort der Variablen gelegt, und kann bei jedem späteren Auftreten der Variablen verwendet werden. Da das Gleichheitszeichen die Syntax dieses Befehls eindeutig macht, kann das Schlüsselwort LET weggelassen werden.

Der Wert einer Variablen kann unter Verwendung derselben neu berechnet werden wie in der inkrementierenden Anweisung:

```
LETB=B+1
```

LET kann verwendet werden, um Werte im Speicher zu speichern, indem eine Speicherreferenz auf der linken Seite der LET-Zuweisung verwendet wird:

```
LET@1024=B/2
```

Wenn diese Anweisung ausgeführt wird, wird die Speicherreferenz zuerst berechnet, dann wird der Ausdruck ausgewertet und seine Wert gespeichert. Eine Wort-Referenz speichert das höherwertige Byte an der adressierten Stelle. Das niederwertige Byte wird in der nächsthöheren Adresse gespeichert. Seien Sie vorsichtig bei der Änderung interner Register oder des Bereichs, in dem das Programm im Speicher abgelegt ist, weil unsachgemäße Änderungen katastrophale Ergebnisse haben können.

LIST LIST [anfangszeile [',' endzeile]]

Dieser Befehl wird im interaktiven Modus verwendet, um eine Auflistung der gespeicherten Programmzeilen zu generieren. Die optionalen Zeilennummern geben den Zeilenbereich an, die aufgeführt werden. Wenn nur eine Zahl angegeben wird, wird nur diese Zeile angezeigt. Wenn auch eine Endzeile enthalten ist, werden Anfangszeile bis einschließlich Endzeile aufgelistet. Ein LIST-Befehl ohne Argumente listet alle Zeilen des Programms auf.

Der LIST-Befehl wird im Allgemeinen im Sofortmodus verwendet, Es kann jedoch im Ausführungsmodus für einfachen Text verwendet werden wird bearbeitet. Weil Basic/Debug Programmzeilen nach der Zeilennummer bis zur Laufzeit nicht analysiert, kann man Text verarbeiten, wie im folgenden Programm gezeigt:

```
100 REM THIS PROGRAM PRINTS A MESSAGE N TIMES
110 IF N>0 THEN 200
120 : PRINT "HOW MANY TIMES";
130 : INPUT N
200 REM BEGIN LOOP
210 : LET N=N-1
220 : LIST 1000, 1070
230 : IF N>0 THEN 210
240 STOP
1000|  This is a message saved in memory.  It will be
1010|printed when the program is RUN.  If you tried to
```

```
1020|execute lines 1000 to 1070 you would get an error
1030|message.  But in this program, lines 1000+ are not
1040|executed, just LISTed.
1050|
1060| (Signed)
1070|
```

Fünf Zeilen dieses Programms sind eingerückt, um das Programm strukturiert anzuzeigen und um das Lesen zu erleichtern. Der Doppelpunkt verhindert das Entfernen der Leerzeichen vor der ersten Anweisung der Zeile. Wenn das Programm ausgeführt wird, wird die Meldung genau so gedruckt, wie es in den Zeilen 1000-1070 erscheint, einschließlich der vertikalen Striche am linken Rand. Der vertikale Strich wird benötigt, um Zeile 1000 einzurücken; die anderen sind für Konsistenz enthalten. Zusammenfassend: Verwenden Sie einen Doppelpunkt, um eine Anweisung einzurücken, weil Basic/Debug es als Anweisungsbegrenzer erkennt und verwenden Sie den vertikalen Strich, um Textzeilen einzurücken, da dies das am wenigsten ablenkendes Zeichen auf der linken Seite beim Ausdruck ist.

NEW

Der NEW-Befehl setzt den Zeiger 1O-11 auf den Anfang des Benutzerspeichers, wodurch der Speicherplatz als leer markiert wird und bereit ist, ein neues Programm speichern. Wenn dieser Befehl fälschlicherweise eingegeben wird, brauchen Sie keine Panik bekommen, das gespeicherte Programm ist nicht wirklich weg. Obwohl es möglicherweise nicht editiert werden kann, kann es zumindest durch Setzen der Zeilennummer der ersten Zeile auf eine sehr kleine Zahl wieder aufgelistet werden. Benutzen Sie LET-Anweisung im Direktmodus:

```
LET ^^8=1
```

Das Programm scheint nach dieser Wiederherstellung zu funktionieren, es ist jedoch kein Speicher-Überlaufschutz mehr vorhanden, und das Programm kann zerstört werden.

→ vp: Es muss zusätzlich Register 4-5 auf 20h Byte nach Programmende (oder höher) gesetzt werden, dann ist alles korrekt, z.B.:

```
^4=%89FF
```

PRINT PRINT Argument ,|; Argument ...

```
PRINT HEX (255)
"THE ANSWER IS ";X
(A*100)
+%800 + Z
PRINT A, B, C, D, E
```

Der PRINT-Befehl gibt seine Argumente (Texte oder Zahlenwerte) auf dem Bildschirm aus. Die Trennzeichen ',' und ';' steuern die Art, wie die Argumente ausgegeben werden.

In Anführungszeichen eingeschlossene Zeichen und Leerzeichen werden genau so ausgegeben, wie sie eingegeben wurden. Anführungszeichen sind nicht ausgebbar. Wenn ein Text mit einem Zitat unterbrochen werden muss, verwenden Sie stattdessen das einfache Anführungszeichen oder den Apostroph.

Das Schlüsselwort PRINT kann entfallen, wenn die Anweisung mit einer Zeichenkette oder einem Vorzeichen beginnt. PRINT ohne Argument oder Trennzeichen generiert eine Leerzeile. Jeder PRINT-Anweisung kann ein Doppelpunkt und weitere Anweisung folgen.

Wenn ein Ausdruck als Argument für PRINT eingegeben wird, wertet Basic/Debug ihn aus und listet seinen Dezimalwert auf dem Bildschirm. Nur die signifikanten Stellen werden gedruckt, führende Nullen und Divisionsreste nicht. PRINT gibt Zahlen als ganze Zahlen mit Vorzeichen aus. Eine Methode zum Drucken von vorzeichenlosen Werten mittels '\' ist weiter oben dargestellt.

Um einen Hexadezimalwert auszugeben, verwenden Sie die Syntax:

```
PRINT HEX (Ausdruck)
```

Basic/Debug wertet den Ausdruck aus und gibt sein positives hexadezimales Äquivalent aus. Der PRINT-Befehl kann keine negative Hexadezimalzahl auflisten.

Im Gegensatz zu Zeichenketten muss der HEX-Funktion das Schlüsselwort PRINT vorangestellt werden, ebenso wie jedem Ausdruck, der mit einer Variablen beginnt. Das Schlüsselwort kann jedoch vor einem Ausdruck weggelassen werden, wenn dem Ausdruck ein „+“ oder „-“ vorangestellt ist. Zum Beispiel: -10 + 20 oder +20 - 10 als Anweisungen eingegeben gibt den Wert von 10 aus, aber 20 - 10 führt zu einer Fehlermeldung.

Wenn ein Komma verwendet wird, um Elemente in PRINT zu trennen, wird zwischen jedem Element ein Tabulator generiert. Die Tabulatoren befinden sich in Abständen von acht Leerzeichen auf dem Bildschirm. Zum Drucken linksbündiger Spalten schreiben Sie einfach alle Elemente, die auf einer Zeile in einer PRINT-Anweisung gedruckt werden sollen, durch Kommas getrennt in die Anweisung. Das erste Zeichen des Datenelements erscheint in die Spalte, die den Tabulator enthält. Wenn der Ausdruck länger ist als acht Zeichen, geht Basic/Debug bis zum nächsten verfügbaren Tabulator, um das nächste Element zu drucken.

Um einen Argument direkt nach dem anderen ohne Abstand zu drucken, verwenden Sie ein Semikolon als Trennzeichen. Zum Beispiel gibt

```
PRINT"AUSGABE=";X
```

den Wert der Variablen X direkt nach dem Gleichheitszeichen aus. Wird eine PRINT-Anweisung mit einem Semikolon beendet, wird kein abschließender Wagenrücklauf-Zeilenvorschub erzeugt. Das nächste Argument einer nachfolgenden PRINT-Anweisung erscheint in derselben Zeile wie das Argument, das vor dem Semikolon steht. Ein Komma am Ende der PRINT-Anweisung unterdrückt auch den Wagenrücklauf, jedoch erscheint das nächste zu druckende Element am nächsten Tabstopp.

Um rechtsbündig ausgerichtete Spalten zu drucken, müssen führende Leerzeichen hinzugefügt werden. Basic/Debug kann nur in Anführungszeichen eingeschlossene Leerzeichen drucken. Das folgendes Beispielprogramm fügt führende Leerzeichen zu N hinzu:

```
200 IF N<10000 THEN PRINT " ";
210 IF N<1000 THEN PRINT " ";
220 IF N<100 THEN PRINT " ";
230 IF N<10 THEN PRINT " ";
240 PRINT N
```

Basic/Debug kann die meisten Steuerzeichen drucken, wie z.B. den Signalton (bell, ^G), wenn sie in einer Zeichenkette in Anführungszeichen enthalten sind. Die folgende Steuerzeichen können nicht gedruckt werden:

```
Rückschritt (^H)
Escape      (ESC)
Wagenrücklauf   (CR)
Zeilenvorschub  (LF)
löschen     (DEL)
Null        (NUL)
```

Der Zirkumflex „^“ zeigt an, dass die Strg-Taste gedrückt bleibt, während die angegebene Taste gedrückt wird. Wenn Steuerzeichen gedruckt werden, kann es sein, dass der Cursor-Zeiger von Basic/Debug nicht mehr die korrekte Position des Bildschirm-Cursors enthält. Das Drucken in Spalten mit Kommatrennzeichen schlägt dann fehl. Zum Beispiel:

```
5 X=0
10 PRINT "X^G", X
20 PRINT "X", X
```

Wenn das obige Programm ausgeführt wird, erscheint folgende Ausgabe:

```
X      0
X       0
```

Die Anweisung in Zeile 10 fügt nur sieben Leerzeichen ein, weil das Control-G-Zeichen den Basic/Debug-Cursor schon um eine Stelle nach rechts von der aktuellen Cursorposition gesetzt hat.

REM REM Kommentar

```
REM CONTROL-SCHLEIFE
REM UNTERPROGRAMM NAME
REM CODE-ERKLAERUNG
```

Der REM-Befehl wird verwendet, um Kommentare, Anmerkungen oder andere erklärende Nachrichten in den Code einzufügen. Basic/Debug ignoriert alles, was dem REM-Schlüsselwort folgt, daher muss REM und sein Kommentar der letzte Befehl in einer Zeile sein. Der gemäßigte Gebrauch von Anmerkungen in einem Programm erleichtert das Lesen und Pflegen. Bemerkungen nehmen jedoch Platz im Speicher ein und sollten für maximale Platzausnutzung weggelassen werden.

RETURN RETURN|RET

```
RETURN
RET
```

RETURN ist immer die letzte Anweisung eines Unterprogramms und kann als RET abgekürzt werden. Es braucht kein Argument, weil GOSUB die nächste Zeilennummer speichert, die nach RETURN auszuführen ist. RETURN muss die letzte Anweisung in einer Zeile sein.

Wenn ein Unterprogramm ein anderes aufruft, kehrt der RETURN-Befehl am Ende des zweiten Unterprogramms zum ersten zurück. Auf diese Weise können Unterprogramme so tief verschachtelt

werden, wie der für den GOSUB-Stack verfügbaren Speicher reicht.

RUN RUN [Ausdruck ',' Ausdruck ...]

```
RUN
RUN 17, %200, 23
```

Dieser Befehl initiiert die sequentielle Ausführung aller Anweisungen, die im Speicher abgelegt sind. RUN wird nur im Sofortmodus genutzt. Datenwerte für den ersten IN-Befehl können durch Kommas getrennt dem Schlüsselwort RUN folgen:

```
RUN 45,-583
```

STOP STOP

```
STOP
```

STOP beendet die Programmausführung ordnungsgemäß und löscht den GOSUB-Stack. Eine STOP-Anweisung erfolgt automatisch nach der letzten Programmzeile, daher kann ein beendender STOP-Befehl aus dem Programm weggelassen werden, um Speicherplatz zu sparen.

Die Programmausführung wird oft durch einen Fehler abrupt beendet. Nach dem Ändern der fehlerhaften Anweisung im Direktmodus kann der Nutzer den Lauf neu starten, indem er GOTO mit den entsprechenden Zeilennummer nutzt, oder Sie setzen das Programm mit einem STOP-Befehl zurück, und starten das Programm mit RUN erneut von Anfang an.

Fehler

Fehler treten auf, wenn Basic/Debug eine Anweisung nicht versteht. Ein Fehler bringt das System in den Direktmodus zurück. Alle Variablen und der GOSUB-Stack bleiben dabei unverändert. Es wird eine Fehlermeldung ausgeben. Fehlermeldungen erscheinen am Terminal im folgenden Format:

```
Fehlercode AT Zeilennummer
```

Die numerischen Fehlercodes sind unten aufgelistet. Wenn der Fehler aufritt, während ein Programm läuft, enthält die Fehlerausgabe eine Zeilennummer. Gibt es Fehler im Direktmodus, wird keine Zeilennummer aufgelistet. Ein Fehler tritt auf, wenn das Schlüsselwort oder Argument nicht erkennbar ist oder nicht ausführbar ist, oder im Falle einer IN- oder INPUT-Anweisung, wenn die Dateneingabe durch den Betreiber unverständlich ist. Ein ^G (Strg-G, Bell) wird mit der Fehlermeldung an den Bildschirm gesendet.

Fehler (Z8671):

```
11  Program line has a line number 0 or greater than 32768.
17  Memory full; new line not inserted.
26  No program to RUN.
37  GOTO is not at the end of program line.
41  Cannot GOTO negative or zero line number.
44  Line number in GOTO does not exist.
```

```
66  GOSUB is not at the end of the line.
71  Unrecognizable statement type beginning with GO.
81  Unrecognizable statement type, or '=' missing from LET statement.
98  LET is missing its '='.
140 Quote missing in PRINT statement.
171 RETURN is not at the end of the line.
172 GOSUB stack underflow.
175 The GOSUB for this RETURN no longer exists.
181 STOP is not at the end of the line.
207 INPUT variable name is missing.
210 IN or INPUT expects variable name.
247 LIST is not at end of line.
310 Unrecognizable relation in IF statement.
346 Out of memory on GOSUB or expression evaluation.
381 Divide by zero.
391 Missing parenthesis in AND or USR call.
427 Syntax error in expression, or unrecognizable statement type.
431 Missing right parenthesis in expression.
```

Speichernutzung

Standard: 1020 (Basic-Programm) bis FFEF (Arbeitspeicher)

```
xx=10, yy=FF
```

Wenn nur ROM vorhanden ist, so ist xx=0 und yy=0. Dann werden nur interne Register genutzt. In diesem Fall sind nur die Variablen A..L sicher nutzbar. M..Z wird vom Gosub-Stack mitgenutzt.

```
; yy = hi(highest RAM), if no RAM, yy=0 means internal registers
; xx = hi(BASIC-Pgm)

; yyF1 - yyFF Unused.
; yy68 - yyF0 Input line buffer, used for editing in immediate mode and
;       user response to IN or INPUT request in run mode.
;       Reg. 68-7F im ROM-Mode Input-Buffer sowie Expression Evaluation
stack.
;       Expression Evaluation stack grows from 7F (hex) down, and the line
;       buffer grows from 68 (hex) up.
; yy56 - yy67 Unused.
; yy54 - yy55 Storage for variable z.
; yy53 - yy52 Storage for variable Y.
; ..
; yy21 - yy22 Storage for variable A.
; yy20  Base of GOSUB stack. Stack grows down to lower memory addresses,
;   and may extend until it reaches the top of the user's Basic/Debug
program.
;       Reg. 40-67 im ROM-Mode GOSUB-Stack

; Register
```

```
----------

; 04-05 => moves up from xx00    High boundary of user program plus stack
reserve (20h).
; 06-07 => moves down from yy20  Low boundary and top of GOSUB stack.
; 08-09 => xx00                  Bottom of user memory; first line of user
program.
; 0A-0B => yy20                  Top of user memory, high boundary of GOSUB
stack.
;                Initially set to yy20 of high page of RAM.
; 0C-0D => yy68 to yyF0          Last character entered in line buffer.
Backspace
;                subtracts one from this pointer; escape resets it
;                to the beginning of the buffer. Rl2 is the page
;                number for variables and the input buffer.
; 0E-0F => yy68 to yyF0           Next value to be used from line buffer.
INPUT
;                command resets to the beginning of the buffer;
;                IN uses all values in the buffer before resetting.

; 10-1F     internal Basic/Debug
; 20        current cursor location
; 40-7F     Expression Evaluation Stack
```

Programmformat

Programmzeilen werden im Folgenden Format gespeichert:

```
06 60       L I S T            0
Zeilennummer    Anweisung in ASCII    Null
in binär
```

Das Ende des Programms im Speicher wird durch FFFF (hex) gekennzeichnet. Beispiel:

```
10 I=5
20 PRINT "WELCOME TO BASIC/DEBUG"
30 I=I-1: IF I>0 GOTO 20
40 STOP

8800h: 00 0A 49 3D 35 00 00 14 50 52 49 4E 54 22 57 45 ; ..I=5...PRINT"WE
8810h: 4C 43 4F 4D 45 20 54 4F 20 42 41 53 49 43 2F 44 ; LCOME TO BASIC/D
8820h: 45 42 55 47 22 00 00 1E 49 3D 49 2D 31 3A 20 49 ; EBUG"...I=I-1: I
8830h: 46 20 49 3E 30 20 47 4F 54 4F 20 32 30 00 00 28 ; F I>0 GOTO 20..(
8840h: 53 54 4F 50 00 FF FF 00 00 00 00 00 00 00 00 00 ; STOP.ÿÿ.........
```

Direkte Zeichen-Ein-Ausgabe

Basic/Debug-Programme können durch Aufrufen der direkten Ein-/Ausgabetreiber via USR- oder GO@-Anweisung Binärdaten lesen und schreiben.

```
input   orig Z8671  %54
output  orig Z8671  %61
```

Das folgende Beispielprogramm druckt das Hex-Äquivalent eines ASCII-Zeichens

```
10 PRINT "INPUT A CHARACTER, PLEASE";
20 C = USER (%54)
30 PRINT" THE HEX VALUE OF ";
40 GO@ %61, C
50 PRINT" IS "; HEX (C);". SHALL WE DO ANOTHER?";
60 Q = USER (%54)
70 PRINT : IF Q = %59 GOTO 10
80 REM %59 IS AN ASCII "Y".
```

Mastermind

Beispielprogramm nach „Zilog z8671-7-chip-computer“

Erläuterungen

- Zeile 20 initialisiert den Zähler T2 für Zufallszahlen
- Zeile 30 erzeugt 4 Zufallszahlen. Dazu ist 4x eine beliebige Taste zu drücken (USR(84) = char input)
- ^10+2 = YY22 Pointer auf Variable A
- THEN, LET, PRINT vor „“, GOTO vor Zahl nach IF, Leerzeichen können entfallen,
  Für Tempo und Lesbarkeit sollten sie stehen.
- „Y/N“:INPUT X - Eingabe von Variablen ist zulässig, „Y“ wird als Variable ausgewertet
  Die nachfolgende Zeile vergleicht X mit dem Wert Y. Gute Idee!

```
10 REM MASTERMIND
20 @243=7:@242=10:@241=14               Init Timer
40 X=USR(84):A=@242-1:X=USR(84):B=@242-1:       jeweils Warten auf
Tastendruck, A..D=Zufallszahl 0..9
   X=USR(84):C=@242-1:X=USR(84):D=@242-1
50 "":I=0                      I=Anzahl Versuche
100 "GUESS ",:IN E,F,G,H              Eingabe 4 Werte (kommagetrennt)
110 I=I+1
300 J=^10+2:K=J+8                  J zeigt auf A, K zeigt auf E
301 L=0                          L Zähler 1..4
302 R=0:P=0                      R Zähler richtige Zahl, P Zähler richtiger
Platz
310 IF ^J=^K LET P=P+1                  Test auf gleiche Zahl, gleicher
```

```
Platz
320 J=J+2:K=K+2:L=L+1:IF 4>L GOTO 310           nächste Plätze (J zeigt auf
B , K auf F etc.), next L
330 J=^10+2:K=J+8                   J zeigt auf A, K zeigt auf E
331 L=0                          L Zähler 1..4
340 IF ^J=^K LET R=R+1:^J=^J+10:L=3         Wenn A=E, dann Zähler richtige
Zahl +1, A=A+10 setzen (ausblenden),L auf Ende
341 J=J+2                        nächste Plätze
350 L=L+1: IF 4>L GOTO 340               next L
351 J=^10+2                      J zeigt auf A
352 L=0                          L Zähler
360 K=K+2:IF ^10+17>K GOTO 340              nächster Platz K auf F etc.,
next
363 J=^10+2:K=J+8                   J zeigt auf A, K zeigt auf E
366 IF ^J>9 LET ^J=^J-10             ggf. wieder 10 abziehen
367 J=J+2                        nächster Platz
368 IF ^10+9>J GOTO 366                  Ende nach D
369 Y=1:N=0                      Yes/No-Abfrage init.
370 "RIGHT ";R;" PLACE ";P              Anzeige Auswertung
380 IF 4>P GOTO 100                 wenn weniger als 4 am richtigen Platz,
weiter raten
390 X=0:Y=1                      Yes/No-Abfrage vorbereiten
400 "RIGHT IN ";I;" GUESSES;";"PLAY ANOTHER Y/N": Anzeige Anzahl Versuche
    INPUT X                      Eingabe Variablenname Y oder N
410 IF X=Y GOTO 10                  Wenn Eingabe == 1 (Wert von Y), dann
neues Spiel

:RUN
1111                        Zufallszahlen init
GUESS   ? 1,2,3,4
RIGHT 2 PLACE 0
...
GUESS   ? 3,7,1,0
RIGHT 4 PLACE 4
RIGHT IN 15 GUESSES;PLAY ANOTHER Y/N
? N
:
```

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

Permanent link:
https://hc-ddr.hucki.net/wiki/doku.php/elektronik/z8671/handbuch

Last update: **2024/02/14 11:13**