

BCS 3-Emulator

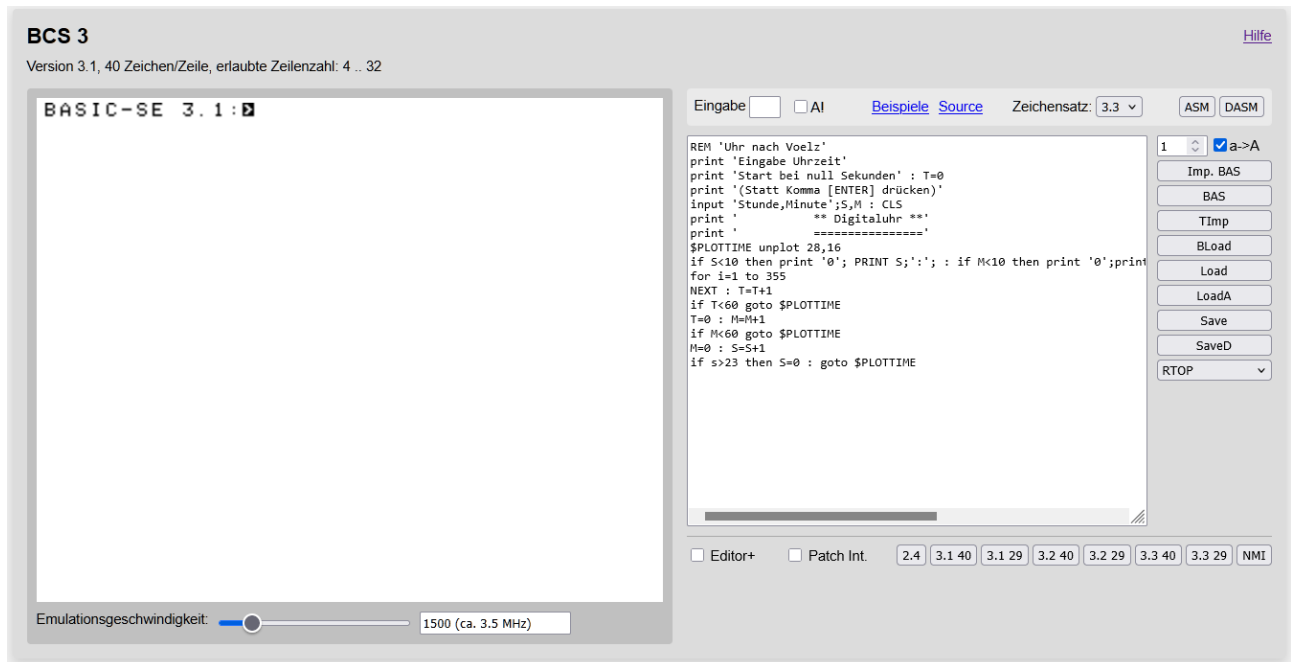
Es wird der Heimcomputer BCS 3 in den Versionen 2.4, 3.1, 3.2 und 3.3 emuliert, für die Versionen 3.x jeweils wahlweise mit 29 oder 40 Zeichen je Zeile und mit maximal 32 Zeilen.

Inhalt

Übersicht zur Bedienung.....	2
Texteingabe.....	4
Erweiterter Editor.....	5
Preprocessing von Quelltexten.....	5
Umwandlung von Zeichen.....	6
Syntaxanpassung.....	6
Automatische Zeilennummerierung.....	6
Symbolische Label.....	7
Makros.....	7
Pixelgrafik.....	9
Grafikbibliothek.....	10
Integrierter Assembler.....	11
Verwendung.....	11
Syntax des Assemblers.....	12
Inline-Assembler.....	13
Integrierter Disassembler.....	14
Technisches.....	15
Patches.....	15
Patches für die 40-Zeichen-Version.....	15
Patches für die Tastatureingabe.....	16
Patches für LOAD und SAVE(D).....	17
Patches für erweiterte Zeilenzahlen.....	18
Patches für Befehlserweiterungen der Interpreter.....	18
Bildschirmsteuerung.....	19
Wichtige Adressen.....	19
Undokumentiertes.....	25
Variablenbezeichner.....	25
Nicht deklarierte Zeichenketten (Versionen 3.x).....	25
Aufbau des Variablenbereichs.....	26
Zeilenaufbau.....	27
Verarbeitung einer Befehlszeile durch den Interpreter.....	27
Zeilennummern (Versionen 3.x).....	29
PEEK und ASC (Versionen 3.x).....	29
Erweitertes USR (Versionen 3.x).....	30
Speicherbereiche und Speichererweiterungen.....	30
Abweichungen vom Original.....	31
Links.....	32

Übersicht zur Bedienung

Beim Aufruf sollte auf einem hinreichend breiten Bildschirm folgende Darstellung erscheinen:



Links befindet sich der virtuelle Bildschirmbereich, daneben der Steuerbereich. Reicht der Platz für die Darstellung nicht aus – etwa auf einem Smartphone – wird der Steuerbereich unterhalb des Bildschirmbereichs dargestellt.

Der virtuelle Bildschirm lässt sich per Mausklick aktivieren. Hier ist für die 3.x-Versionen – wie im Original – zunächst eine Zeilenzahl anzugeben.

Ist der Bildschirm aktiv, erfolgt die Bedienung genau wie im Original, es können also per Tastatur Programme eingegeben und ausgeführt werden. Ein Doppelklick auf den Bildschirmbereich zeigt eine Reihe aktueller Einstellungen an, ein nochmaliger Doppelklick schaltet deren Anzeige wieder aus.

Beim Start ist zunächst die Version 3.1 mit 40 Zeichen/Zeile aktiv. Die anderen Emulationsvarianten können über die entsprechenden Buttons am unteren Rand des Steuerbereichs ausgewählt werden. Die Taste **NMI**, ebenfalls am unteren Rand des Steuerbereichs, bricht ein laufendes BASIC-Programm ab, ohne dessen Variablen zu zerstören (das ist das NMI-Verhalten des Originals).

Die Checkbox **Patch Int.** links neben den Auswahlknöpfen legt fest, ob Patches zur Erweiterung der Interpreter geladen werden sollen. Die Interpretererweiterungen sind initial nicht aktiv. Eine Änderung der Einstellung wird mit der nächsten Laden eines Interpreters wirksam.

Aktuell existieren derartige Erweiterungen nur für die Interpreter der Versionen 2.4 und 3.1.

Unter dem virtuellen Bildschirm befindet sich ein Regler für die Emulationsgeschwindigkeit, wobei die MHz-Angabe nur eine grobe Abschätzung ist. Genau genommen wird einfach die Zahl der Maschinenzyklen des U880 festgelegt, die emuliert werden, bevor die Steuerung kurzzeitig an den Browser zurückgegeben wird. Die reale Geschwindigkeit hängt vom verwendeten Rechner und vom

verwendeten Browser ab. Zudem kommt es vor, dass der Browser nach ein paar Wechseln der Emulationsvariante eine JIT-Kompilierung vornimmt und der Ablauf so wesentlich schneller wird.

Die Felder ganz oben im Steuerbereich folgende Bedeutung:

- **Eingabe** ist ein Textfeld, das ebenfalls genutzt werden kann, um Zeichen in den Bildschirmbereich einzugeben. Es existiert nur deshalb, weil auf Geräten ohne echte Tastatur (z.B. Smartphones) keine Möglichkeit besteht, im Bildschirmbereich (das ist ein HTML-Canvas) Eingabeereignisse zu fangen. Bei jeder Zeicheneingabe in dieses Feld wird eine entsprechende Eingabe in den Bildschirmbereich simuliert.
- **A!** ist eine Checkbox, mit der sich bei der Eingabe die Umwandlung aller Buchstaben in Großbuchstaben erzwingen lässt. Damit wird ein Verhalten ähnlich der Originaltastatur erreicht, was besonders für der Eingabe per Touchscreen nützlich ist.
- **Beispiele** ist ein Download-Link zu den vorhandenen Beispielen (als ZIP-Archiv).
- **Source** ist ein Download-Link zu den Quellen des Emulators (ebenfalls als ZIP-Archiv). Wird das Archiv in ein Verzeichnis entpackt, kann der Emulator von dort aus auch lokal aufgerufen werden (index.html aufrufen).
- **Zeichensatz** erlaubt die Umschaltung des verwendeten Zeichengenerators. Standardmäßig ist für die 3.x-Versionen des Interpreters der von Version 3.3 aktiv. Die Umschaltung kann jederzeit während des laufenden Betriebes erfolgen und wird mit der nächsten Zeichenausgabe sofort wirksam.
- **ASM** ruft den integrierten Assembler auf.
- **DASM** ruft den integrierten Disassembler auf.

Der große Textbereich darunter dient der Eingabe und Bearbeitung von BASIC- und Assembler-Quelltexten außerhalb des eigentlichen Interpreters und somit vor allem der Bequemlichkeit. Ganz links unterhalb des Textbereichs befindet sich eine Checkbox **Editor+**, mit der sich der erweiterte Texteditor ein- und ausschalten lässt (Erläuterung folgt weiter unten)

Die Knöpfe rechts neben dem Textbereich haben folgende Bedeutung:

- **Imp. BAS** importiert den BASIC-Quelltext aus dem Textbereich in den Interpreter. Dabei werden fehlende Zeilennummern automatisch ergänzt, wobei der Wert des Eingabefeldes darüber die Schrittweite der Nummernvergabe festlegt. Die mit „a → A“ beschriftete Checkbox legt fest, ob beim Import Kleinbuchstaben außerhalb von Zeichenkettenliteralen und Schlüsselwörtern automatisch in Großbuchstaben umgewandelt werden sollen.
- **BAS** ist die Umkehrung davon – es wird also ein im Interpreter vorhandenes Programm in Textform umgewandelt und in den Textbereich kopiert.
- **TImp** importiert eine BASIC- oder Assembler-Quelldatei in den Textbereich. Der Quelltext sollte im Zeichensatz ISO-8859-1 angegeben sein.
- **BLoad** importiert einen BASIC-Quelltext aus einer Datei (Zeichensatz ISO-8859-1) direkt in den Interpreter. Auch hier werden fehlende Zeilennummern automatisch ergänzt.

- **Load** lädt ein BASIC-Programm im Binärformat aus einer Datei in den Interpreter. Die Funktion kann auch aus dem Interpreter über das BASIC-Kommando LOAD aufgerufen werden.
- **LoadA** lädt den Inhalt einer Binärdatei (Daten oder Maschinenprogramm) an eine frei wählbare Adresse. Dazu erscheint zunächst ein Dialog zur Eingabe der Adresse...

Maschinencode oder Binärdaten laden

Adresse (hex):

... gefolgt von einem Dialog zur Auswahl der zu ladenden Datei.

Ein ggf. geladenes Programm wird dabei nicht gelöscht.

Vorsicht: Es gibt keinerlei Prüfung der angegebenen Adresse, es ist also durchaus möglich, beliebige Programm- und Datenbereiche zu überschreiben.

- **Save** speichert ein BASIC-Programm im Binärformat in eine Datei, deren Name vorher über einen Dialog abgefragt wird. Sofern der Browser das Schreiben von Dateien nicht direkt unterstützt (z.B. Firefox), entspricht dies einem Download. Die Funktion kann auch über das BASIC-Kommando SAVE aus dem Interpreter aufgerufen werden.
- **SaveD** verhält sich wie **Save**, gibt jedoch nicht nur das Programm, sondern auch den gesamten Variablenbereich mit aus. Die Funktion kann auch über das BASIC-Kommando SAVED aus dem Interpreter aufgerufen werden. Unter SaveD befindet sich noch eine Combobox zur Auswahl des beim Speichern verwendeten Endes. Hier gibt es drei Optionen (auch für die Ausgabe aus dem Interpreter):
 - **RTOP:** Ausgabe bis zum vom System gemeldeten Speicherende. Dies ist die Voreinstellung und entspricht dem Standardverhalten des Originals.
 - **Variable:** Die Ausgabe endet mit dem Ende des tatsächlich verwendeten Variablenbereichs. Das ist platzsparend, ignoriert aber zusätzlich in den freien Speicherplatz Geschriebenes (nur Versionen 3.x).
 - **Alles:** Es wird alles bis zum Ende des verfügbaren Speichers (also auch Bereiche hinter dem gemeldeten Ende, etwa der Inhalt des Grafikspeichers im Emulator) gespeichert.

Texteingabe

Da sich eine normale PC-Tastatur erheblich von der des BCS 3 unterscheidet, ergeben sich Auswirkungen auf die Art der Texteingabe:

- Alle Zeichen des BCS 3-Zeichensatzes, die eine Entsprechung auf der Tastatur haben, können direkt eingegeben werden – mit einer Ausnahme: Für den Backslash (\) geht das nicht, da der entsprechende Zeichencode beim BCS 3 einem Backspace entspricht.

- Bei der direkten Eingabe in den Bildschirmbereich wird die Groß- und Kleinschreibung umgekehrt. Ohne Shift-Taste werden also Großbuchstaben, mit Shift-Taste Kleinbuchstaben erzeugt. Dies kommt dem Verhalten der Originaltastatur des BCS 3 nahe, die keine Eingabe von Kleinbuchstaben erlaubt.

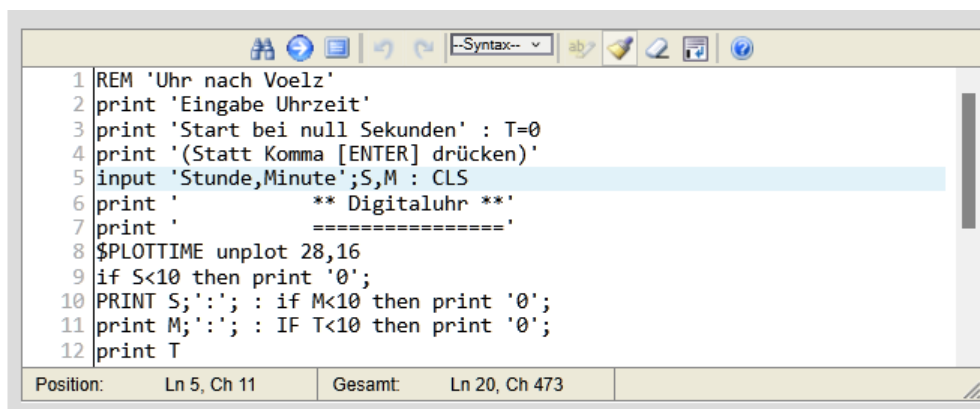
Weil es die Umlaute ä, ö und ü im BCS 3 nur als Kleinbuchstaben gibt, werden sie immer als Kleinbuchstaben übernommen.

Wird die Strg-Taste in Verbindung mit einer gültigen Zeicheneingabe verwendet, wird der Zeichencode um 32 verringert, so dass, ggf. in Verbindung mit der Shift-Taste, die Grafikzeichen mit den Codes 1..31 eingegeben werden können. Strg+Shift+ß liefert im Zeichensatz 3.2/3.3 außerdem das Ω-Symbol.

- Bei der Eingabe in das Textbearbeitungsfeld gibt es zunächst keine Konvertierungen, dies geschieht im Rahmen des Preprocessings beim späteren Import in den Interpreter.

Erweiterter Editor

Zur bequemerem Bearbeitung größerer Quelltexte gibt es einen erweiterten Texteditor, der sich über die Checkbox **Editor+** ein- und ausschalten lässt. Er bietet eine Reihe von Zusatzfunktionen, wie Suchen, Ersetzen, Redo/Undo und Syntaxhervorhebung für BCS-BASIC und U880/Z80-Assembler. Beim Einschalten ersetzt der Editor das Texteingabefeld:



In der Combobox oben lässt sich die Syntaxhervorhebung für Basic oder Assembler auswählen. Außerdem gibt es einen Knopf zur Umschaltung des Editors in den Vollbildmodus.

Hinweis: Beim Aufruf von Funktionen, die Text aus dem Eingabefeld lesen oder in das Eingabefeld schreiben (ASM, DASM, Imp. BAS, BAS, Timp), wird der erweiterte Editor automatisch geschlossen.

Preprocessing von Quelltexten

Beim Import von Basic-Quelltexten in den Interpreter – aus dem Textbearbeitungsfeld oder beim Laden mit **BLoad** – werden automatische Konvertierungen vorgenommen, um eine für den BCS 3 gültige Quelle zu erzeugen und das Schreiben gut lesbarer Quelltexte (hoffentlich) zu vereinfachen.

Umwandlung von Zeichen

- Alle Buchstaben, die nicht Teil von Zeichenkettenliteralen sind, werden generell in Großbuchstaben umgewandelt, wenn sie Bestandteil von Schlüsselwörtern sind, sonst (für Buchstaben in Bezeichnern) sofern dies nicht über die Checkbox „a → A“ deaktiviert wird.
- Zeichen in Zeichenkettenliteralen bleiben unverändert – mit Ausnahme von ä, ö und ü, die ggf. in Kleinbuchstaben konvertiert werden.
Außerdem können in Zeichenkettenliteralen auch Zeichencodes in der Form `\xhh` angegeben werden, wobei `hh` für zwei Hexadezimalziffern steht. Dies erlaubt z.B. die Angabe von Zeichen im Bereich von 1..1FH in Literalen.
- Ungültige Zeichen (alle mit einem Zeichencode > 127, für die es keine spezielle Konvertierung gibt) werden in Leerzeichen umgewandelt.
- Für einige Sonderzeichen wird eine Konvertierung in den BCS 3-Zeichensatz vorgenommen.

Syntaxanpassung

Importierte Quelltexte werden, sofern das möglich ist, an die syntaktischen Besonderheiten des BCS 3 angepasst, um die Portierung ‚fremder‘ Quellen zu vereinfachen:

- Zeilen, die mit einem Apostroph (‘) beginnen, werden als reine Kommentare aufgefasst und übersprungen. Es wird hier keine REM-Anweisung erzeugt.
- Zeichenkettenliterate können auch in doppelte Anführungszeichen eingeschlossen sein. Es wird dann eine entsprechende Umwandlung vorgenommen.
- Der in anderen BASIC-Dialekten übliche Ungleich-Operator `<>` wird in `#` umgewandelt.
Vorsicht: Bei Version 2.4 hat `#` nicht die Bedeutung des Ungleich-Operators.

Automatische Zeilennummerierung

Beim Import von BASIC-Quelltexten in den Interpreter werden fehlende Zeilennummern automatisch ergänzt, wobei die im Eingabefeld über dem Button **Imp. BAS** eingestellte Schrittweite verwendet wird. Ist dort Null eingestellt, wird die automatische Numerierung nicht durchgeführt.

Enthält eine Quelltextzeile bereits eine Zeilennummer, wird die automatische Numerierung danach fortgesetzt.

Mit der Angabe von `,+‘`, gefolgt von einer Zahl, kann einmalig eine andere Schrittweite festgelegt werden. Dies erlaubt die Festlegung eines fixen Abstands zwischen zwei Zeilen, was für Adressrechnungen für die Ziele von GOTO- oder GOSUB-Anweisungen nützlich sein kann.

Mit Angabe von `,*‘`, gefolgt von einer Zahl, kann die nächste Zeilennummer als Vielfaches der angegebenen Zahl festgelegt werden. Dies ist nützlich, um spätere Restarts des Programms per GOTO auf eine gut merkbare Zeilennummer festzulegen.

Symbolische Label

Wenn man die automatische Vergabe von Zeilennummern durchgehend verwenden will, besteht die Schwierigkeit, Sprungziele geeignet zu formulieren. Um damit bequem umgehen zu können, erlaubt der BCS 3-Emulator symbolische Label in BASIC-Quelltexten.

Ein Label beginnt mit dem Dollarzeichen (\$) bzw. im Zeichensatz 3.1 mit ₭), gefolgt von einem Bezeichner, bestehend aus Buchstaben und dem Punkt (.). Die Groß- bzw. Kleinschreibung ist dabei unerheblich. Label müssen innerhalb eines Quelltextes eindeutig sein. Label können direkt am Zeilenanfang (führende Leerzeichen werden ignoriert) oder nach einer Zeilennummer (einschließlich der Pseudonummern mit + und *) definiert und als Verweis überall dort, wo eine Zeilennummer als Argument von GOTO oder GOSUB stehen kann, verwendet werden. Dies schließt auch Ausdrücke zur Berechnung des Sprungziels ein. Außerdem können sie innerhalb von Zeichenketten stehen, wo sie ebenfalls als Zeilennummern aufgelöst werden.

Statt eines definierten Labels kann als Referenz auch die Zeichenfolge \$\$ verwendet werden, was einem Verweis auf die aktuelle Programmzeile entspricht.

Hier als Beispiel ein Auszug aus einem Programm zur Berechnung von Netztrafos (trafo_2.bas in der Beispielsammlung):

```
$INC.KERN J=J+1
...
PRINT 'Wickelquerschnitt: ';AW;' qmm'
PRINT 'Fensterquerschnitt: ';T(J,6);' qmm'
PRINT 'Für grösseren Kern Start'
PRINT 'mit GOTO $KERNPLUS'
PRINT 'Viel Spass beim Wickeln!'
END
*100 $KERNPLUS CLS :Y= USR(36H)
GOTO $INC.KERN
```

Das Beispiel definiert und verwendet die Label \$INC.KERN und \$KERNPLUS.

Die Angabe von *100 vor der Definition von \$KERNPLUS sorgt dafür, dass die betreffende Zeile eine durch 100 teilbare Nummer erhält. Die berechnete Zeilennummer wird an den Benutzer als Hinweis für den Restart ausgegeben.

Makros

Es ist möglich, Makros, ähnlich denen des C-Präprozessors, zu definieren, was u. a. helfen soll, besser merkbare Bezeichner, z.B. als Alias für Variablen, zu definieren oder Unterprogramme über einen lesbaren Namen aufrufbar zu machen.

Makros können einzeilig oder mehrzeilig sein. Ihre Definition entspricht der in C und hat die Form

```
#define macroname[(argumentlist)] body
```

Dabei bezeichnet *macroname* den Namen des Makros. Die optionale Argumentliste ist eine kommaseparierte Liste beliebiger Bezeichner, die als formale Parameter des Makros dienen; *body* schließlich ist die eigentliche Implementierung, also der Text, der für *macroname*, ggf. unter Ersetzung der Argumente, eingesetzt werden soll.

Steht *body* in derselben Zeile wie `#define`, handelt es sich um ein einzeliges Makro, andernfalls um ein mehrzeiliges. Mehrzeilige Makros müssen mit `#endif` abgeschlossen werden, das in einer eigenen Zeile notiert werden muss. Ein paar Beispiele:

```
#define deek(addr) (peek(addr+1)*256+peek(addr))
#define doke(addr,val) poke(addr,(val) AND 0FFH,int((val) / 256)
#define CursorBack
    doke(3c08h,deek(3c08h)-1)
#endif
#define enableGr(enable) OUT 2,enable
```

Variadische Makros

Wird bei der Makrodefinition als letztes (oder einziges) Element von *argumentlist* die Ellipse (`. . .`) angegeben, so lässt sich ein Makro mit variabler Argumentzahl erzeugen. Im Implementierungsteil lässt sich dann über `NARGS` auf die Anzahl der übergebenen Argumente und mit `ARG(i)` auf ein einzelnes Argument zugreifen, wobei *i* von 0 bis `NARGS-1` läuft.

Eine Iteration über die Argumente ist mit

```
#loop(idxname, startidx, endidx, step)
    loopimpl
#endloop
```

möglich. Dabei ist *idxname* der Bezeichner für den Index zur Verwendung mit `ARG`, *startidx* der Startwert für den Index, *endidx* der Endwert und *step* die Schrittweite. Ist *step* negativ, wird die Schleife rückwärts durchlaufen – *startidx* sollte dann größer als *endidx* sein.

Auch hierfür ein Beispiel:

```
#define drawLine(x1,y2,y1,y2,mode)
    GA(1)=3:GA(2)=x1:GA(3)=y1:GA(4)=x2:GA(5)=y2:GA(6)=mode:O= USR(1800H)
#endif
#define drawPolyline(mode,...)
    #loop(i,1,NARGS-4,2)
        drawLine(ARG(i),ARG(i+1),ARG(i+2),ARG(i+3),mode)
    #endloop
#endif
```

Tokenverkettung

Innerhalb des Implementierungsrumpfes (*body*) eines Makros kann das Symbol `##` zur Verkettung aufeinander folgender Symbole verwendet werden. Dies ist z. B. nützlich, um Bezeichner aus übergebenen Argumenten eines Makros zu konstruieren. Beispiel:

```
#define SETDEF(NAME,CAPACITY)
dim NAME##D(CAPACITY)
dim NAME##V(5)
#endif
```

Achtung: Die Behandlung ist hier recht primitiv: Die `##`-Symbole werden an Ende der Expansion eines Makros einfach entfernt, ohne dass dabei ein Kontext beachtet würde. Dies geschieht also

auch, wenn ## innerhalb eines Zeichenkettenliterals steht, weshalb man dies innerhalb von Makrodefinitionen vermeiden sollte.

Hinweise:

- Bei Namen von Makros und deren formalen Parametern sowie bei den zur Definition verwendeten Schlüsselwörtern ist die Groß- bzw. Kleinschreibung signifikant.
- Die Verarbeitung von Makros stellt den Beginn des Preprocessings dar. Dabei werden zunächst alle Makrodefinitionen des Quelltextes erfasst. In einem zweiten Durchlauf erfolgt dann die Ersetzung. Daraus ergibt sich, dass die Position der Makrodefinitionen beliebig ist, d. h., sie können durchaus auch am Ende des Quelltextes stehen.
- Für die Unterscheidung von Makros sind nur die Namen von Bedeutung, nicht etwaige formale Parameter. Ein ‚Überladen‘ ist also nicht möglich. Bei mehrfacher Definition von Makros unter demselben Namen ‚gewinnt‘ stets das zuletzt definierte.
- Formale Parameter eines Makros, für die bei der Expansion kein konkreter Wert angegeben wird, werden im Ergebnis durch eine leere Zeichenkette ersetzt; eine Fehlermeldung oder Warnung gibt es hier nicht.
- Die Verwendung von Makros kann zwar zu leichter lesbarem, mitunter aber auch zu weniger effizientem und umfangreicherem Code führen. Es ist also stets zu bedenken, ob ihr Einsatz sinnvoll ist.

Beispiele zur Verwendung von Makros sind im Unterordner ‚macro‘ des Beispielsarchivs zu finden.

Pixelgrafik

Der Emulator beinhaltet als Erweiterung eine s/w-Pixelgrafik, deren Auflösung von der Bildschirmeinstellung abhängt und 8 x Zeichenzahl pro Zeile in horizontaler bzw. 8 x Zeilenzahl in vertikaler Richtung beträgt. Die maximale Auflösung (bei 32 Zeilen zu 40 Zeichen) beträgt also 320 x 256 Pixel.

Die Pixelgrafik benutzt den Adressbereich ab BC00H als Variablen- und ab C000H als Grafikspeicher.

Die Nutzung, d. h. Anzeige der Grafik, wird über den I/O-Port 2 gesteuert und kann mit der BASIC-Anweisung `OUT 2,1` ein- bzw. mit `OUT 2,0` ausgeschaltet werden.

Die Grafik kann direkt über POKE-Befehle angesteuert werden – siehe hierzu das Programm `graphics_n1.bas` aus der Beispielsammlung.

Grafikbibliothek

Im Adressbereich ab 1800H gibt es außerdem eine Grafikbibliothek, die wichtige Grafikfunktionen als Maschinencode implementiert und so eine schnelle Grafikausgabe erlaubt.

Sie wird von BASIC-Seite über zwei Unterprogramme angesteuert:

- `USR(1804H)` muss einmalig zur Initialisierung aufgerufen werden.
- `USR(1800H)` dient dem Aufruf der eigentlichen Grafikfunktionen.

Für die Verwendung der Bibliothek wird vorausgesetzt, dass ein Zahlenfeld mit dem Namen `GA` und mindestens 6 Elementen definiert ist, das zur Übergabe von Parametern verwendet wird. In einem BASIC-Programm, das die Bibliothek benutzen will, muss also vor dem Aufruf von `USR(1804H)` eine Definition `DIM GA(6)` stehen. Ist dies nicht der Fall, liefert `USR(1804H)` als Ergebnis -1, sonst die Adresse des ersten Elements von `GA`.

Nachfolgende Tabelle beschreibt die einzelnen Funktionen und die dafür im Feld `GA` vor dem Aufruf von `USR(1800H)` anzugebenden Parameter.

Funktion	Parameter GA(i)						Beschreibung
	1	2	3	4	5	6	
CLR	0						Grafik löschen
SETPX	1	x	y	m			Setzt ($m \neq 0$) oder löscht ($m=0$) den Bildpunkt an Position x, y.
GETPX	2	x	y				Gibt die Bitmaske für den Bildpunkt an Position x, y zurück; Null bedeutet, dass der Punkt nicht gesetzt ist.
LINE	3	x1	y1	x2	y2	m	Zeichnet ($m \neq 0$) oder löscht ($m=0$) eine Linie von (x1,y1) nach (x2,y2).
RECT	4	x1	y1	x2	y2	m	Zeichnet ($m \neq 0$) oder löscht ($m=0$) ein Rechteck von (x1,y1) nach (x2,y2).
RECTF	5	x1	y1	x2	y2	m	Zeichnet ($m \neq 0$) oder löscht ($m=0$) ein gefülltes Rechteck von (x1,y1) nach (x2,y2).
ELLIPSE	6	cx	cy	rx	ry	m	Zeichnet ($m \neq 0$) oder löscht ($m=0$) eine Ellipse um (cx, cy) mit den Radien rx und ry.

Im Programm in `graphics_2.bas` aus der Beispielsammlung werden alle hier beschriebenen Funktionen aufgerufen. Die Werte nicht benötigter Parameter in `GA` werden von den jeweiligen Funktionen ignoriert.

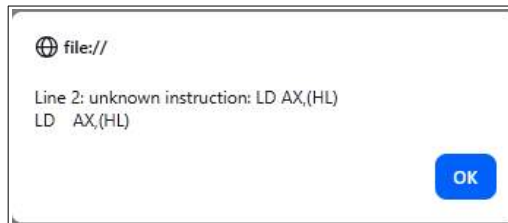
Hinweis: Die Grafikbibliothek ist für Version 2.4 nicht nutzbar, da es dort keine Möglichkeit des Aufrufs von Maschinenprogrammen und keine Arrays gibt.

Integrierter Assembler

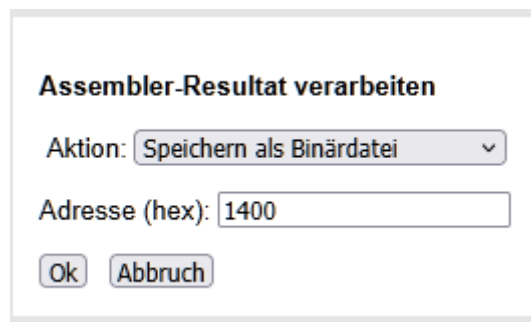
Der Emulator enthält einen integrierten Z80-Assembler auf Basis der JavaScript-Implementierung von [UO Soft](#), ergänzt um einige kleine Verbesserungen und Bugfixes sowie die Möglichkeit, mehrere Anweisungen auf einer Zeile zu notieren.

Verwendung

Der Assembler wird über die Taste **ASM** gestartet und übersetzt einen im Textbereich befindlichen Assembler-Quelltext. Tritt dabei ein Fehler auf, erscheint eine MessageBox mit der Beschreibung des ersten gefundenen Fehlers:



Bei Erfolg erscheint ein Dialog zur weiteren Verarbeitung des Ergebnisses:



Hier ist zunächst eine gewünschte Aktion auszuwählen. Möglichkeiten sind:

- **Speichern als Binärdatei:** Das Ergebnis wird als Binärdatei gespeichert.
- **DATA-Anweisungen in Textfeld:** Das Ergebnis wird als Folge von BASIC-DATA-Anweisungen in den Textbereich geschrieben.
- **POKE-Anweisungen in Textfeld:** Das Ergebnis wird als Folge von BASIC-POKE-Anweisungen in den Textbereich geschrieben.
- **JS-Array in Textfeld:** Das Ergebnis wird als JavaScript-Array-Literal in den Textbereich ausgegeben; ersetzt man die eckigen Klammern durch geschwungene, ist es auch ein gültiges C-Array. Die Ausgabe ist vor allem für Ergänzungen des Emulators selbst gedacht.
- **In den Speicher schreiben:** Das Ergebnis wird direkt in den Arbeitsspeicher geschrieben und steht somit sofort zum Test zur Verfügung.

Die Adressangabe ist für das Erzeugen von POKE-Anweisungen sowie für das direkte Schreiben in den Speicher von Bedeutung und legt hierfür die Anfangsadresse fest. Initial steht in diesem Feld der Wert der ersten ORG-Anweisung des Assembler-Quelltextes oder Null, falls keine ORG-Anweisung angegeben wurde.

Hinweise:

- Die Ausgabe von POKE- bzw. DATA-Anweisungen oder JS-Arrays überschreibt den Inhalt des Texteingabefeldes. Ein darin enthaltener Assembler-Quelltext sollte also vorher gesichert werden.
- Wie beim direkten Laden von Binärdaten aus einer Datei, gibt es auch beim Schreiben der Assembler-Ausgabe in den Speicher keinerlei automatische Vorsichtsmaßnahmen und keine Änderungen an einem ggf. geladenen BASIC-Programm.

Syntax des Assemblers

Der Assembler versteht alle dokumentierten U880/Z80-Befehle, die undokumentierten Schiebebefehle SLL (CB 30H..37H) sowie die Direktiven `ORG`, `EQU`, `DB`, `DW`, `DS`, `DEFM`, `DEFB` und `END`. Eine Syntaxbeschreibung mit einigen Beispielen ist [hier](#) zu finden.

Eine syntaktische Erweiterung bietet die Möglichkeit, mehrere Assembler-Anweisungen in einer Quelltextzeile zu notieren. Hierfür werden die Anweisungen mit `:` getrennt.

Darüber hinaus ist es möglich, an Stelle von numerischen Konstanten beliebige Ausdrücke zu verwenden (genau genommen alle, die in JavaScript zulässig sind). Vorsicht ist hier bei Klammern geboten, da sie u. U. mit der Syntax echter Assembler-Anweisungen kollidieren können.

So würde etwa `LD HL,(2+3)` ein `LD HL,(nn)` statt dem vielleicht gewünschten `LD HL,nn` erzeugen.

Der Assembler kennt keine ‚Tricks‘, wie das automatische Ersetzen von zu weiten Relativsprüngen oder absolute Zieladressen bei Relativsprüngen (abgesehen von Labels) und eigentlich auch keine Makros. Der Quelltext wird aber mit dem Makro-Präprozessor vorverarbeitet, der auch für BASIC-Quelltexte verwendet wird, so dass solche Makros auch in Assembler-Quelltexten verwendet werden können.

Dabei empfiehlt es sich, Makrodefinitionen am Ende des Assembler-Quelltextes zu notieren, weil sich sonst die Zeilennummern der Fehlerbeschreibungen verschieben. Solche Verschiebungen entstehen auch, wenn ein Makro mehrere Quelltextzeilen erzeugt. Sofern möglich, bietet es sich hier an, die Verkettung mehrerer Anweisungen auf einer Zeile mit `:` zu nutzen.

Wie beim BASIC-Preprocessing, werden auch hier Zeilen ignoriert, die mit einem Apostroph beginnen. Dies erlaubt das Auskommentieren von Aufrufen mehrzeiliger Makros.

Hinweis zu Zeichenkettenliteralen:

Der Assembler akzeptiert auch Zeichenkettenliterals, die in Hochkommata geklammert sind. So wäre etwas wie

```
DB 'Hallo ASM',1eh
```

für ihn durchaus akzeptabel. In solchen Literalen dürfen dann auch Anführungszeichen als Elemente stehen.

Zu beachten ist, dass der Assembler nichts von besonderen Zeichenkodierungen im Zeichensatz des BCS 3, z. B. für Umlaute, weiß und demzufolge auch keine automatischen Umwandlungen vornehmen kann.

Inline-Assembler

Der Assembler kann auch ‚inline‘ in BASIC-Quelltexten verwendet werden. Ein eingebetteter Assembler-Block beginnt mit

```
#asm[mode, addr]
```

und endet mit

```
#endasm.
```

Die Parameter *mode* und *addr* von haben dabei folgende Bedeutung:

- **mode** bestimmt die Art und Weise, wie der in den endgültigen den BASIC-Quelltext einzubettende Code erzeugt werden soll. Dabei sind folgende Werte möglich:
 - **poke**: erzeugt eine Folge von POKE-Anweisungen
 - **data**: erzeugt DATA-Anweisungen mit einer anschließenden FOR-NEXT-Schleife, die die als Hilfsvariablen einfache und doppelte Pipe-Symbole (| als Laufvariable und || als Datenvariable) verwendet.
 - **Variablenangabe**: Statt *data* können auch zwei Variablennamen in der Form *v1*; *v2* angegeben werden. Hier werden ebenfalls DATA-Anweisungen erzeugt, jedoch werden *v1* und *v2* als Lauf- bzw. Datenvariable in der FOR-NEXT-Schleife verwendet. Ist nur ein Variablenname angegeben, wird der zweite aus dem angegebenen erzeugt: Besteht der Name aus zwei Zeichen, ist der andere das erste Zeichen des angegebenen, andernfalls wird der zweite Name als Doppelung des ersten erzeugt.
- **addr** legt die Adresse fest, ab der der vom Assembler erzeugte Maschinencode abgelegt werden soll. Hier gibt es folgende Möglichkeiten:
 - **auto**: Die Ablageadresse entspricht dem Wert der ersten *org*-Anweisung im Assembler-Code (0, falls dort nichts angegeben ist).
 - **Absolute Adresse**: Die Ablageadresse wird absolut festgelegt; die Adressangabe kann dabei dezimal oder hexadezimal (nach BASIC-Syntax) erfolgen.
 - **Variablenname**: Die Ablage im Speicher erfolgt beginnend mit dem Wert der angegebenen Variablen. Dies ermöglicht es, die Position des Maschinenprogramms im Speicher zur Laufzeit festzulegen. Für die POKE-Anweisungen werden dann Adressangaben in der Form *VN+n* erzeugt (*VN* ist der Variablenname, *n* ein Offset). Achtung: Dies ist nur für Adressen bis 7FFFH möglich, sonst beschwert sich der BASIC-Interpreter über einen Integer-Überlauf.

Innerhalb des eingebetteten Assembler-Blocks gelten alle Regeln, die auch für den separaten Assembler gültig sind. Mehrere eingebettete Assembler-Blöcke haben keinerlei Bezug zueinander, kennen also definierte Label oder Daten des anderen Blocks nicht.

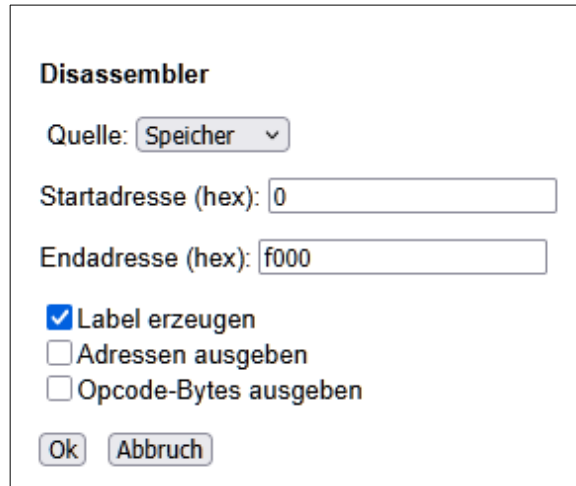
Hinweis: Bei der Ausgabe von DATA-Anweisungen wird kein RESTORE-Aufruf generiert, da in einem Programm u.U. noch an anderen Stellen READ-Anweisungen vorkommen können. Der

Aufruf von RESTORE muss also an passender Stelle per Hand in das Programm eingetragen werden.

Integrierter Disassembler

Der Vollständigkeit halber enthält der Emulator auch einen integrierten Disassembler. Dabei handelt es sich um eine erweiterte Version des in den Beispielen enthaltenen (und in BASIC geschriebenen) Disassemblers, die ursprünglich in [B++](#) implementiert und (mit Gemini-Unterstützung) nach JavaScript portiert wurde.

Der Disassembler wird über die Taste **DASM** gestartet, wobei zunächst folgender Dialog erscheint:



Hier können folgende Einstellungen vorgenommen werden:

- **Quelle:** Hier wird die Quelle für den Maschinencode festgelegt, der vom Disassembler verarbeitet werden soll: ‚Speicher‘ legt den virtuellen Arbeitsspeicher als Quelle fest, ‚Binärdatei‘ eine zu ladende Datei.
- **Startadresse:** Adresse, an der die Verarbeitung beginnt. Ist die Quelle der Arbeitsspeicher, so bestimmt der Wert die tatsächliche Adresse, bei einer Datei handelt es sich um die dem Dateianfang zugeordnete Adresse.
- **Endadresse:** Adresse des letzten Bytes im zu verarbeitenden Bereich. Ist die Quelle der Arbeitsspeicher, so bestimmt der Wert die tatsächliche Ende-Adresse, bei einer Datei ist es das ‚virtuelle‘ Ende.
- **Label erzeugen:** Die Option legt fest, ob der Disassembler Label für gefundene Sprungziele erzeugen und Referenzen in Operanden durch die entsprechenden Label ersetzen soll.
- **Adressen ausgeben:** Die Option legt fest, ob vor jedem Befehl die Adresse ausgegeben werden soll.
- **Opcode-Bytes ausgeben:** Die Option legt fest, ob vor jedem Befehl die zugehörigen Opcode-Bytes ausgegeben werden sollen.

Mit **Ok** wird der Disassembler aufgerufen; falls als Quelle ‚Binärdatei‘ ausgewählt wurde, erscheint zunächst noch ein Dialog zur Auswahl der Datei.

Das Ergebnis des Disassemblers wird in den Texteingabebereich geschrieben und ersetzt dort den vorhandenen Text.

Technisches

Kern des Emulators bildet der in JavaScript geschriebene [Z80-Emulator von Matt Howell](#).

Außerdem werden die Inhalte der [Original-ROMs](#) der Versionen 2.4, 3.1, 3.2 und 3.3 sowie der zugehörigen Zeichengeneratoren verwendet – konvertiert in JavaScript-Arrays.

Der Emulator verwendet einen vollständigen (virtuellen) Speicherausbau von 64KB ohne Doppelbelegung von Adressbereichen. Letzteres ergibt zusätzlich nutzbaren Speicher im Bereich von 1800H bis 37FFH sowie oberhalb von BC00H (das gemeldete Speicherende ist auf BBFFH festgelegt, da der Interpreter nur freien Speicher bis 32KB verwalten kann).

Der Emulator benutzt diese Bereiche für eine Pixelgrafik-Erweiterung (siehe oben).

Patches

Weil der Emulator naturgemäß nicht über die Original-Hardware, wie CTC, physisches Keyboard oder Tonbandansteuerung verfügt, sowie zur Anpassung an die Varianten mit 40 Zeilen je Zeile, sind einige Patches der ROM-Inhalte notwendig.

Patches für die 40-Zeichen-Version

Für die Darstellung von 40 Zeichen je Zeile müssen die ROM-Inhalte angepasst werden, wobei es hier Unterschiede zwischen den Versionen des BCS 3 gibt. Nachfolgend sind die Patches für die einzelnen Versionen dargestellt.

Version 3.1

Adresse (Hex)	Byte(s) (Hex)
0020	0E
0038	DD 23 DD 2B 10 EE 47 0E 0 E1 E9
0050	D5
0051	D1
0061	28
008A	29
00A1	A9
00A9	28
0169	29 09 29 29 29 09

Version 3.2

Adresse (Hex)	Byte(s) (Hex)
0020	0E
0038	DD 23 DD 2B 10 EE 47 0E 0 E1 E9

0050	D5
0051	D1
0061	28
008C	29
00A2	C9
00AA	28
019e	29 09 29 29 29 09

Version 3.3

Adresse (Hex)	Byte(s) (Hex)
0020	0E
0038	DD 23 DD 2B 10 EE 47 0E 0 E1 E9
0050	D5
0051	D1
0061	28
008C	29
00A2	DD
00A9	28
019e	29 09 29 29 29 09

Patches für die Tastatureingabe

Für die Tastaturabfrage wird das Original komplett umgangen.

Tritt ein Tastaturereignis ein (Taste gedrückt), wird geprüft, ob es sich um ein für den BCS 3 gültiges Zeichen handelt. Trifft dies zu, wird ggf. der Zeichenkode abgewandelt und aus einer Übersetzungstabelle der Rückgabewert der Original-Tastaturabfrage USR(0E0H) sowie der Tastaturcode im Speicher an den Adressen 13FC/DH und 13FEH abgelegt.

Beim Loslassen der Taste werden die Werte wieder gelöscht.

Für die 3.x-Versionen wird die Tastaturabfrage (UP 0E0H) wie folgt ersetzt:

```
org 00e0h
ld a, (13feh)
ld hl (13fch)
and a,a
ret
```

Für Version 2.4 ist nur der Tastaturcode von Bedeutung. Die Zeicheneingabe (UP 0D2H) wird wie folgt ersetzt:

```
org 00d2h
PUSH HL
PUSH BC
```

```

wait:
    ld a, (13feh)
    and a
    jr z, wait ; Warten auf Zeichencode != 0
    cp 7fh
    jr nz, L2
    ld a, 5dh ; 7FH durch 5DH ersetzen
L2:
    cp 5ch ; Backspace nicht am Bildschirm ausgeben
    jr nz, chout
    push hl
    ld hl, 03c08h ; dafür Cursor eins vor wie bei Ausgabe
    inc (hl)
    pop hl
    jr end
chout:
    push af
    call 088h ; Zeichenausgabe
    pop af
end:
    push af
    xor a
    ld(13feh), a ; Zeichenpuffer leeren
    pop af
    POP BC
    POP HL
    ret

```

Patches für LOAD und SAVE(D)

Da der Emulator kein Kassetten-Interface hat, werden die Befehle LOAD und SAVE bzw. SAVED so verändert, dass sie lediglich einen Wert (0 für LOAD, 1 für SAVE und 2 für SAVED) auf den (virtuellen) Port 1 schreiben. Das Schreiben auf diesen Port wird über einen Callback abgefangen, der die Load- und Save-Funktionen der Emulatorumgebung aufruft – wie die Betätigung der entsprechenden Buttons in der Bedienoberfläche.

Der Patch für LOAD sieht für die 3.x-Versionen so aus:

```

    ld a, 0
    out (1), a
    rst0

```

bzw. als Bytefolge:

```
3E D3 01 C7
```

Er wird für Version 3.1 ab Adresse 0E6C, für Version 3.2 ab 05EC und für Version 3.3 ab 05EF eingetragen.

Der Patch für SAVE/SAVED ist etwas umfangreicher, weil er die SAVE und SAVED unterscheiden muss:

```

    ld a, (iy+00h)
    cp a, 'D' ; fe 44
    ld a, 1
    jr nz L1
    ld a, 2
L1:
    out (1), a

```

```
ret
```

bzw. als Bytefolge:

```
FD 7E 00 FE 44 3E 01 20 02 3E 02 D3 01 C9
```

Für Version 3.1 erfolgt der Eintrag ab Adresse 0D8F, für Version 3.2 ab 0D8B und für 3.3 ab 0D8E.

Version 2.4 sieht die Einsprungsadressen 0800 für LOAD bzw. 0803 für SAVE vor, die im Original aber keine Standardimplementierung haben.

Der Emulator bildet beide Befehle auf OUT-Aufrufe (siehe oben) wie folgt ab:

```
org 0800h
jr LOAD
org 0803h
;schreibt 1 oder 2 auf Port 1
ld a, (DE)
cp 'D'
ld a, 1
jr nz, L1
ld a, 2
L1:
out (1), a
jp 0193h ; Sprung zur Befehlseingabe
; schreibt 0 auf Port 1
LOAD:
ld a, 0
out (1), a
jp 0190h; Sprung zur Initialisierung
```

Patches für erweiterte Zeilenzahlen

Um die Verwendung von Zeilennummern bis 32512 zu ermöglichen (siehe unten), sind folgende Patches erforderlich:

Version	Adresse (Hex)	Byte(s) (Hex)
3.1	01B5	00 7F
	038F	7F
3.2	0195	00 7F
	03AB	7F
3.3	0195	00 7F
	03A7	7F

Diese Patches sind auch für die Original-ROMs und somit für den ‚echten‘ BCS 3 anwendbar.

Patches für Befehlserweiterungen der Interpreter

Unter „[Undokumentiertes](#)“ sind Möglichkeiten zur Erweiterung des Befehlssatzes des Interpreters beschrieben, wobei der Emulator Beispielimplementierungen für die Versionen 2.4 und 3.1 enthält. Diese Erweiterungen sind standardmäßig deaktiviert, können aber vom Benutzer eingeschaltet werden. Implementierungsdetails sind der Quelldatei des Emulators (index.html) und dort den versionsspezifischen ‚settings‘-Objekten (etwa ab Zeile 50) zu entnehmen.

Bildschirmsteuerung

Eine echte Bildschirmsteuerung gibt es im Emulator nicht. Es existiert kein CTC-Baustein und folglich werden auch keine Interrupts erzeugt, so dass keine Patches an den ROM-Inhalten erforderlich sind.

Die Ansteuerung des virtuellen Bildschirms erfolgt einfach dadurch, dass beim Schreibzugriff auf eine Adresse im Bildspeicherbereich (Text oder Grafik) im Emulator ein Flag gesetzt wird, dass ein Bildschirm-Update erforderlich ist. Dieses Flag wird aller 100 ms abgefragt, wobei ggf. der gesamte Bildschirm im Hintergrund als Bitmap neu aufgebaut und eingeblendet wird. Anders als beim Original, verbrauchen Bildaufbau und -ausgabe somit keine Rechenkapazität des virtuellen Prozessors.

Wichtige Adressen

Konstanten

Adresse	Bemerkung
0018/9H	Enthält die Anfangsadresse des BWS, die z.B. zur Unterscheidung der Versionen des Interpreters genutzt werden kann. Inhalte sind: 3C50H für Version 2.4, 3C80H für Version 3.1, 3CA0H für Version 3.2 bzw. 3CB4H für Version 3.3.
002FH	Enthält die Anzahl der Zeichen/Zeile für Version 2.4 (immer 27)
0061H	Enthält die Anzahl der Zeichen/Zeile, also 29 oder 40 für die 3.x-Versionen
C000H	Startadresse des Grafikspeichers
E800H	Beginn des freien Speichers nach dem Grafikspeicher
1804H	Initialisierungsroutine der Grafikbibliothek
1800H	Aufrufroutine für Grafikfunktionen

Allgemeine Variablen Version 2.4

Adresse	Bemerkung
3C00/1H	Endadresse RAM (im Emulator initial BBFFH)
3C04/5H	Ergebnispuffer für Ausdrücke. Hier legt der Interpreter die Ergebnisse von Berechnungen ab.
3C06/7H	Programmlänge in Bytes
3C08/9H	Adresse des Cursors (im BWS)

Der Text-Bildschirmspeicher reicht immer von 3C50H bis 3D9FH und das BASIC-Programm beginnt stets an Adresse 3DA1H. Der Stack beginnt unterhalb des BWS.

Allgemeine Variablen Version 3.x

Die Symbolnamen und der Großteil der Anmerkungen in der folgenden Tabelle sind den Assembler-Listings der Versionen 3.1 und 3.3 von Frank Prüfer/Volker Pohlars entnommen.

Adresse	Symbol	Bemerkung
3C00/1H	PBEG	Adresse Programmstart
3C02/3H	PLEN	Programmlänge in Bytes
3C04/5H	RAMTOP	Endadresse RAM (im Emulator initial BBFFH)
3C06H	ZZ	Anzahl Bildschirmzeilen
3C08/9H	CPOS	Adresse des Cursors (im BWS)
Weitere Arbeitszellen (Hilfsvariablen)		
3C0A/BH	ez len	Univ. Merzkelle, u.a. Länge Eingabezeile
3C0C/DH	arnd	RND-Merkzelle
8C0E . . 11H	areg2	Arithmetikpuffer 2 (wird nach Operationen auf Null gesetzt)
3C12/13H	acb2	Arithmetik: 2. Operand/Ergebnis, CB
3C14/15H	alh2	Arithmetik: 2. Operand/Ergebnis LH
3C16H	assign2	Arithmetik: 2. Operand, Vorzeichen/Kommaverschiebung
3C17 . . 1AH	areg1	Arithmetikpuffer 1
3C1B/CH	acb1	Arithmetik: 1. Operans, CB
3C1D/EH	alh1	Arithmetik: 1. Operand, LH
3C1FH	assign1	Arithmetik: 1. Operand, Vorzeichen/Kommaverschiebung
3C20 . . 23H	ZWIREG	Zwischenregister 2. Operand bei Mult./Div.
3C24/5H	BZ	Adresse Anfang Eingabezeile im BWS (nicht Anfang der Zeile, in der der Cursor steht, sondern Cursorposition beim Beginn einer Eingabe)
3C26H/7H	READPT	READ-Pointer (Aktuelles Element bei READ)
3C28H	TACOD	Tastencode nach Ausführung von INKEY\$, nur bei Version 3.1 gesetzt, dort zur Feststellung, ob Taste gedrückt (sonst ist der Wert 0), geeignet
3C29/AH	KONV1	Konvertierungspuffer Zeilennummer
3C2B	KONV2	Konvertierungspuffer Zeile, danach Stack, 84 Bytes bei Version 3.1, 116 bei 3.2 und 136 bei 3.3; danach BWS, Länge von Zeilen/Spalten des Bildschirms abhängig, im Original höchstens bis 3FFFH

Variablen der Grafik-Erweiterung, initialisiert durch USR(1804H)

Adresse	Bemerkung
BC00/1H	Anzahl der Pixel in vertikaler Richtung
BC02/3H	Anzahl der Pixel in horizontaler Richtung
BC06/7H	Größe des genutzten Grafik-Bildspeichers in Bytes
BC08/9H	Adresse des ersten Elements des GA-Feldes
BC0A/BH	Byte-Adresse des aktuellen Pixels
BC0CH	Maske des aktuellen Pixels
BC0DH	Index des Bits in BC0A/BH, der dem aktuellen Pixel entspricht.
BC0EH	Anzahl der implementierten Grafikfunktionen
BC10/1H	Startadresse der Grafik-Funktionstabelle
BC30H . . BC47H	temporärer Variablenbereich der Grafik

Adressen von Systemroutinen

Die Adressen für Version 3.2 sowie die Beschreibungen sind größtenteils dem [Manuskript zur Version 3.2](#) von Frank Prüfer entnommen.

Adresse (Hex) Version				Beschreibung
2.4	3.1	3.2	3.3	
0088	0028	0028	0028	Ausgabe eines Zeichens aus A auf den Bildschirm (bei Version 3.x =Befehl RST 28H; U880-Maschinencode: 0EFH)
	0036	0036	0036	Sperren des Bildschirminterrupts, der Rechner arbeitet dann bis zur nächsten INPUT- oder PRINT-Anweisung wesentlich schneller
002C	0056	0056	0056	Löschen des Bildschirms
00D2	00C4	00C6	00C6	Eingabe eines Zeichens von der Tastatur nach A; wartet auf Tastendruck; bei Version 2.4 und 3.1 mit, sonst ohne Echoausgabe
	00E0	00E0	00E0	Tastaturstatusabfrage: A=0 wenn gerade keine Taste gedrückt, sonst A=Tastencode; wartet <u>nicht</u> auf Tastendruck HL enthält Tastatur-Abfragecode
0131	0138	0130	0130	Eingabe einer mit ENTER abgeschlossenen Zeichenkette von der Tastatur auf den Bildschirm; Ausgang: Version 2.4: DE=Adresse erstes Zeichen, HL=Adresse Cursor

				Version 3.x: IY=Anfangsadr. der Zeile, Endekennzeichen der Zeile: (ASCII=7FH)
015A	0212	0241	023D	Startadresse für BASIC-Kommandoeingabeebene; Einsprung in das Interpreter-Hauptprogramm (entspricht END-Anweisung)
02B5	0358	0377	0373	Ausgabe alphanumerischer Texte ab DE bis einschließlich Endekennung <NL> (1EH) oder <NUL> (00H; dann wird kein abschließender Zeilenwechsel ausgegeben); Version 3.1 wandelt alle Buchstaben in Kleinbuchstaben um.
0536				Berechnung von Integer-Ausdrücken in Version 2.4, <u>Eingang:</u> DE=Anfangsadresse des ASCII-kodierten Ausdrucks, Endekennzeichen z.B. Komma (2CH), Doppelpunkt (3AH) oder <NL> (1EH); der Ausdruck darf alle Operatoren enthalten, (AND bzw. OR als Token); Standardfunktionen müssen ebenfalls im Intern-Format angegeben sein (vgl. Schlüsselworttabelle). <u>Ausgang:</u> Ergebnis in HL sowie Arbeitszelle 0C04/5H, DE: Adresse des Endekennzeichens des Ausdrucks.
	0824	0813	081A	Berechnung von Integer-Ausdrücken in Version 3.x. Es wird die normale Ausdrucksberechnung aufgerufen und anschließend geprüft, ob das Ergebnis vom Integer-Typ ist. Alle weiteren Bemerkungen siehe nächste Zeile.
	082F	0820	0827	Berechnung beliebiger alphanumerischer Ausdrücke, <u>Eingang:</u> IY=Anfangsadresse des ASCII-kodierten Ausdrucks, Endekennzeichen z.B. Komma (2CH), Doppelpunkt (3AH) oder <NL> (1EH); der Ausdruck darf alle Operatoren enthalten, (AND bzw. OR als Token); Standardfunktionen müssen ebenfalls im Intern-Format angegeben sein (vgl. Schlüsselworttabelle). <u>Ausgang:</u> CBLH: Berechnungsergebnis; IY: Adresse des Endekennzeichens des Ausdrucks; DE und IX: unverändert . <u>Achtung:</u> Wenn man dem Aufruf einen zur Laufzeit eingegebenen String übergibt, muss man dafür sorgen, dass direkt am Ende des Ausdrucks tatsächlich so ein Endekennzeichen steht.
0803	0D8F	0D8B	0D8E	SAVE-Routine
0800	0E6C	0E5C	0E5F	LOAD-Routine
Tabellen				
076B... 07DD	0F12... 0FCD	0EF9... 0FC9	0EF9... 0FC9	Schlüsselworttabelle; Aufbau: Schlüsselwort im ASCII-Code (Bit7=0; bei Funktionen jeweils inkl. öffnender Klammer) gefolgt vom zugehörigen Token (Werte 0BCH...0FEH, Bit7=1)
07DE... 07FF	0FCE... 0FFF	0FCA... 0FFF	0FCA... 0FFF	Tabelle der Einsprungsadressen der Anweisungen; Aufbau: Je Eintrag 2 Bytes Einsprungsadresse. Die Tabelle enthält nur die Adressen der ‚echten‘ Anweisungen, nicht der Bestandteile von Ausdrücken, wobei der Token-Wert dem L-Byte der Adresse des Eintrags in der Tabelle entspricht.

Schlüsselwörter → Tokens (Hex.)

Nachfolgende Tabelle zeigt die internen Tokenwerte zu den jeweiligen Schlüsselwörtern. Dies sollte es prinzipiell ermöglichen, BASIC-Programme zwischen den Versionen zu konvertieren, sofern keine Schlüsselwörter verwendet werden, die in der jeweils anderen Version nicht existieren.

Schlüsselwort	2.4	3.1	3.2 / 3.3
END	DE	CE	CA
REM	E0	D0	CC
GOSUB	E2	D2	CE
CLEAR	E4		
CLS		D4	D0
IF	E6	D6	D2
THEN	D1	C0	C7
INPUT	E8	D8	D4
PRINT	EA	DA	D6
?		DA	D6
RETURN	EC	DE	DA
LOAD	F6	E8	E6
LET	F4	E6	E4
SAVE	F8	EA	E8
GOTO	EE	E0	DC
CALL			DE
IN(	D9	C8	C2
OUT	FC	EE	EC
BYTE	DD		
RUN	F0	E2	E0
LIST	F2	E4	E2
DIM		F4	F2
USR(		B4	BC
PLOT		F6	F4
UNPLOT		F8	F6
FOR		F0	EE
TO		BC	C9
STEP		BA	C8
NEXT		F2	F0
ASC(			C4
CHR\$(		BE	C3
INKEY\$		B8	C1
DATA		FE	FE
NEW	FE	FA	F8
PEEK(	D7	C6	BE

POKE	FA	EC	EA
RND		CA	
RND(	DA		BD
OR	D6	C4	C5
AND	D2	C2	C6
READ		FC	FC
RESTORE		DC	D8
INT(		B2	C0
LEN(		B0	BF
RANDOMIZE			FA

Startadressen der BASIC-Anweisungen im Interpreter

Nachfolgende Tabelle zeigt die Einsprungadressen der Behandlungsroutinen für die einzelnen ‚echten‘ Anweisungen des Interpreters.

Achtung: Dies sind keine Unterprogramme, müssen also mit JP nn aufgerufen werden und kehren nicht zum Aufrufer zurück.

Vom Interpreter selbst werden die Anweisungen während der Ausführung einer Befehlszeile aufgerufen und springen nach ihrer Abarbeitung entweder zur weiteren der Verarbeitung Befehlszeile in den Interpreter zurück oder brechen das Programm auf verschiedene Weise ab: END, SAVE, LOAD und LIST kehren zur Kommandoeingabe zurück, RUN startet das Programm neu und NEW führt einen kompletten Kaltstart durch.

Alle anderen Anweisungen springen zur weiteren Programmausführung auf die Adresse 0387H (Version 2.4), 0439H (Versionen 3.1 und 3.2) bzw. 0444H (Version 3.3) .

Anweisung	2.4	3.1	3.2	3.3
END	0193	0212	0241	023D
REM	0387	0435	0433	042F
GOSUB	0395	044F	044F	045A
CLEAR	03D0			
CLS		04A7	04A5	04B0
IF	0379	0424	0423	041F
INPUT	039B	045A	0456	0461
PRINT	03C0	0490	048E	0499
?		0490	048E	0499
RETURN	0398	0456	0437	0442
LOAD	0800	0E6C	0E5C	0E5F
LET	03D6	04AC	04AA	04B5
SAVE	0803	0D8F	0D8B	0D8E
GOTO	0354	03FA	03FB	03F7
CALL			05E1	05EB
OUT	0400	05AD	056C	0577
RUN	0335	03BF	03BC	03B8

LIST	02E2	0386	039E	039A
DIM		04D0	04BA	04C5
PLOT		0522	057A	0585
UNPLOT		0530	0587	0591
FOR		053B	0508	0513
NEXT		0545	0512	051D
DATA		0435	0433	042F
NEW	0171	01C2	01E5	01E5
POKE	03ED	059A	055C	0567
READ		05DD	05B3	05BD
RESTORE		05BB	0592	059C
RANDOMIZE			03F4	03F0

Undokumentiertes

Hier sind noch einige Informationen zusammengetragen, die in den bisherigen Veröffentlichungen zum BCS 3 nicht enthalten waren, mitunter aber nützlich sein können.

Variablenbezeichner

Der Interpreter des BCS 3 unterscheidet streng genommen – das ist vom Emulator unabhängig – bei Bezeichnern von Variablen zwischen Groß- und Kleinschreibung. Im Original spielt dies keine Rolle, da die Tastatur keine Eingabe von Kleinbuchstaben zulässt. Da im Emulator Kleinbuchstaben eingegeben werden können, ist diese Eigenschaft aber nutzbar, wodurch die Zahl der nutzbaren Variablen vergrößert wird. Entsprechende Programme können auch mit SAVE gespeichert und z.B. in den [JKCEMU](#) geladen werden.

Beim Import von BASIC-Quelltexten, die diese Eigenschaft ausnutzen, muss die Checkbox „a → A“ deaktiviert sein.

Hinweis: Der Interpreter des BCS 3 akzeptiert alle Zeichen im Bereich von 40H (@) bis 7EH (|) als ‚Buchstaben‘ in Bezeichnern.

Ein Extrembeispiel dafür findet sich in der Datei `komische_variablen.bas` im example-Archiv.

Nicht deklarierte Zeichenketten (Versionen 3.x)

Der Interpreter lässt es zu, Werte an Zeichenkettenvariablen zuzuweisen, ohne dass zuvor eine Deklaration mit DIM durchgeführt wurde. Solche nicht deklarierten Zeichenketten sind in der Verwendung eingeschränkt – es sind keine indizierten Zugriffe auf einzelne Zeichen möglich; außerdem funktionieren Verkettungen mit dem +-Operator nicht. Zudem ist die Länge auf 255 Bytes begrenzt. Einfache Zuweisungen und das Lesen der Zeichenkette sind aber möglich. Intern wird für solche Variablen einfach die Länge der zur Initialisierung verwendeten Zeichenkette und deren Adresse abgelegt.

Sinnvoll verwenden lässt sich so eine Zeichenkette immer dann, wenn im BASIC-Programm keine echten Manipulationen vorgenommen werden müssen. Es wird dann weniger Speicher benötigt als für eine deklarierte Variable, weil keine Kopie der initialisierenden Zeichenkette vorgenommen wird. Außerdem lässt sich so ein Alias auf irgendeinen Speicherbereich anlegen, der im Basic-Programm wie eine Zeichenkette behandelt wird.

Aufbau des Variablenbereichs

In den 3.x-Versionen beginnt der Variablenbereich unmittelbar hinter dem Programm (also bei Programmstart+Programmlänge) und ist eine einfache Aneinanderreihung der Variablen, wobei ein Variableneintrag folgende Struktur hat:

Byte-Index	Bedeutung	Bemerkung
0	Erstes Zeichen des Namens; zur Kennzeichnung eines Feldes wird hier 80H addiert	Auch deklarierte Zeichenketten sind Felder, haben also einen um 80H erhöhten ‚Anfangsbuchstaben‘
1	Zweites Zeichen des Namens oder 0; zur Kennzeichnung eines zweidimensionalen Feldes wird hier 80H addiert	Für zweidimensionale Felder mit nur einem Buchstaben im Namen steht hier 80H. Felder aus Zeichenketten sind im sind eigentlich zweidimensionale Felder aus Zeichen, also steht hier dann ‚\$‘+80H.
2	FE oder Low-Byte der Größe bzw. erstes Byte des Datenwerts	FE kennzeichnet einen nicht deklarierten String. Für einfache Variablen steht hier das erste Byte des Datums, für Felder das Low-Byte der Größe.
3	Länge oder High-Byte der Größe bzw. zweites Byte des Datenwerts	Für nicht deklarierte Strings ist dies die Gesamtlänge, bei einfachen Variablen das zweite Byte des Datums und bei Feldern das High-Byte der Größe
4..5	Adresse oder Elementzahl der letzten Dimension bzw. High- Word des Datenwerts	Bei nicht deklarierten Strings handelt es sich um die Adresse, bei Feldern um die Elementzahl (EZ) und bei einfachen Variablen um das dritte und vierte Byte des Datums.
6..	Datenbereich von Feldern	Hierzu gehören auch deklarierte Strings. Die Größe ergibt sich aus den Werten in Bytes 2 und 3.

Hinweise:

- Einfache (numerische) Variablen sind immer 6 Bytes lang, wobei die letzten vier den Wert beinhalten.
- Die ‚Größe‘ (BL) meint die Gesamtgröße des Datenbereichs von Feldern in Bytes.
- Der Datenbereich existiert nur bei deklarierten Feldern.

- Die Elementgröße (ES) beträgt 1 für Zeichenketten und 4 für numerische Werte.
Die Byte-Größe der ‚letzten‘ Dimension beträgt demnach $EZ * ES$ und die Größe der ‚ersten‘ Dimension $BL / (EZ * ES)$.

In der Datei `sysinfo.bas` im example-Archiv findet sich ein Beispiel zur Iteration über den gesamten Variablenbereich.

In Version 2.4 werden Variablen vom Speicherende an abwärts angelegt. Dabei ist ‚abwärts‘ wörtlich gemeint: Der erste Buchstabe des Namens der ersten definierten Variablen steht auf dem letzten Speicherplatz, der zweite auf dem vorletzten (kann Null sein, wenn der Name nur aus einem Buchstaben besteht), dann folgen das H-Byte und das L-Byte des Wertes. Da es nur Integer-Variablen und keine Strings oder Felder gibt, sind alle Variableneinträge 4 Bytes lang.

Zeilenaufbau

Ein BASIC-Programm wird zeilenweise abgelegt (die Startadresse steht in allen Versionen unter 3C00/1H). Jede Zeile beginnt mit einer 2 Byte breiten Zeilennummer und endet mit dem Ende-Byte 1EH. Der Zeileninhalt besteht aus Anweisungen, die ‚gepackt‘ abgelegt sind. Dies bedeutet: Eine Anweisung beginnt stets mit einem Token, das eine Anweisung (Schlüsselwort) repräsentiert (Ausnahme: fehlt ein solches Token, wird die Zuweisung LET als Standard angenommen). Darauf folgen die zur Anweisung gehörigen Informationen (Ausdrücke, Parameter) in Textform. Mehrere Anweisungen sind – wie im ursprünglichen Quelltext – durch einen Doppelpunkt (Versionen 3.x) bzw. ein Semikolon (Version 2.4) getrennt. Leerzeichen kommen außerhalb von Zeichenketten generell nicht vor.

Es ist grundsätzlich möglich, Quelltextzeilen zur Laufzeit zu manipulieren, etwa um Benutzer Ausdrücke zur Berechnung eingeben zu lassen. Voraussetzung dafür ist, dass die betreffende Zeile von Anfang an hinreichend lang ist, z.B. eine REM-Anweisung mit einer langen Zeichenkette enthält. Bei der Manipulation der Zeile ist dafür zu sorgen, dass am Ende wieder eine gültige (komprimierte) Quelltextzeile vorliegt.

Einen so realisierten einfachen Interpreter für numerische Ausdrücke gibt es in der Datei `eval.bas` im Beispiel-Archiv.

Verarbeitung einer Befehlszeile durch den Interpreter

Eine Befehlszeile besteht aus einer Folge von Anweisungen, die jeweils durch ein Semikolon (Version 2.4) bzw. einen Doppelpunkt voneinander getrennt sind. Eine gültige Befehlszeile enthält mindestens eine Anweisung; sie endet mit einem Zeilenumbruch (Zeichencode 1EH).

Anweisungen

Eine Anweisung beginnt – wie oben schon beschrieben – stets mit einem Anweisungstoken, mit einer einzigen Ausnahme: Da der LET-Befehl in der Notation weggelassen werden darf, wird beim Fehlen eines Anweisungstokens eine Zuweisung angenommen. Über das Anweisungstoken wird die

Adresse einer Behandlungsroutine bestimmt und zu dieser gesprungen. Dabei wird ein Zeiger auf das nächste Byte (Zeichen) in der Befehlszeile übergeben – in Version 2.4 im Registerpaar DE, in den 3.x-Versionen in IX. Eine Behandlungsroutine ist selbst für die Verarbeitung aller weiteren zur Anweisung gehörenden Zeichen (Parameter) verantwortlich und setzt – sofern sie zur Befehlsabarbeitung zurückkehrt – den Zeiger auf das nächste zu verarbeitende Zeichen.

Die Besonderheit der Behandlung der Zuweisung (das Weglassen von LET) ermöglicht die recht einfache Erweiterung des Interpreters um weitere (benutzerdefinierte) Anweisungen, was allerdings die Verfügbarkeit eines zusätzlichen ROM-Bereichs voraussetzt.

Man kann durch einen Patch der Original-ROMs den Sprungbefehl zur LET-Behandlung auf eine eigene Routine umleiten, die zusätzliche Schlüsselwörter für Anweisungen kennt. Wird eine solche Anweisung gefunden, wird sie behandelt und anschließend zur weiteren Befehlsbearbeitung (nächste Anweisung), ebenfalls per Sprungbefehl, zurückgekehrt, andernfalls zur LET-Behandlung gesprungen.

Der Emulator implementiert über eine solche Erweiterung beispielhaft eine CALL-Anweisung (Aufruf von Maschinencode-Unterprogrammen) für den Interpreter der Versionen 2.4 und 3.1.

Nachfolgende Tabelle fasst die Aufruf-Patch-Adressen, die Adressen LET-Behandlung sowie die Rückkehradressen zur weiteren Verarbeitung für die einzelnen Versionen zusammen.

	Version			
	2.4	3.1	3.2	3.3
Patch-Adresse	0344/5H	03E8/9H	03E3/4H	03DF/E0H
LET-Adresse	03D6H	04ACH	04AAH	04B5H
Nächste Anweisung	0387H	0439H	0439H	0444H

Parameter, Ausdrücke und Funktionen

Der typische Fall der Übergabe zusätzlicher Informationen an Anweisungen (IF-THEN, FOR-TO-NEXT sind hier die Ausnahmen) ist die Verwendung einer (ggf. auch leeren) Parameterliste, wobei als Trennzeichen (meist) ein Komma verwendet wird. Die einzelnen Parameter wiederum sind (numerische oder für die 3.x-Versionen auch alphanumerische) Ausdrücke, die ihrerseits aus Operanden und Operatoren bestehen können.

Für die Verarbeitung von Ausdrücken können die entsprechenden Routinen aus der Tabelle unter [Adressen von Systemroutinen](#) verwendet werden. Im Hinblick auf benutzerdefinierte Erweiterungen sind innerhalb der Ausdrücke die Operanden besonders interessant, weil hierzu die Standardfunktionen gehören. In allen Versionen des Interpreters wird hier eine Kette durchlaufen, in der das Vorkommen von Standardfunktionen anhand der ihnen zugeordneten Token geprüft wird. Wird eine solche Funktion gefunden, wird sie aufgerufen und ihr Rückgabewert als Operand verwendet. Hängt man sich in den Anfang dieser Kette ein, lassen sich benutzerdefinierte Funktionen als Erweiterungen des Interpreters implementieren. Obwohl das Prinzip immer gleich ist, gibt es in der konkreten Umsetzung ein paar Unterschiede zwischen Version 2.4 und den 3.x-Versionen.

In Version 2.4 wird das Doppelregister DE zur Adressierung des nächsten Zeichens in der zu verarbeitenden Zeile verwendet. Ein Funktionsaufruf (innerhalb der Operandenbestimmung) gibt sein Ergebnis, das stets vom Integertyp ist, sowohl in HL als auch in IY zurück.

In den Versionen 3.x ist enthält IY die Adresse des nächsten Zeichens und das Ergebnis wird in BC und HL zurückgegeben.

Erweiterungen sollten – dies gilt für alle Versionen – zunächst das Vorkommen einer der zusätzlichen Funktionen prüfen. Im Erfolgsfall wird sie behandelt und anschließend mit RET zum Aufrufer zurückgekehrt. Ansonsten erfolgt ein Sprung zur ursprünglichen Operandenberechnung.

Auch hierfür gibt es im Emulator Beispielimplementierungen, die die erweiterte Form der USR()-Funktion für die Versionen 2.4 und 3.1 bereitstellen.

Nachfolgende Tabelle fasst die Aufruf-Patch-Adressen und die Adressen der ggf. aufzurufenden Originalroutinen zur Operandenbestimmung für die einzelnen Versionen zusammen.

	Version			
	2.4	3.1	3.2	3.3
Patch-Adresse	064F/50H	09D0/1H	09DA/BH	09E1/2H
Original-UP	057BH	08A5H	0893H	089AH

Zeilennummern (Versionen 3.x)

Der Interpreter akzeptiert die Eingabe von Zeilennummern im Hexadezimalformat. Außerdem kann er eigentlich auch mit Zeilennummern größer 9999 umgehen. Im Original wird dies lediglich dadurch verhindert, dass beim Start (bzw. in der NEW-Routine) eine END-Anweisung mit der Nummer 9999 (für Version 3.1) bzw. 9984 (Version 3.2 und 3.3) eingetragen wird. Außerdem prüft der LIST-Befehl das H-Byte der Zeilennummer und bricht bei 27H (entspricht Zeilennummer 9984) ab. Mit dem Patch, der oben angegeben ist und im Emulator verwendet wird, lässt sich das beheben. Es sind dann Zeilennummern bis 32512 verfügbar.

Nützlich ist dies u. U. für die Portierung von Fremdprogrammen, die Zeilen oberhalb von 10000 verwenden, oder generell bei längeren Programmen, wenn man größere Abstände der Zeilennummern verwendet.

PEEK und ASC (Versionen 3.x)

Die Funktion PEEK ist in der Interpreterversion 3.1 auf Zeichenketten anwendbar und liefert dort den Code des ersten Zeichens. Somit kann PEEK an Stelle der ASC-Funktion verwendet werden, die es erst ab Version 3.2 gibt. Ab Version 3.2 funktioniert der Missbrauch von PEEK allerdings nicht. Um eine Abfrage eines Zeichencodes versionsübergreifend zu bekommen, könnte man z.B folgendes Unterprogramm mit Aufrufmakro schreiben:

```
$ORD IF PEEK(18H)=80H GOTO $OLD.ORD
  @A=ASC(@$) : GOTO $END.ORD
$OLD.ORD @A=PEEK(@$)
$END.ORD RETURN
```

```
#define ord(aval,char) @$=char : GOSUB $ORD : aval=@A
```

Im Argument *char* wird hier die Zeichenkette oder das Zeichen übergeben, *aval* bezeichnet die Variable, in der das Ergebnis abgelegt wird.

Erweitertes USR (Versionen 3.x)

In den Versionen 3.2 und 3.3, wie auch in der gepatchten Version 3.1, kann die Funktion USR mit einem zusätzlichen Argument aufgerufen werden, das an das jeweilige Unterprogramm weitergegeben wird. Sie sollte auch stets mit einem solchen Argument aufgerufen werden, auch dann, wenn es vom Unterprogramm nicht benötigt wird, weil andernfalls der Rückgabewert des USR-Aufrufs undefiniert ist.

Speicherbereiche und Speichererweiterungen

Im Original-BCS3 wird die Adressleitung A13 nicht dekodiert, weshalb die Speicherbereiche 0000H bis 1FFFH und 2000H bis 3FFFH ‚doppelt belegt‘ sind. Da sowohl Interpreter als auch Tastaturabfrage und Bildschirmsteuerung aber sehr diszipliniert sind und nur die spezifizierten Adressbereiche benutzen, lässt sich durch die Dekodierung von A13 freier Adressraum für zusätzliche ROM- und/oder RAM-Bereiche schaffen. Nutzbar ist hierfür mindestens der Adressbereich zwischen 1800H und 37FFH.

Der Bereich zwischen 1000H und 13FFH wird für die Tastaturabfrage benötigt; der von 3800H bis 3BFFH wird von der Bildschirmsteuerung verwendet und ist auf den RAM-Bereich zwischen 3C00H und 3FFFH gemapped, was auch die Begrenzung der Anzahl der Bildschirmzeilen im Original erklärt – der BWS darf nicht über 3FFFH hinaus reichen.

Im Bereich 1400H bis 1800H wird durch die Interrupt-Routine der Bildschirmsteuerung auf Adresse 1400H zugegriffen (in Version 2.4 lesend, in den 3.x-Versionen schreibend), weshalb der Bereich nicht anderweitig genutzt werden kann, zumindest nicht unter Einschluss der Adresse 1400H.

Oberhalb von 4000H ist Platz für RAM-Erweiterungen, theoretisch bis FFFFH. Allerdings kennt der Interpreter keine vorzeichenlosen Integer-Werte, weshalb bei ‚zu großem‘ Speicherausbau ein ‚negativer Wert‘ für den freien Speicher erkannt wird, was zu eigenartigen Seiteneffekten führen kann. Deshalb ist es zweckmäßig, hier eine Begrenzung zu setzen (im Emulator ist dies die Adresse BBFFH, die für alle Versionen und Zeilenzahlen für freien Speicher knapp unter 32K sorgt). Dies erreicht man durch eine Lücke im RAM-Ausbau, einen eingefügten ROM-Bereich oder dadurch, dass man Zugriffe auf eine einzelne Adresse blockiert (so macht es der Emulator).

Oberhalb davon kann man dann wieder beliebig Speicher einbauen, der – anders als der vom Interpreter erkannte Speicher – beim Aufruf von RUN auch nicht gelöscht wird.

Abweichungen vom Original

Tastaturabfrage mit `USR(0E0H)`

Das Aufrufergebnis von `USR(0E0H)` entspricht für Tasten, die auch im Original-BCS3 verfügbar sind, dem der Emulation für den BCS3 im [JKCEMU](#). Ob dies in jedem Fall dem des Originals entspricht, ließ sich nicht überprüfen. Für weitere Tasten, z.B. Umlaute, wird der Zeichencode zurückgegeben.

Mit dem Beispielprogramm `inkeytest.bas` können die Rückgabewerte getestet werden.

BASIC-Befehle `LOAD`, `SAVE` und `SAVED`

Die BASIC-Befehle führen zum Aufruf derselben Lade- und Speicherroutinen, die auch über die entsprechenden Buttons erreichbar sind. Nach dem Aufruf von `LOAD` wird allerdings nicht die erste Programmzeile angezeigt, sondern es erfolgt einfach ein NMI-Aufruf. Beim Abbruch des Ladedialogs bleibt das vorherige Programm erhalten.

Beim Speichern mit `SAVED` wird zusätzlich ein End-Block an die Datei angehängt, dessen Struktur dem bei der Tonbandausgabe, abzüglich des letzten Bytes mit dem festen Wert `AAH`, entspricht und der die Programmlänge enthält. Dies ist erforderlich, um beim späteren Laden mit `LOAD` die korrekte Länge wieder setzen zu können. Das Format des angehängten Blocks ist kompatibel zum Import des JKCEMU (dank eines Hinweises von Jens Müller), wodurch so gespeicherte Programme im Prinzip auch in den JKCEMU geladen werden können.

Zeilennummern

Der Emulator kann für die 3.x-Versionen mit Zeilennummern bis 32512 umgehen – siehe oben.

Verfügbare Zeilenzahl

Als Zeilenzahl kann für die 3.x-Versionen des Interpreters und unabhängig von der Spaltenzahl 4..32 angegeben werden. Das liegt daran, dass der Emulator weder auf knappen Speicherplatz noch auf das Timing oder die Adresszugriffe der Bildschirmsteuerung Rücksicht nehmen muss.

Für die Version 2.4 wird allerdings nur die Original-Bildschirmgröße von 12 Zeilen zu 27 Zeichen unterstützt.

Bildschirmdarstellung in Version 2.4

Der Bildschirm wird beim Löschen nicht mit Punkten, sondern mit Leerzeichen gefüllt; Zeilenendezeichen ist – wie bei den 3.x-Versionen – das Zeichen `7FH`.

Befehlserweiterungen

Der Emulator stellt zur Demonstration der Realisierung von Erweiterungen optional die Anweisung `CALL` und die Funktion `USR()`, beide in der Semantik der Version 3.2/3, für die Versionen 2.4 und 3.1 bereit.

Links

BCS3-Seite unter Homecomputer DDR von Volker Pohlers

<https://hc-ddr.hucki.net/wiki/doku.php/homecomputer/bcs3>

Hier gibt es auch die Original-[Dokumentationen zum BCS 3](#) (aus den rfe-Artikeln, die ROM-Inhalte sowie die [Beschreibung der Version 3.2](#) von Frank Prüfer.

Jenner's WWW-Homepage

<https://www.jennergruhle.de/bcs3.html>

Hier sind die ZASM-Listings für die Versionen 3.1 (40 Zeichen) und 3.2 (29 Zeichen) zu finden, dazu auch die des MC-Edit für die Versionen 3.1 und 3.3.

GitHub-Seite zum Z80-Emulator von Matt Howell

<https://github.com/DrGoldfire/Z80.js/tree/master>

JKCEMU-Seite von Jens Müller

<http://www.jens-mueller.org/jkcemu/>

jsz80asm – Z80 Assembler von UO Soft

<https://github.com/uosoft/jsz80asm>

EditArea (der erweiterte Editor)

<https://cdolivet.com/editarea/> bzw. <https://github.com/cdolivet/EditArea>

R.-Erik Ebert, September 2026
erik@euseb.de