Unterprogramme in Maschinesprache für Microsoft BASIC Interpreter für MS-DOS

Martin Hepperle, 2024

Nachdem Microsoft seine BASIC Interpreter für viele CP-M Systeme mit 8-bit CPUs und 64 kB Speicher entwickelt hatte, wurde das System auch auf Intel Prozessoren mit 16-bit Registern und MS-DOS übertragen. Bei diesen Versionen wurde das BASIC Grundsystem beibehalten. Durch Nutzung einer gemeinsamen, 8-bit basierten, Code-Basis und der segmentierten Architektur der neuen Intel Prozessoren, war es Microsoft sehr schnell möglich, ein BASIC für diese Systeme zur Verfügung zu stellen. Die Möglichkeit der neuen Rechner, bis zu 1 MB Speicher anzusprechen, wurde dabei aber nicht ausgenutzt. Es erfolgte lediglich eine Trennung des BASIC Systems in einen Code- und einen Datenbereich, die nun jeweils 64 kB groß sein durften. Der restliche Speicher des Rechners lag brach (er kann, allerdings mühsam, byteweise durch `DEF SEG` und `PEEK`/`POKE` Anweisungen adressiert werden).

Manche Funktionen kann man in BASIC nur sehr mühsam oder gar nicht implementieren. Manchmal ist auch die vorzeichenbehaftete Integer Zahlendarstellung in BASIC hinderlich. In solchen Fällen kann es sinnvoll sein, die gewünschte Funktion an ein Unterprogramm in Maschinensprache zu übertragen. Der Aufruf eines Unterprogramms in Maschinensprache ist aber immer mit einem gewissen Zusatzaufwand verbunden. Deshalb ist es oft schneller, kleine Aufgaben direkt in BASIC zu erledigen – die Multiplikation einer Integer Zahl mit der Konstanten 2 läuft in BASIC schneller ab, als extra ein Unterprogramm aufzurufen, das diese Funktion durch ein auf CPU-Ebene schnelleres linksschieben um eine Bitposition durchführt. Beim Aufruf müssen Register gesichert, Parameter übergeben und bei der Rückkehr zu BASIC die gesicherten Register wieder hergestellt werden. All dies kostet Zeit. Man muss im Zweifelsfall ausprobieren, welch Variante die schnellere oder kleinere ist.

Es gibt mehrere Möglichkeiten, Unterprogramme in Maschinensprache mit dem BASIC Interpreter zusammen im Rechner zu verwalten.

Hier werden zwei Möglichkeiten gezeigt, wie man Unterprogramme in Maschinensprache laden und von BASIC aus ansprechen kann.

Methode A: Routinen innerhalb des BASIC Datensegments

Als erste Variante können die Bytefolgen der Maschinensprache-Routinen mit `READ` Befehlen aus `DATA` Statements gelesen und durch `POKE` Befehlen in eine, meist als `INTEGER` Feld deklarierte, Variable abgelegt werden. Dazu wird die Anfangsadresse dieser Variable mit `VARPTR` ermittelt und anschließend werden die Code-Bytes einzeln in die folgenden Speicherzellen kopiert.

Die Routine an dieser Adresse kann später beliebig oft mit `USR` oder `CALL` Befehlen aufgerufen werden. Ein Vorteil dieser Variante ist, dass der Code vollständig im BASIC Programm untergebracht ist und keine externen Dateien benötigt werden. Nachteilig ist, dass man die kryptischen `DATA` Statements manuell oder mit einem Extra-Programm erzeugen und in das BASIC Programm integrieren muss. Außerdem belegen die `DATA` Anweisungen zusätzlich zum `INTEGER` Feld Speicherplatz im BASIC Programmbereich.

Als zweite Variante kann man anstelle der mühsamen Kombination von `DATA` Statements und Verwendung der `READ` und `POKE` Befehle den Code auch mit dem `BLOAD` Befehl aus einer Datei in das `INTEGER` Feld einlesen. Damit bleibt der BASIC Code kürzer und muss (bis auf die Dimensionierung des Felds) nicht mehr verändert werden, wenn die Maschinensprache-Routinen geändert werden. Die Umwandlung in das `BLOAD` Format kann mit einem kleinen Hilfsprogramm erfolgen [1], oder einfacher, schon beim Schreiben des Assembler-Quellcodes berücksichtigt werden. Es muss aber zur Laufzeit zusätzlich zum BASIC Programm auch die Datei mit dem Maschinencode bereitgestellt werden.

Eine dritte Variante schließt einen Bereich am oberen Ende des BASIC Datensegments von der Benutzung durch BASIC aus. Hierzu wird die Option /M beim Start des BASIC Interpreters verwendet, um eine obere Endadresse im Datensegment abzusenken. Dann kann man den Code ebenfalls mit dem `BLOAD` Befehl aus einer Datei in diesen fest definierten Bereich einlesen und braucht keine Feldvariable dafür zu definieren. Auch hier muss zur Laufzeit zusätzlich die Datei mit dem Maschinencode bereitgestellt werden.

Nachteil aller drei Varianten dieser Methode „A“ ist, dass der Maschinencode Speicher im BASIC Datensegment belegt, wodurch weniger Speicher für BASIC Daten zur Verfügung steht. Außerdem ist besonders zu beachten, dass BASIC bei Anlage neuer einfacher Variablen (keine Felder) die Position von bereits definierten Feldern verschiebt. Deshalb darf man die Adresse der Maschinencode-Routinen im `INTEGER` Feld erst dann mittels `VARPTR` ermitteln, wenn alle einfachen Variablen definiert, d.h. wenigstens einmal mit einem Wert besetzt wurden. Am sichersten ist, die Adresse erst unmittelbar vor dem Aufruf mit `CALL` oder `USR` ermitteln oder alle einfachen Variablen vorab mit einem beliebigen Wert zu initialisieren.

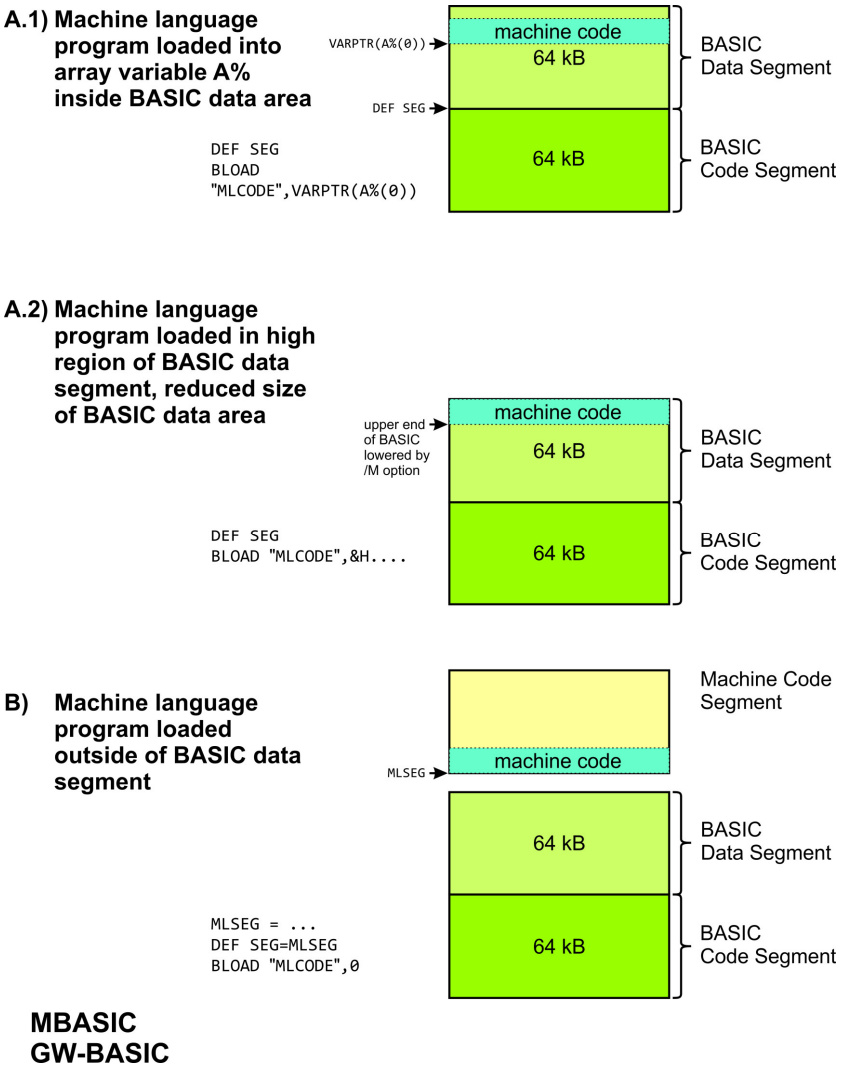

Abbildung 1: Möglichkeiten, Routinen in Maschinensprache in den Speicher zu platzieren. A) innerhalb und B) außerhalb des BASIC Speicherbereichs.

Methode B: Routinen oberhalb des BASIC Datensegments

Um den Möglichkeiten der 16-bit Intel Prozessoren entgegen zu kommen, wurde für die MS-DOS Versionen des BASIC ein neues Schlüsselwort `DEF SEG` eingeführt. Damit lässt sich eine Segmentadresse für die nachfolgenden Aufrufe bestimmter Funktionen (`PEEK`, `POKE`, `CALL`, `CALLS`, `USR`, `BLOAD`, `BSAVE`) festlegen. Dieses Segment kann auch außerhalb des BASIC Datenbereichs liegen. Außerdem wurde noch das Schlüsselwort `CALLS` zugefügt, bei dem Adressen der Parameter als *Segment:Offset* Paar anstelle der reinen 16-bit *Offset* Adressen beim `CALL` übergeben werden.

Mit Hilfe von `DEF SEG` ist es so möglich, Unterprogramme in Maschinensprache zu verwenden, ohne dass dafür Platz im knappen BASIC Datenbereich verschwendet werden muss.

Dazu kann man solche Unterprogramme in den Hauptspeicher außerhalb des BASIC Datensegments bringen. Wesentlich ist dabei, dass man deren Adresse kennt und dass sich der verwendete Speicherbereich nicht mit dem vom BASIC System oder dem MS-DOS Betriebssystem verwendeten überschneidet. Schon bei einem kleinen MS-DOS System mit Minimalausstattung steht in der Regel genügend Platz oberhalb der beiden von BASIC genutzten 64-kB Segmente zur Verfügung um dort auch größere Unterprogramme unterzubringen.

Von Vorteil ist, dass dieser Speicherbereich nicht von BASIC beeinflusst wird, sodass nie eine Verschiebung der Funktionsadressen auftritt. Man muss die Adresse der Unterprogramme mit der `VARPTR` Funktion also nicht mehr nach Definition der letzten einfachen Variablen vor den `CALL` oder `CALLS` Aufruf platzieren.

Ein Risiko, das noch besteht, ist, dass man bei sehr kleinen Systemen mit 256 kB Speicher womöglich Treiber am oberen Ende des Speicher überschreiben könnte. Dies wäre zwar auch überprüfbar, wurde hier aber wegen der geringen Wahrscheinlichkeit auf so ein System zu treffen, ignoriert.

Der erste Schritt ist, herauszufinden, welchen Speicherbereich das BASIC selbst verwendet. Je nach Systemkonfiguration kann das unterschiedlich sein.

Wenn man die undokumentierten BASIC Datenstrukturen durchforscht, findet man aber eine Speicherstelle, an der die Adresse des BASIC Datensegments abgelegt ist. Diese ist bei den beiden weit verbreiteten BASIC Versionen MBASIC 5.28 und GWBASIC 3.32 unterschiedlich, aber auf die gleiche Weise nutzbar.

Alternativ kann man auch ein kleines Unterprogramm in Maschinensprache verwenden um das genutzte Datensegment zu bestimmen.

Wenn man dann die 64 kB Größe des BASIC Datensegments berücksichtigt, kann man ausrechnen, wo der freie Speicher oberhalb BASIC beginnt und sein Maschinenspracheprogramm dort platzieren.

Um die umständliche `POKE` Sequenzen zu vermeiden, verwende ich den `BLOAD` Befehl. Dieser lädt eine Binärdatei in einem speziellen Format in einen wählbaren Speicherbereich.

Für den eigentlichen Aufruf verwende ich das Schlüsselwort `CALL` weil es beliebig viele Parameter erlaubt. Die `USR` Funktion bietet nur die Übergabe eines einzigen Parameters. Der einzige Vorteil von `USR` ist, dass man sie als Funktion mit einem Rückgabewert aufruft, was etwas elegantere Programme ergeben kann und dass `USR` auch in sehr alten Microsoft BASIC Versionen vorhanden ist.

Für die Erzeugung eines Unterprogrammmoduls sind folgende Schritte notwendig:

- Schreiben des `.ASM` Assembler Quellcodes der eines oder mehrere Unterprogramme enthalten kann. Dabei wird die Ladeadresse mir `ORG 100H` festgelegt.
- Übersetzen mit einem Assembler in eine `.OBJ` Objektdatei. Ich verwende hier den Microsoft `MASM`.
- Binden der `.OBJ` Datei in eine `.COM` Datei, was z.B. direkt mit Microsoft `LINK 5.15` möglich ist. Alternativ kann man auch mit älteren `LINK` Versionen arbeiten; diese erzeugen eine `.EXE` Datei, die man dann mit `EXEBIN` in eine `.COM` Datei umwandeln muss.
- Die Sequenz `MASM` und `LINK` kann man leicht in ein `BATCH` Skript verpacken.

Beim Programmieren kann man folgendermaßen vorgehen:

- Am Anfang des BASIC Programms ermittelt man eine Segmentadresse oberhalb des BASIC Bereichs und lädt dann mit `BLOAD` die `.BIN` Datei an diese Stelle. Dies muss nur einmal beim Start des Programms erfolgen. Die Segmentadresse wird später noch benötigt.
  Um eine Adresse oberhalb des BASIC Datensegments zu erhalten, kann man die interne BASIC Systemvariable `SAVSEG` verwenden und `&H1001` dazu addieren (jedes Inkrement der Segmentadresse erhöht die physikalische Adresse um `16` Bytes, sodass `16×&H1000 = 65535` Bytes und als Sicherheitspuffer wird noch ein Paragraph von 16 Bytes zugefügt).
  Je nach BASIC Version kann man die Segmentadresse so festlegen:

```
330 REM ================================================
340 REM --- MSBASIC 5.28 für MS-DOS (interne BASIC Variable SAVSEG an &H4D0)
```

```
350 PRINT "Für MS BASIC 5.28" : A=&H4D0 : GOTO 400
360 REM ================================================
370 REM --- GWBASIC 3.23 für MS-DOS (interne BASIC Variable SAVSEG an &H496)
380 PRINT "Für GW BASIC 3.23" : A=&H496 : GOTO 400
390 REM
400 MLSEG=PEEK(A)+256*PEEK(A+1)+&H1001
410 . . .
```

- Anschließend definiert man für jedes Unterprogramm eine Variable mit einem Offset in diesen Speicherblock. Wenn man am Anfang des Assemblerprogramms eine Sprungtabelle zu den einzelnen Routinen einfügt, lassen sich diese Offsets ganz einfach durch systematisches inkrementieren um die Länge eine Sprungbefehls (3 Bytes) ermitteln. Der kleine Nachteil, dass ein weiterer Sprung eingeführt wird, wird durch eine klarere und weniger fehleranfällige Programmstruktur aufgewogen.
  Auch dies muss nur einmal beim Start des Programms erfolgen.
  Im BASIC Programmbeispiel mit zwei Funktionen sieht das so aus:

```
460 DEF SEG=MLSEG
470 BLOAD "BITS.BIN",0
480 REM ----- Position in der Sprungtabelle
490 LSHIFT=0
500 RSHIFT=3
```

  Der Anfang eines dazu passenden Assemblermoduls mit mehreren Funktionen sieht dann so aus:

```
; ------------------------------------------------
ENTRY:
;      Sprungtabelle, falls mehr als eine Funktion in diesem
;      Modul implementiert sind
       JMP NEAR PTR LSHIFT  ; Offset = 0  jeder NEAR JMP belegt 3 Bytes
       JMP NEAR PTR RSHIFT  ; Offset = 3
```

- Um eine Routine aufzurufen, setzt man mit `DEF SEG =` die Segmentadresse für den folgenden `CALL`. Die Parameter müssen jeweils Variablen sein, weil diese immer als Adressen übergeben werden. Man kann also keine Konstanten wie in `CALL SUB(1)` verwenden, sondern muss `I%=1 : CALL SUB (I%)` verwenden.
  Eine typische Aufrufsequenz für die erste Funktion `LSHIFT` sieht dann so aus:

```
640 REM Wähle Code-Segment für CALL
650 DEF SEG=MLSEG
660 A%=16383
670 PRINT A%;" << 1 = ";
680 CALL LSHIFT(A%)
690 PRINT A%
```

Zusätzlich ist es wichtig, die folgenden Punkte zu beachten:

- Die Intel Prozessoren verwenden einen *dekrementierenden* Stapel, d.h. wenn ein Wert mit einem `PUSH` „auf“ den Stapel kopiert wird, wird zuerst der Inhalt des Stapelzeigers `SP` um die Größe des Werts (z.B. 2 Bytes bei einem 16-bit Offset) verringert und dann der Wert an der neuen Adresse abgelegt. Umgekehrt kopiert ein `POP` den Wert vom Stapel und erhöht danach den Stapelzeiger `SP` entsprechend. `SP` zeigt also immer auf den zuletzt auf den Stapel gelegten („oben“ liegenden) Wert.
- Die Unterprogramme werden mit einem `FAR CALL` aufgerufen. Sie müssen deshalb im Assembler-Code auch als `PROC FAR` deklariert werden, damit die `RET` Anweisung den Stapel korrekt hinterlässt.
- Parameter werden als 16-bit Offset in das BASIC Datensegment auf dem Stack übergeben. Sie werden beim Aufruf mit `CALL` von links nach rechts auf dem Stapel abgelegt, gefolgt vom 16-bit Programmsegment `CS` und dem 16-bit Programmzähler `IP` für den Rücksprung. Beim Eintritt in das Unterprogramm sieht der Stapel also so aus:

```
Stapel
| ... hohe Adressen
|       +6  A% Offset Adresse 1. Parameter
|       +4  B% Offset Adresse 2. Parameter
|       +2  Rücksprung Adresse Segment CS, von BASIC mit CALL FAR auf den Stapel
| SP -> +0  Rücksprung Adresse Offset  IP, von BASIC mit CALL FAR auf den Stapel
v ... niedrige Adressen
```

- Bei der Rückkehr zum BASIC Programm muss das Maschinenspracheprogramm den Stapel wieder „aufgeräumt“ übergeben, d.h. wenn beispielsweise zwei Parameter übergeben wurden muss mit einem `RET 4` zurückgekehrt werden (es werden die 2×2 Bytes der Parameter-Adressen vom Stapel genommen und dann mit einem `RET FAR` zu der noch auf dem Stapel liegenden Stelle `CS:IP` zurückgesprungen).
- Die Register `DS`, `ES`, `SS` enthalten stets die Adresse des BASIC Datensegments. `CS` ist auf die Segmentadresse der Maschinensprache-Routine gesetzt. Das Unterprogramm muss die Register `BP`, `DS`, `ES`, `SS` bewahren, d.h. vor einem Rücksprung wieder herstellen.
- Numerische Variablen als Parameter werden als Adresse der BASIC Zahlenrepräsentation übergeben. Zum Beispiel eine normale Integer-Variable:

```
Integer Parameter: Adresse -> 2 Bytes Integer Wort
```

- Zur Übergabe von Feldern verwendet man das erste Element des Felds. Das Unterprogramm erhält damit dessen Adresse und die weiteren Feldelemente liegen dahinter. Bei mehrdimensionalen Feldern variiert der erste Parameter zuerst. Zu beachten ist, das Fließkommazahlen im „Microsoft Binary Format“ (MBF) abgebildet werden und nicht dem IEEE-754 Standard entsprechen.
- Wenn Zeichenketten an ein Unterprogramm übergeben werden, wird die Adresse einer Struktur („String Descriptor“), bestehend aus einem Byte für die aktuelle Länge der Zeichenkette, gefolgt von der 16-bit Adresse der Zeichenkette übergeben. Die definierte maximale Länge einer Zeichenkette ist darin nicht enthalten.

```
Zeichenketten Parameter: Adresse -> 1 Byte Länge (max. 255)
                                    2 Bytes Adresse der Zeichenkette
```

  Zeichenketten dürfen gelesen und auch beschrieben werden, jedoch darf ihre aktuelle Länge auf keinen Fall verändert werden. Das bedeutet, dass sie vom aufrufenden Programm in der passenden Länge vorbelegt werden müssen, wenn das Unterprogramm Zeichen zurückschreiben soll. In den Funktionen `BINSTR` und `HEXSTR` wird deshalb zuerst überprüft, ob die übergebene Zeichenkette die richtige Länge hat.

  Zu beachten ist auch, dass Zeichenketten, die statisch direkt im Programm definiert werden, auch im Programm und nicht im separaten Zeichenkettenbereich gespeichert bleiben:

```
100 A$="Hello World"
110 CALL UPC$(A$)
```

  Beim Ablauf des Programms verändert die `UPC$` Funktion den Text im Programm und ein folgendes `LIST 100` gibt dann die modifizierte Zeile aus:

```
100 A$="HELLO WORLD"
```

  Das Unterprogramm verändert also den Programmtext. Wenn dies nicht erwünscht ist, muss die Zeichenkette z.B. durch Zusammenhängen zweier Teilketten dynamisch im BASIC Zeichenkettenbereich angelegt werden:

```
100 A$="Hello"+" World"
110 CALL UPC$(A$)
```

  Ein folgendes `LIST 100` gibt weiterhin die originale Zeile aus:

```
100 A$="Hello"+" World"
```

Beispielroutinen BITS für MSBASIC 5.28 und GWBASIC 2.32

Die folgenden Unterprogramme sind in den Beispielen `BITS1.ASM` und `BITS2.ASM` enthalten. Dazu gehören die Testprogramme `BITS1.BAS` und `BITS2.BAS`. Diese demonstrieren die beiden, oben erklärten Möglichkeiten, Maschinenspracheprogramme in BASIC zu integrieren.

```
CALL LSHIFT(A%)
```

Schiebt den 16-bit Wert `A%` um 1 Bit nach links. Das rechte Bit 0 wird mit Null besetzt.

```
CALL RSHIFT(A%)
```

Schiebt den 16-bit Wert `A%` um 1 Bit nach rechts. Im Gegensatz zum BASIC `\` Operator wird das Vorzeichenbit 15 nicht kopiert, sondern zu Null gesetzt.

```
CALL BITAND(A%,B%)
```

Bitweises Und. Setzt den 16-bit Wert `A%` auf (`A%` AND `B%`). Dasselbe kann mit dem `AND` Schlüsselwort in BASIC erreicht werden.

```
CALL BITIOR(A%,B%)
```

Bitweises Inklusives Oder. Setzt den 16-bit Wert `A%` auf (`A%` OR `B%`). Dasselbe kann mit dem `OR` Schlüsselwort in BASIC erreicht werden.

```
CALL BITEOR(A%,B%)
```

Bitweises Exklusives Oder. Setzt den 16-bit Wert `A%` auf (`A%` XOR `B%`). Dasselbe kann mit dem `XOR` Schlüsselwort in BASIC erreicht werden.

```
CALL LROT(A%)
```

Rotiert die Bits im 16-bit Wert `A%` um 1 Bit nach links. Das linke Bit 15 wandert nach Bit 0.

```
CALL RROT(A%)
```

Rotiert die Bits im 16-bit Wert `A%` um 1 Bit nach rechts. Das rechte Bit 0 wandert nach Bit 15.

```
CALL FLIP(A%)
```

Invertiert die Bits im 16-bit Wert `A%`. '1' Bits werden zu '0' und umgekehrt. Dasselbe kann mit dem `NOT` Schlüsselwort in BASIC erreicht werden.

```
CALL GETSEG(CS1%,CS2%,DS%,ES%,SS%)
```

Gibt Code, Data und Extra Segment Register zurück.

das erste Code Segment `CS1%` ist das BASIC Code Segment, `CS2%` ist das Segment das von der Maschinensprache Routine verwendet wird (vorher mit `DEF SEG` gesetzt).

```
CALL PEEK16(SEG%,OFF%,VAL%)
```

Gibt in `VAL%` das 16-bit Wort an `SEG%:OFF%` zurück. Arbeitet ähnlich wie zwei PEEKs und ein vorangegangenes `DEF SEG SEG%`.

```
CALL POKE16(SEG%,OFF%,VAL%)
```

Kopiert das 16-bit Wort `%VAL` an die Speicherstelle `SEG%:OFF%`. Ähnlich wie zwei aufeinander folgende POKE Befehle mit vorangegangenem `DEF SEG SEG%`.

```
CALL UPC(STR$)
```

Wandelt die Buchstaben in der Zeichenkette `STR$` in Großbuchstaben [a-zäöü] → [A-ZÄÖÜ].

```
CALL LWC(STR$)
```

Wandelt die Buchstaben in der Zeichenkette `STR$` in Kleinbuchstaben um [A-ZÄÖÜ]→[a-zäöü].

```
CALL BINSTR(NUM%,STR$)
```

Wandelt die vorzeichenlose Zahl `NUM%` in eine Kette `STR$` von Binärziffern (0, 1).

STR$ muss mit beliebigen 8 oder 16 Zeichen vorbelegt werden, um ein Byte oder eine 16-bit Ganzzahl umzuwandeln.

```
CALL HEXSTR(NUM%,STR$)
```

Wandelt die vorzeichenlose Zahl `NUM%` in eine Kette `STR$` von Hexadezimalziffern (0-9, A-F).

STR$ muss mit beliebigen 2 oder 4 Zeichen vorbelegt werden, um ein Byte oder eine 16-bit Ganzzahl umzuwandeln.

```
CALL REVSTR(STR$)
```

Kehrt die Folge der Zeichen in `STR$` um.

Quellen

[1] C. A. Crayne, D. Girard, „The Serious Assembler“, Baen, 1985.