Computer Shop

John Hardman
Andrew Hewson

Maschinencode-Routinen für den ZX Spectrum

Die 40 besten Programme mit einer Einführung und Erklärungen

ADD A, A 87
ADD A, B 80
ADD A, C 81
ADD A,D 82
ADD A, E 83
ADD A, H 84
ADD A, L 85

DEC A 3D
DEC B 05
DEC BC 0B
DEC C 0D
DEC D 15
DEC DE 1B
DEC E 1D
DEC H 25
DEC HL 2B
DEC IX DD2B
DEC IY FD2B
DEC L 2D
DEC SP 3B

Springer Basel AG

Computer Shop
Band 11

John Hardman / Andrew Hewson

Maschinencode – Routinen für den ZX Spectrum

Die 40 besten Programme, mit einer Einführung und Erklärungen

Aus dem Englischen von Ursula Rapp

Springer Basel AG

Die Originalausgabe erschien 1982 unter dem Titel:
"40 Best Machine Code Routines for the ZX Spectrum with Explanatory Text"
bei Hewson Consultants, Wallingford, Oxon, England
© 1982 Hewson Consultants

ZX Spectrum ist ein eingetragenes Warenzeichen der Firma
Sinclair Research Ltd, Cambridge, GB.

Der Inhalt des ZX Spectrum 16K ROM ist urheberrechtlich geschützt und
Eigentum der Firma Sinclair Research Ltd.

Die Programme und Routinen in diesem Buch unterliegen dem Urheberrecht
und sind nur für den privaten Gebrauch bestimmt.

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Hardman, John:
Maschinencode-Routinen für den ZX Spectrum : d.
40 besten Programme mit e. Einf. u. Erkl. / John
Hardman ; Andrew Hewson. Aus d. Engl. von Ursula
Rapp. – Basel ; Boston ; Stuttgart : Birkhäuser,
1983.
(Computer-Shop ; Bd. 11)
Einheitssacht.: Forty best machine code routines
for the ZX Spectrum with explanatory text 〈dt.〉
ISBN 978-3-7643-1559-7

NE: Hewson, Andrew:; GT

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte
vorbehalten. Kein Teil dieses Buches darf ohne schriftliche Genehmigung
des Verlages in irgendeiner Form durch Fotokopie, Mikrofilm, Kassetten
oder andere Verfahren reproduziert werden. Auch die Rechte
der Wiedergabe durch Vortrag, Funk und Fernsehen bleiben vorbehalten.

© 1983 Springer Basel AG
Ursprünglich erschienen bei der deutschsprachigen Ausgabe:
Birkhäuser Verlag, Basel 1983
Umschlaggestaltung: Bruckmann + Partner
ISBN 978-3-7643-1559-7 ISBN 978-3-0348-6775-7 (eBook)
DOI 10.1007/978-3-0348-6775-7

ISBN 978-3-7643-1559-7 ISBN 978-3-0348-6775-7 (eBook)
DOI 10.1007/978-3-0348-6775-7

Meiner Familie für ihre Geduld und besonders für Debbie, John und Mairi für ihre unaufhörliche Unterstützung. J. H.

Mit Dank für Janet, Gordon und Louise. A. D. H.

Inhaltsverzeichnis

Teil A

1 *Einleitung*

Maschinencode – wozu? 11
Wie man Maschinencode erlernen kann 12

2 *Die interne Struktur des ZX Spectrum*

Das Speicherabbild 14
PEEK und POKE 16
Die Displaydatei 18
Die Attribute 20
Der Druckerpuffer 22
Der BASIC-Programmbereich 22
Zahlendarstellung mit fünf Bytes 25
Der Variablenbereich 26
ROM-Routinen 26

3 *Z8ØA-Maschinensprache*

Bits 28
Bytes 30
Adressen 30
Die Register des Z8ØA 31
  Das Akkumulator-Register a 31
  Das Flag-Register f 31
  Die Zählregister b und c 33
  Die Adreßregister de und hl 34
  Die Indexregister ix und iy 34
  Der Stapelzeiger sp 35
  Der Befehlszähler pc 35
  Die Wechselregister af', bc', de', hl' 36
Über den Befehlssatz 36
Glossar von Maschinencode-Befehlen 37
  No Operation 37
  Laden 38
  Push und Pop 38
  Exchange 38
  8-Bit-Addition und -Subtraktion 39
  8-Bit-und, -oder, -entweder–oder 39
  Vergleichen 39

8-Bit-Inkrementierung und -Dekrementierung ... 39
16-Bit-Inkrementierung und -Dekrementierung ... 39
16-Bit-Addition und -Subtraktion ... 39
Sprung, Aufruf und Rücksprung ... 39
Bit-Befehle ... 40
Links-Dezimal-Rotation ... 41
Rechts-Dezimal-Rotation ... 41
Akkumulator-Operationen ... 41
Restart ... 42
Blockbehandlung ... 42

Teil B

4 Einleitung

Maschinencode-Ladeprogramm ... 45
Programm BP Maschinencode-Ladeprogramm ... 46

5 Verschiebe-Routinen (Scroll)

Verschieben von Attributen nach links ... 51
Verschieben von Attributen nach rechts ... 52
Verschieben von Attributen nach oben ... 54
Verschieben von Attributen nach unten ... 55
Verschieben um ein Zeichen nach links ... 56
Verschieben um ein Zeichen nach rechts ... 58
Verschieben um ein Zeichen nach oben ... 59
Verschieben um ein Zeichen nach unten ... 61
Verschieben um ein Pixel nach links ... 63
Verschieben um ein Pixel nach rechts ... 64
Verschieben um ein Pixel nach oben ... 66
Verschieben um ein Pixel nach unten ... 68

6 Display-Routinen

Verschmelzung von Bildern ... 71
Bildschirm-Inversion ... 72
Vertikale Zeichen-Inversion ... 73
Horizontale Zeichen-Inversion ... 75
Zeichen-Rotation im Uhrzeigersinn ... 76
Attributsänderung ... 79
Attributswechsel ... 80
Ausfüllen eines Gebiets ... 82
Zeichnen von Figuren ... 88
Vergrößern und Kopieren des Bildschirms ... 93

7 Routinen zum Manipulieren von Programmen

Streichen von Programmblöcken 101
Austauschen von Zeichen 103
REM-Beseitigung 104
REM-Erzeugung 108
Kompaktprogramm 110
Laden von Maschinencode in DATA-Anweisungen 113
Umwandlung von Buchstaben (klein/groß) 118

8 Routinen aus dem Werkzeugkasten

Umnumerieren 120
Freier Speicherplatz 128
Programmlänge 129
Zeilenadresse 130
Speicherkopie 132
Wert Null für alle Variablen 134
Variablenliste 137
Suchen und Auflisten 140
Suchen und Ersetzen 144
ROM-Suche 147
Instr$ 150

Anhang 155

Teil A

1 Einleitung

Dieser Band hat zum Ziel, sowohl für den Anfänger, als auch für den erfahrenen Computerbenutzer ein Buch bereitzustellen, in dem er rasch mehrere nützliche, interessante oder unterhaltsame Maschinencode-Routinen für den ZX Spectrum nachschlagen kann. Zu diesem Zweck besteht das Buch aus zwei Teilen. Teil A beschreibt die Merkmale des Spectrum, die für den Maschinencode-Programmierer von Interesse sind – was man überhaupt unter einer Maschinencode-Routine versteht, die wichtigen internen Merkmale und Routinen des Spectrum und die Struktur der Maschinensprache selbst.

Teil B schildert dann die eigentlichen Routinen. Sie werden in einer Standardform dargestellt, die in allen Einzelheiten am Anfang dieses Teils erklärt wird. Die Routinen sind in sich vollständig, so daß sie einzeln und ohne Bezug auf andere Routinen geladen werden können.

Man muß nicht unbedingt verstehen, wie eine Routine funktioniert, wenn man sie benutzen will, denn jede Routine kann mit dem einfachen Maschinencode-Ladeprogramm, das am Anfang von Teil B aufgelistet ist, geladen werden. Wenn Sie also wirklich ungeduldig sind und eine Routine fahren wollen, zum Beispiel die Routine "Variablenliste", blättern Sie einfach zur entsprechenden Seite, geben das Maschinencode-Ladeprogramm ein und fahren es mit RUN. Dann geben Sie die Dezimalzahlen aus der Spalte "einzugebende Zahlen" ein. Wenn alle Zahlen geladen sind, vergleichen Sie den Wert der Prüfsumme, der vom Maschinencode-Ladeprogramm ausgedruckt wird, mit dem bei der Routine angegebenen Wert. Wenn die beiden Werte übereinstimmen, können Sie sicher sein, daß Sie die Zahlen richtig eingegeben haben (es sei denn, Sie haben zwei oder mehrere Fehler gemacht, die sich gegenseitig aufheben). Nun können Sie die Routine verwenden.

Wenn Sie sich nicht zutrauen, eine lange Routine wie "Variablenliste" zu fahren, aber trotzdem so schnell wie möglich mit Maschinencode anfangen wollen, dann wählen Sie doch eine kürzere Routine. Auf diese Art verschwenden Sie nicht zuviel Zeit, wenn Sie den Faden verlieren oder zuviele Fehler machen. Die Routine "Verschieben von Attributen nach unten" (Scrollen) ist dafür ideal. Man muß wiederum einfach nur das Maschinencode-Ladeprogramm, das am Anfang von Teil B steht, eingeben und mit RUN laufen lassen und dann die Zahlen in der Spalte "einzugebende Zahlen" abschreiben. Wenn Sie damit fertig sind, sollten Sie sich davon überzeugen, daß die Prüfsumme richtig ist.

Wenn Sie die Verwendung von Maschinencode noch etwas aufschieben können, dann lesen Sie bitte weiter. Der Rest dieses Kapitels führt in die Grundideen dieses Buches ein und erklärt ausführlich, wie es aufgebaut ist. Wir empfehlen dem Anfänger, diese Information sorgfältig zu lesen; der erfahrene Benutzer wird sie wahrscheinlich nur zu überfliegen brauchen.

Der Mikroprozessor Z8ØA, der dem ZX Spectrum zugrunde liegt, versteht BASIC-Wörter wie PRINT, IF, TAB usw. nicht direkt. Statt dessen gehorcht er einer ihm eigenen Sprache – der Maschinensprache, häufig auch Maschinencode genannt. Die Anweisungen im ROM des Sinclair, die dem Spectrum seine "Persönlichkeit" geben, sind in dieser Sprache verfaßt. Sie bestehen aus einer Vielzahl von Routinen zur Eingabe, zum Auflisten, Interpretieren und Ausführen des besonderen BASIC-Dialekts, den der Spectrum verwendet. Tatsächlich sind die Routinen Gruppen von Anweisungen der Art "WHAT TO DO IF" (deutsch: was tun, wenn). Zum Beispiel sagen sie dem Z8ØA, was er tun soll, wenn der nächste BASIC-Befehl aus dem Wort PRINT besteht, was er dann tun soll, wenn das nächste Objekt ein Variablenname ist, und was er darauf tun soll, wenn der nächste Gegenstand ein Komma ist, usw.

Der Maschinencode besteht aus einer Folge von natürlichen Zahlen, die jeweils kleiner als 256 sind; er bestimmt die Handlungsweise des Z8ØA, indem er acht Schalter setzt, entsprechend dem Muster der Binärdarstellung dieser Zahl. Die Binärdarstellung von 237 ist zum Beispiel 111Ø11Ø1. Wenn also 237 vorgefunden wird, sind die acht Schalter jeweils ein-, ein- ein-, aus-, ein-, ein-, aus-, eingeschaltet.

Nun braucht aber der menschliche Benutzer den Befehl nicht unbedingt in der Binärform der Zahl zu betrachten, nur weil die Maschine dieser Version gehorcht. Wir verwenden gewöhnlich Dezimalzahlen, und deshalb erkennt das Maschinencode-Ladeprogramm in Teil B diese Form. Jedoch ist selbst ein String (d. h. eine Zeichenfolge) von Dezimalzahlen schwer zu interpretieren. Deshalb werden die Dezimalzahlen normalerweise nochmals umgewandelt in eine spezielle *Assemblersprache*, die zwar etwas rätselhaft, aber bei einiger Übung nicht zu schwierig ist. Jede Routine in Teil B wird sowohl in Assemblersprache als auch in Dezimalzahlen aufgelistet.

Es gibt ein besonderes Programm, das Assembler heißt und das bequem benutzt werden kann, um viele Maschinencode-Befehle in ein neues Programm zusammenzubringen oder zu *assemblieren* (von englisch: to assemble – deutsch: zusammensetzen). Daher kommt auch der Name Assemblersprache. Assembler sind komplizierte Programme, da Maschinencode sehr ausführlich ist und da sie meistens selbst in Maschinencode geschrieben sind. Es gibt jetzt schon mehrere solche Assembler für den Spectrum. Die Routinen in diesem Buch könnten auch mit Hilfe eines Assemblers geladen werden; jedoch sollte betont werden, daß das Maschinencode-Ladeprogramm diesem Zweck völlig angemessen ist.

Man benötigt nur eine einzige Zahl, um die einfacheren Z8ØA-Befehle zu spezifizieren. Der Befehl, den Inhalt von Register c in Register d zu kopieren, besteht zum Beispiel aus der Dezimalzahl 81. (Die Bedeutung des Wortes Register wird in Kapitel 3 ausführlicher erklärt. Im Moment reicht es aus, sich c und d als etwas ähnliches wie BASIC-Variable vorzustellen.) Für diese Befehle gibt es eine eindeutige Zuordnung zwischen der Dezimalzahl und der Version in Assemblersprache, so daß die Dezimalzahl 81 beispielsweise in Assemblersprache als ld d,c geschrieben wird. "ld" steht übrigens für "load" (deutsch: laden). Viele Befehle in Assemblersprache bestehen aus ähnlichen einfachen Abkürzungen, und aus diesem Grunde heißen sie oft Mnemonics (d. h. Gedächtnishilfen).

Man braucht für kompliziertere Anweisungen zwei, drei oder sogar vier Zahlen, um sie vollständig zu spezifizieren. In diesem Fall wird ein einziges Assembler-Mnemonic verwendet, das für alle diese Zahlen steht. In Tabelle 1.1 stehen ein paar Beispiele von Zahlen und ihren Mnemonics, sowie eine kurze Erklärung.

Tabelle 1.1
Einige Beispiele von Maschinencode-Befehlen für den Z8ØA

Ref	Dezimal	Assemblersprache	Kommentar
(a)	81	ld, d,c	d mit dem Inhalt von c laden
(b)	14 27	ld c, 27	die Zahl 27 in c laden
(c)	14 13	ld c, 13	die Zahl 13 in c laden
(d)	33 27 52	ld hl, 13339	13339 in das Registerpaar hl laden, man beachte, daß 27+256*52=13339; 27 wird in l abgelegt; 52 wird in h abgelegt
(e)	221 33 27 52	ld ix, 13339	13339 in das Registerpaar ix laden

Zeile (a) in der Tabelle ist das oben erwähnte Beispiel ld d,c. Die Zeilen (b) und (c) zeigen, wie eine natürliche Zahl kleiner als 255 unter Verwendung zweier aufeinander folgender Zahlen in ein Register geladen werden kann: die erste Zahl legt fest, welcher Vorgang ausgeführt werden soll, die zweite, welche Zahl geladen werden soll. Zeile (d) führt vor, wie eine große ganze Zahl in zwei Register, h und l, geladen werden kann. Dieses Mal bestimmen die zweite und dritte Zahl die zu ladende Zahl. Das letzte Beispiel in Zeile (e) veranschaulicht einen Code aus vier Zahlen, mit dem eine große ganze Zahl in das Registerpaar ix geladen werden kann. Man beachte, daß drei der vier Codezahlen auch in Zeile (d) erscheinen. In der Tat spezifiziert die erste Zahl das Paar ix anstatt hl.

Die Struktur der Maschinensprache wird in Kapitel 3 detaillierter beschrieben; eine vollständige Liste von Assembler-Mnemonics wird in Anhang A angegeben. Eine recht wichtige Frage, die hier beantwortet werden sollte, ist:

Maschinencode – wozu?

Bei jeder Programmiersprache auf jedem Computer scheint es immer Aufgaben zu geben, welche der Computer schlecht lösen kann. Die Problematik besteht darin, daß manche Aufgaben entweder nicht in der verfügbaren Sprache aufgeschrieben werden können oder, wenn dies möglich ist, sie nur sehr langsam ausgeführt werden. Der ZX Spectrum bildet in dieser Hinsicht keine Ausnahme.

Man betrachte zum Beispiel das Problem, die ganze Bildschirmanzeige am oberen Ende des RAM zu speichern, d. h. zu sichern (von englisch: to save), oder sie wieder zurückzukopieren, vielleicht mit der Absicht eines Cartoon-Effekts durch "Hin- und Herschalten" zwischen verschiedenen Anzeigen. Die Displaydatei und die Attribute für die Farbgebung belegen zusammen 6912 Bytes. Deshalb ist es notwendig, RAMTOP nach unten zu verlegen, und zwar auf der 16k-Maschine auf 32768 – 6912 = 25856, um damit genügend Platz für die Kopie des Displays außerhalb des BASIC-Bereichs zur Verfügung zu stellen. (Auf der 48k-Maschine sind es 65536 – 6912 = 58624.) Das folgende einfache BASIC-Programm speichert die Bildschirmanzeige; aber es braucht lange – ungefähr 7Ø Sekunden.

```
10  FOR i = 0 to 6911
20  POKE 25856 + i, PEEK (16384 + i)
30  NEXT i
```

Der Spectrum braucht so lange, weil er die meiste Zeit damit verbraucht, die Befehle zu decodieren, bevor er sie ausführt. Er braucht auch einen bestimmten Zeitraum zum Konvertieren von Zahlen aus der 2-Byte-Form ganzer Zahlen, die der Z80A versteht, in die 5-Byte-Dezimalform, in welcher der Schleifenzähler vorliegt, und umgekehrt; ebenso zum Ausführen von 5-Byte-Arithmetik. Er führt folgende Schritte durch:

1) Addition von i zu 16384
2) Konvertieren des Ergebnisses in 2-Byte-Form
3) Auffinden des Inhalts der PEEK-Adresse
4) Addition von i zu 25856
5) Konvertieren des Ergebnisses in 2-Byte-Form
6) Speichern des gefundenen Wertes in der POKE-Adresse
7) Addition von eins zum Wert von i und Speichern des Ergebnisses
8) Subtraktion von i von 6911. Wenn das Ergebnis positiv oder null ist, soll ein Sprung zu 1) folgen.

Bei jedem Schleifendurchgang muß der Spectrum den Befehl erneut decodieren, weil er kein "Gedächtnis" für die früheren Operationen hat. Man kann leicht einsehen, daß der Computer mehr als 99% der Zeit damit verbringt, die Ausführung der Aufgabe vorzubereiten, anstatt sie auszuführen. Es ist deshalb nicht überraschend, daß eine Maschinencode-Routine zur Speicherung des Bildschirms mehr oder weniger sofort mit der Ausführung beginnt. Ein Beispiel für eine solche Routine wird in Teil B angegeben.

Wie man Maschinencode erlernen kann

Die Maschinensprache des Mikroprozessors Z80A ist sehr kompliziert. Um alle ihre Möglichkeiten zu verstehen, braucht man ein gutes Nachschlagewerk, viel Überlegung und viel Übung. Es stehen mehrere Bücher zur Verfügung. Das Standardwerk ist "Die Programmierung des Z80" von Rodnay Zaks, erschienen bei Sybex (ISBN 3-88745-006-X). Es enthält eine Fülle von Informationen über die Hardware-Organisation des Mikroprozessors; es listet außerdem vollständige Einzelheiten des Befehlssatzes auf. Dem Anfänger mag es recht ungeheuerlich erscheinen, weil es mehr als 600 Seiten hat.

Eine einfachere Darstellung ist "ZX Spectrum Maschinencode" von Ian Stewart und Robin Jones, erschienen bei Birkhäuser (ISBN 3-7643-1535-0). Das Buch beginnt auf einer elementaren Stufe; es behandelt auf unterhaltsame Art die wichtigsten Aspekte des Maschinencode.

Dieses Buch soll nicht nur eine Einführung in den Maschinencode für den Anfänger sein, sondern es soll auch für den erfahrenen Benutzer brauchbar sein. Es gibt dem Leser einen starken Anreiz, Maschinencode zu lernen, weil es ihm Routinen verschafft, die er in seine eigenen BASIC- oder Maschinencode-Programme mit oder ohne Umarbeitung einbauen kann.

Die meisten Routinen sind eng mit der Struktur des ZX Spectrum ver-

knüpft, und deshalb befaßt sich das nächste Kapitel ausführlicher mit diesem Thema. Es behandelt zum Beispiel die Form der Displaydatei, den Programmbereich und den Variablenbereich; es erklärt das Layout von BASIC-Programmzeilen und führt in die 5-Byte-Gleitkomma-Arithmetik ein. In Teil B wird der Inhalt dieses Kapitels vorausgesetzt.

Das dritte Kapitel erklärt die Maschinensprache des Z8Ø im einzelnen und beschreibt dabei viele Punkte, die später vorausgesetzt werden. Es umfaßt ein Glossar des Befehlssatzes, welches die meisten herausragenden Sachverhalte einschließt, ohne die detaillierten Beschreibungen in Zaks' Buch zu wiederholen.

2 Die interne Struktur des ZX Spectrum

Ein Computer ist eine Maschine, die eine Folge von Befehlen speichern und sie dann ausführen kann. Natürlich braucht er dazu einen Speicher, in dem die Befehle abgelegt werden können. Der ZX Spectrum hat zwei verschiedene Speichertypen. Der erste Typ ist ein Nur-Lese-Speicher (englisch: *R*ead *O*nly *M*emory, kurz ROM), der die feste, vom Hersteller in die Maschine eingebaute Befehlsmenge enthält. Der zweite Typ ist ein Speicher mit wahlfreiem Zugriff (englisch: *R*andom *A*ccess *M*emory, kurz RAM).

Der Speicher mit wahlfreiem Zugriff oder RAM, wie wir ihn ab jetzt nennen werden, ist der Notizblock des Spectrum. Wenn der Computer eine Aufgabe ausführt, schaut er ständig, was im RAM steht (er "liest" aus dem Speicher). Außerdem ändert er dabei den RAM-Inhalt (er "schreibt" in den Speicher). Der Spectrum benutzt seinen Notizblock nicht wahllos. Verschiedene Teile des RAM werden verwendet, um verschiedene Arten von Information zu speichern. Zum Beispiel wird ein vom Benutzer eingegebenes BASIC-Programm in einem Teil des RAM gespeichert, während die Variablen, die vom Programm benutzt werden, anderswo gespeichert werden. Die Größe des Notizblocks ist begrenzt, und deshalb ist der Computer darauf bedacht, der Information, die er enthält, gerade genug Platz (und nicht mehr) zuzuteilen. Demgemäß wird der übrige Platz immer an einem Ort zusammengefaßt. Wenn nun beispielsweise ein Benutzer eine Zeile zu seinem Programm hinzufügen will, kann die Information im RAM verschoben werden, wobei ein Teil des übrigen Platzes verbraucht wird, um die zusätzliche Zeile unterzubringen.

Der größte Teil dieses Kapitels erklärt ausführlich, wie der Spectrum das RAM aufbaut, denn viele Routinen aus Teil B sind so angelegt, daß sie das RAM manipulieren. Deshalb muß der Leser den Inhalt dieses Kapitels verstehen, wenn er den Aufbau der Routinen wirklich begreifen und sie nicht nur blind benutzen soll. Das Kapitel umfaßt die Displaydatei, die Attribute, den Druckerpuffer, die Systemvariablen, den Programmbereich und den Variablenbereich. Der letzte Teil beschreibt die Routinen im ROM, auf die in Teil B verwiesen wird.

Das Speicherabbild

Es gibt 16384 Speicherplätze im RAM des nicht-erweiterten ZX Spectrum. (Die erweiterte Version enthält zusätzlich weitere 32768 Speicherplätze, was zusammen 49152 Speicherplätze ergibt.) Jeder Speicherplatz kann eine einzelne ganze Zahl zwischen Ø und 255 einschließlich aufnehmen. Er wird durch seine Speicheradresse, eine natürliche Zahl, gekennzeichnet.

Die Adressen Ø bis 16383 werden dem Festwert-Speicher, dem ROM, zugewiesen. Deshalb ist 16384 die erste Adresse, die dem RAM zugeteilt wird. Tabelle 2.1 ist das Speicherabbild des Spectrum. Sie zeigt, wie das RAM von der Adresse 16384 ausgehend benutzt wird. Die Displaydatei zum Beispiel, welche die aktuelle, auf dem Bildschirm gezeigte Information enthält, nimmt die Speicherplätze 16384 bis 22527 ein. Die Attribute, welche die Farbe, die Helligkeit,

usw. der Bildschirmanzeige bestimmen, folgen unmittelbar danach auf den Plätzen 22528 bis 23295.

Die ersten fünf Startadressen in Spalte eins von Tabelle 2.1 sind alle fest, weil die Displaydatei, die Attribute usw. alle einen festen Platzbereich einnehmen. Der fünfte Bereich ist den Microdrive-Abbildern zugeteilt. Wenn ein Microdrive an den Spectrum angeschlossen ist, so enthält dieser Bereich Information über die Anordnung der Daten im Microdrive. Wenn kein Microdrive angeschlossen ist, wird der Bereich nicht gebraucht. In diesem Fall wird der sechste Bereich, Kanal-Information, unmittelbar nach dem vierten Bereich, den Systemvariablen, plaziert. Dies geschieht in Übereinstimmung mit der Arbeitsweise des Spectrum, wo immer möglich, Platz zu sparen. Deshalb ist die Startadresse der Kanal-Information und aller folgenden Bereiche nicht festgelegt, sondern kann im RAM auf- und abgleiten.

Der Spectrum verfolgt die Startadressen aller dieser Bereiche, indem er den gegenwärtigen Wert jeder Adresse im Bereich der Systemvariablen speichert. Dieser Bereich liegt vor dem Microdrive-Abbild und reicht von Platz 23552 bis 23733 einschließlich. Es kann also nicht die Rede davon sein, daß auch dieser Bereich im RAM auf- und abgleiten würde! Die Adresse innerhalb des Bereiches, der die Startadressen aller gleitenden Bereiche enthält, ist in Tabelle 2.1 in der zweiten Spalte aufgelistet. Beispielsweise ist die Adresse des BASIC-Programmbereiches in Zelle 23635 innerhalb des Bereiches der Systemvariablen abgelegt.

Tabelle 2.1
Das Speicherabbild. Der Stapelzeiger, sp, ist nicht im RAM, sondern in Register sp im Z8ØA Mikroprozessor abgelegt.

Startadresse oder Name der Systemvariablen	Speicherplatz der Systemvariablen	Speicherinhalt
16384	–	Displaydatei
22528	–	Attribute zur Farbgebung
23296	–	Druckerpuffer
23552	–	Systemvariablen
23734	–	Microdrive-Abbild
CHANS	23631	Kanal-Information
PROG	23635	BASIC-Programm
VARS	23627	Variablen
E_LINE	23641	Befehl/Zeile, die gerade editiert wird
WORKSP	23649	INPUT – Daten
STKBOT	23651	Beginn Rechnerstapel
STKEND	23653	Beginn freier Platz
sp	–	Maschinenstapel und GOSUB-Stapel
RAMTOP	2373Ø	Maschinencode-Routinen des Benutzers, bzw. Ende des BASIC-Bereichs
UDG	23675	selbstdefinierte Zeichen (englisch: *u*ser *d*efined *g*raphics)
P-RAMT	23732	Ende des RAM

Es ist ziemlich umständlich, sich auf jede Systemvariable durch die Adresse, in der sie aufbewahrt wird, zu beziehen. Deshalb gibt man ihr einen Namen – PROG im Fall des Speicherplatzes, in dem die Adresse des BASIC-Programmbereichs steht. Diese Namen sind nur für den Gebrauch des Benutzers gedacht, sie werden vom Spectrum nicht erkannt. Wenn man also die Zeile

PRINT PROG

eingibt, so wird die Fehlermeldung "2 Variable not found" ausgedruckt; es sei denn, eine BASIC-Variable mit Namen PROG wurde vorher von einem Programm oder vom Benutzer erzeugt. Der Wert einer solchen BASIC-Variablen hätte jedoch nichts mit dem Wert der Systemvariablen PROG zu tun.

PEEK und POKE

Das Speicherabbild ist der Schlüssel, mit dem man versteht, wie der Spectrum das RAM benutzt. Aber die Schlüssel zur Erforschung des RAM sind die BASIC Keywords PEEK und POKE. Sie erlauben es dem Benutzer, den Inhalt jedes Speicherplatzes anzuschauen beziehungsweise zu verändern.

PEEK ist eine Funktion der Form:

PEEK Adresse

Die Adresse kann eine natürliche Zahl zwischen Ø und 65535 sein. Sie kann auch ein arithmetischer Ausdruck sein, dessen Auswertung eine solche positive Zahl ergibt. Es ist wichtig, einen arithmetischen Ausdruck in Klammern zu schreiben, denn

PEEK 16384 + 2

wird interpretiert als: 2 zum Ergebnis von

PEEK 16384

dazuaddiert. Hingegen wird

PEEK (16384 + 2)

interpretiert als

PEEK (16386).

Der Wert, der durch die PEEK-Funktion erhalten wird, ist diejenige Zahl, die gegenwärtig in der fraglichen Adresse abgelegt ist. Dies ist immer eine positive ganze Zahl zwischen Ø und 255 einschließlich. Oben wurde erklärt, daß die Systemvariable PROG in der Adresse 23635 abgelegt ist, aber daß der Wert von PROG, eine Adresse im RAM, immer viel größer als 255 ist. Man braucht deshalb zwei benachbarte Adressen, 23635 *und* 23636, um ihn unterzubringen. Der Wert von PROG kann gedruckt werden durch Eingabe von:

PRINT "PROG ="; PEEK 23635 + 256 * PEEK 23636

Auf diese Art werden alle Adressen in zwei benachbarten Speicherplätzen aufbewahrt. Sie können durch folgende Eingabe inspiziert werden:

PRINT PEEK erste Adresse + 256 * PEEK folgende Adresse

Wenn zum Beispiel ein Spectrum ohne angeschlossenen Microdrive benutzt wird, so existiert der Bereich des Microdrive-Abbilds nicht, und die Kanal-Information folgt unmittelbar auf den Bereich der Systemvariablen. So ist dann der Wert der Systemvariable CHANS derselbe wie die Startadresse des Microdrive-Abbilds, wenn es dieses gäbe, nämlich 23734. CHANS ist in 23631 *und* 23632 abgelegt, und deshalb ergibt die Eingabe

PRINT PEEK 23631 + 256 * PEEK 23632

den Wert 23734.

Die Funktion PEEK kann benutzt werden, um den Inhalt irgendeiner Zelle im Speicher, einschließlich der festen Befehle im ROM, anzuschauen. Sie ist deshalb ein sehr wichtiges Werkzeug. Mit PEEK an irgendeine Stelle zu gehen, wird im Spectrum keinen Zusammenbruch und keine Verfälschung von Programmen oder Variablen verursachen. Gelegentlich können die Resultate von PEEK irreführend sein, weil sich der Inhalt des Speicherplatzes, an den man mit PEEK geht, während oder gleich nach der Ausführung des Befehls ändern kann. Wenn man beispielsweise mit PEEK zum Inhalt derjenigen Zellen geht, die der oberen linken Ecke der Bildschirmanzeige zugewiesen sind, und wenn die Resultate durch den Befehl PRINT in die obere linke Ecke des Bildschirms gedruckt werden, so ist die Information schon überholt bis der Benutzer sie sieht.

Der Befehl POKE ist insgesamt gefährlicher als die Funktion PEEK. Der Benutzer greift nämlich durch seinen Aufruf in die Funktionsweise des Spectrum ein. So ist es gut möglich, Unsinn aus der Information im RAM zu machen, indem man diesen Befehl benutzt. Dies kann nämlich bewirken, daß der Rechner zusammenbricht oder anhält und einen Fehlercode anzeigt.

Der Befehl hat folgende Form:

POKE Adresse, Zahl

Wiederum ist die Adresse eine natürliche Zahl zwischen Ø und 65535 einschließlich oder ein arithmetischer Ausdruck, der nach seiner Auswertung eine solche Zahl ergibt. In diesem Fall ist es nicht notwendig, einen arithmetischen Ausdruck in Klammern zu schreiben, da POKE ein Befehl und keine Funktion ist. Daher kann es auch nicht als Ganzes ausgewertet werden. Die Zahl, die in die Speicherzelle "eingePOKEt" wird, muß zwischen Ø und 255 einschließlich liegen.

Der Spectrum akzeptiert einen POKE-Befehl und führt ihn aus, selbst dann, wenn er damit eine Zahl in einer Adresse im ROM (d. h. einer Adresse zwischen Ø und 16383) ablegen soll. Allerdings erreicht diese Zahl ihr Ziel nie. Diese Tatsache kann am Fahren des folgenden Programms mit RUN gezeigt werden:

```
10  PRINT PEEK 0
20  POKE 0,92
30  PRINT PEEK 0
```

Die Zeilen 10 und 30 drucken den Wert 243, der zufälligerweise der Inhalt der Zelle 0 ist. Zeile 20 hat keine Wirkung.

Die Displaydatei

Das normale Display besteht aus 24 Zeilen mit jeweils 32 Zeichen. Wir haben gesehen, daß die Displaydatei die Zellen 16384 bis 22527 besetzt, d. h. insgesamt 6144 Zellen. Daher beträgt die Anzahl der Zellen, die pro Zeichen verwendet werden,

$$6144 / (24 * 32) = 8$$

Die einfachste Art, einen Gesamteindruck vom Aufbau des Displays zu bekommen, ist es, mit PRINT ein Bild auf den Bildschirm drucken zu lassen, den Bildschirm auf Band mit SAVE abzuspeichern, dann den Bildschirm zu löschen, und schließlich das Bild wieder zurückzuladen. Das Programm P2.1 speichert den Bildschirm mit SAVE und lädt ihn mit LOAD. Zu Erzeugung des Originalbildes wird das 5. Grafikzeichen verwendet.

Programm P2.1

Ein Programm, um den Bildschirm mit SAVE zu speichern, zu löschen und wieder mit LOAD zu laden.

```
100  FOR i = 0 TO 703
110  PRINT "▐";
120  NEXT i
130  SAVE "Bild" SCREEN$
140  CLS
150  INPUT "Spulen Sie die Kassette zurueck, spielen Sie sie ab und
     druecken Sie dann eine Taste"; Z$
160  LOAD "Bild" SCREEN$
```

Wenn das Bild vom Band zurückgeladen wird mittels LOAD, wird klar, daß das Display in drei Teilbereiche mit jeweils acht Zeichenzeilen (gewöhnliche Schreibzeilen) zerlegt ist. Jede Zeichenzeile ist außerdem in acht Zeilen, bestehend aus Pixeln (von engl.: picture element, deutsch: Bildelement), zerlegt. Erstaunlicherweise lädt der Spectrum mit LOAD nicht die acht Pixel-Zeilen, welche die erste Zeichenzeile bilden, gefolgt von den acht Pixel-Zeilen, welche

die zweite Zeichenzeile bilden usw. Statt dessen lädt er die obersten Pixel-Zeilen der ersten acht Zeichenzeilen, gefolgt von den nächsten Pixel-Zeilen derselben acht Zeichenzeilen usw. Die auf diesen oberen Teil des Displays folgenden acht Zeichenzeilen bilden den mittleren, die letzten acht Zeichenzeilen den unteren Teil des Displays.

Man kann die Form der Anzeige auch dadurch verstehen, daß man sich überlegt, wo die acht Bytes abgelegt sind, die verwendet werden, um das Zeichen in der linken oberen Ecke des Bildschirms zu erzeugen. Das erste Byte erzeugt das oberste Achtel dieses Zeichens und befindet sich am Anfang der Displaydatei in Adresse 16384. Wenn man ein bißchen experimentiert, zeigt es sich, daß

```
POKE 16384,Ø
```

die oberste Zeile mit acht Pixeln, die zum ersten Zeichen gehört, leert. Hingegen läßt

```
POKE 16384,255
```

alle Pixel aufleuchten. Wenn man Zahlen zwischen Ø und 255 POKEt, so hat das eine scheckige Wirkung.

Diejenige Zeile mit acht Pixeln, die an zweiter Stelle von oben im ersten Zeichen auf dem Bildschirm steht, wird nicht von der in Zelle 16385 abgelegten Zahl erzeugt, sondern diese Zelle wird für die oberste Pixel-Zeile im benachbarten Zeichen benutzt. Es gibt 32 Zeichen in einer Zeile und acht Zeilen in einem Teilbereich, so daß die zweitoberste Zeile mit acht Pixeln im ersten Zeichen durch diejenige Zahl erzeugt wird, die in Zelle

16384 + 32 * 8 = 1664Ø

steht.

Ähnliche Überlegungen kann man über die übrigen sechs Zeilen mit jeweils acht Pixeln anstellen. Die Form des Zeichens in der linken oberen Ecke des Bildschirms wird durch den Inhalt der Adressen

16384, 1664Ø, 16896, 17152, 174Ø8, 17664, 1792Ø, 18176

bestimmt.

Das Programm P2.2 ermöglicht es dem Benutzer zu experimentieren, indem er mit POKE verschiedene Zahlen in diese acht Zellen setzt.

Programm P2.2

Ein Programm, um das Zeichen in der linken oberen Ecke (LOE) aufzubauen.

```
1Ø  REM Routine, um ein Zeichen in die LOE des Bildschirms zu setzen
2Ø  INPUT "Ein Zeichen wird von acht Bytes gebildet, die jeweils zwi-
    schen Ø und 255 einschließlich liegen. Geben Sie die Nummer des
    Bytes (Ø bis 7) ein. "; n
```

```
3Ø  IF n < Ø OR n > 7 OR n <> INT n THEN BEEP .2,24: GO TO 2Ø
4Ø  INPUT "Geben Sie den Inhalt des Bytes ein ";m
50  IF m < Ø OR m > 255 OR m <> INT m THEN BEEP .2,24: GO
    TO 4Ø
6Ø  POKE 16384 + 8 * 32 * n,m : GO TO 2Ø
```

Jede Zelle in der Displaydatei überwacht den Zustand von acht Pixeln auf dem Bildschirm. Diese Kontrolle wird folgendermaßen ausgeübt: die Zahl, die in einer vorgegebenen Zelle abgelegt ist, wird in ihre Binärdarstellung umgewandelt. Dann werden die acht Pixel gemäß dem Null-Eins-Muster der acht binären Ziffern gesetzt. Zum Beispiel hat 24Ø die Binärdarstellung

1111ØØØØ

Wenn nun eine Zelle die Zahl 24Ø enthält, leuchten vier der zugehörigen acht Pixel auf; die restlichen vier Pixel bleiben dunkel.

Um es noch einmal zusammenzufassen: die Displaydatei besteht aus 6144 Zellen, wobei je acht Zellen einer Zeichenposition zugewiesen sind. Jede Zelle schreibt den Zustand eines horizontalen Streifens von acht Pixeln vor. Die Zellen, die einer vorgegebenen Zeichenposition zugewiesen sind, liegen nicht nebeneinander; statt dessen ist das Display in drei Teilbereiche aufgeteilt. Innerhalb jedes Teilbereiches trennen 256 Zellen immer 2 der 8 Bytes einer Zeichenposition.

Die Attribute

Der Inhalt der Displaydatei bestimmt nur, welche Pixel auf dem Bildschirm aufleuchten. Die Farbe des Hintergrunds, diejenige des Vordergrunds, sowie die Helligkeits- und Blinkzustände können mit den Funktionen PAPER, INK, BRIGHT und FLASH angesteuert werden. Sie werden durch die Attribute bestimmt. Der Bereich der Attribute zur Farbgebung (etc.) belegt die Zellen 22528 bis 23295, wobei jeder der 768 Zeichenpositionen eine Zelle zugeordnet wird. Im Gegensatz zur Displaydatei werden die Speicherplätze den Zeichenpositionen in natürlicher Weise zugeteilt, d. h. man fängt in der linken oberen Ecke an und geht von links nach rechts und von oben nach unten.

Jede Zelle legt sowohl die Farbe des Vordergrunds (INK, deutsch: Tinte) als auch des Hintergrunds (PAPER) an der Stelle, welcher sie zugewiesen ist, fest. Dabei kann sie aus den acht Farben auswählen, die über den obersten Tasten auf der Tastatur des Spectrum zu sehen sind. Sie bestimmt auch, ob die Stelle hell (BRIGHT) ist, und ob sie blinkt (FLASH). Die vier Parameter werden unter Benutzung der folgenden Rechnung codiert:

Wert des Attributs = 128 * FLASH + 64 * BRIGHT +
8 * PAPER + INK

FLASH und BRIGHT haben den Wert 1, wenn der entsprechende Zustand vorliegt, sonst den Wert Ø. PAPER und INK haben den Wert der verlangten

Farbe, wie er auf der Tastatur angegeben wird (rot hat zum Beispiel den Wert 2). Programm P2.3 decodiert Attribute, d. h. wenn man den Wert eines Attributs vorgibt, druckt es die zugehörigen Farben von Hintergrund (PAPER) und Vordergrund (INK) usw. aus.

Programm P2.3

Ein Programm zur Decodierung eines Attributs.

```
 1Ø  REM Attribut-Decodierer
 2Ø  DATA "Black", "Blue", "Red", "Magenta", "Green", "Cyan",
     "Yellow", "White", "Bright", "Flash"
 3Ø  DIM c$ (8,7)
 4Ø  FOR i = 1 TO 8
 5Ø  READ c$ (i)
 6Ø  NEXT i
1ØØ  REM Attribut-Decodierer
11Ø  INPUT "Geben Sie eine Zahl zwischen Ø und 255 ein. Dieses
     Programm decodiert ihre Interpretation in der Attributsdatei"; n
12Ø  IF n <Ø OR n > 255 OR n <> INT n THEN BEEP .2,24: GO
     TO 11Ø
2ØØ  PRINT "Ink Farbe ist"; c$ (1 + n - 8 * INT (n/8))
21Ø  PRINT "Paper Farbe ist"; c$ (1 + INT (n/8) - 8 * INT (n/64))
22Ø  IF INT (n/64) = 1 OR INT (n/64) = 3 THEN PRINT "Zeichen ist
     BRIGHT"
23Ø  IF n > 127 THEN PRINT "Zeichen mit FLASH"
3ØØ  PRINT AT 6,Ø; "■■■■■■■■■■■■■■■■■
     ■■■■■■■■■■■■■■■"
31Ø  FOR i = 2272Ø TO 22751
32Ø  POKE i, n
33Ø  NEXT i
5ØØ  INPUT "Zum Wiederholen drücken Sie ENTER"; Z$
51Ø  CLS
52Ø  GO TO 11Ø
```

Der Druckerpuffer

Die 256 Speicherzellen im RAM, die auf die Attribute folgen, werden benutzt, um zeitweise eine unvollständige Zeile von Zeichen abzuspeichern, die später zum Drucker transportiert werden soll. Der Puffer ist notwendig, da in einem BASIC-Programm die Anweisung LPRINT vorkommen kann. Damit kann es einen Teil einer Zeile drucken, die mit einem Strichpunkt oder Komma endet, womit angezeigt wird, daß der Rest der Zeile erst noch folgen soll.

In manchen Fällen kann auch der Befehl TAB in ähnlicher Weise fungieren. Die unvollständige Zeile kann nicht sofort an den Drucker weitergegeben werden, weil dieser nur eine vollständige Zeile ausgeben kann. Währenddessen rückt er das Papier vor und bereitet sich damit auf die nächste Zeile vor. Deshalb wird der eine Teil der Zeile vorläufig im Druckerpuffer gespeichert bis das Programm mit LPRINT den anderen Teil ausdruckt.

Viele der Routinen in Teil B benutzen den Druckerpuffer, um Daten aus BASIC oder von der Tastatur auf die Routinen zu übertragen (Parameterübergabe). Der Puffer ist für diesen Zweck geeignet, weil sein Speicherbereich festgelegt ist und weil der Benutzer ihn wahrscheinlich für keinen anderen Zweck verwenden will, wenn er eine Maschinencode-Routine aufruft.

Der BASIC-Programmbereich

Wenn ein Microdrive an den Spectrum angeschlossen ist, muß der Anfang des BASIC-Programmbereichs festgestellt werden durch Abfrage der Systemvariablen PROG, die in 23635 steht. Wenn kein Micordrive angeschlossen ist, beginnt der Bereich bei 23755. In den folgenden Bemerkungen wird vorausgesetzt, daß kein Microdrive angeschlossen ist.

Wie man in Abbidlung F2.1 sieht, druckt das Vierzeilenprogramm P2.4 den Inhalt der 18 Speicherplätze am Beginn des Programmbereichs. Diese 18 Zellen werden benutzt, um die erste Zeile zu speichern, d. h.:

```
10  REM Peek program
```

Man kann viel über die Codierungsmethoden von Programmen lernen, wenn man Abbildung F2.1 genau studiert.

Programm P2.4

Ein Programm, um den Inhalt der ersten 18 Speicherzellen in Programmbereich auszudrucken.

```
10  REM Peek program
20  FOR i = 23755 TO 23772
30  PRINT i, PEEK i
40  NEXT i
```

23755	Ø
23756	1Ø
23757	14
23758	Ø
23759	234
2376Ø	8Ø
23761	1Ø1
23762	1Ø1
23763	1Ø7
23764	32
23765	112
23766	114
23767	111
23768	1Ø3
23769	114
2377Ø	97
23771	1Ø9
23772	13

Abbildung F2.1
Die Form, in der die Zeile 10 REM Peek program im Programmbereich gespeichert wird.

Die Zeilennummer, nämlich 1Ø, wird in den ersten beiden Zellen untergebracht in der Form:

Zeilennummer = 256 * PEEK erste Adresse + PEEK zweite Adresse

Man beachte, daß die Konvention des Z8ØA, den Inhalt der zweiten Adresse mit 256 zu multiplizieren und zum Inhalt der ersten Adresse zu addieren, nicht angewendet wird.

Die Konvention wird jedoch bei den nächsten beiden Speicherzellen, 23757 und 23758, benutzt, die zusammen die Länge der restlichen, in Zelle 23759 beginnenden Zeile abspeichern. Die in diesem Fall gespeicherte Zahl ist

14 + 256 * Ø = 14

Folglich beginnt die nächste Zeile in Zelle

23759 + 14 = 23773

In Zelle 23759 ist die Zahl 234 gespeichert. Dies ist der Zeichencode von REM. Die nächsten zwölf Zellen enthalten die Zeichencodes der elf Buchstaben und des Leerzeichens im Titel:

Peek program

Schließlich enthält die Zelle 23772 die Zahl 13, welcher der Code für das ENTER-Zeichen ist; dies zeigt an, daß die Zeile zu Ende ist. Tabelle 2.2 faßt die Methode zusammen, mit der Programme im Programmbereich codiert werden.

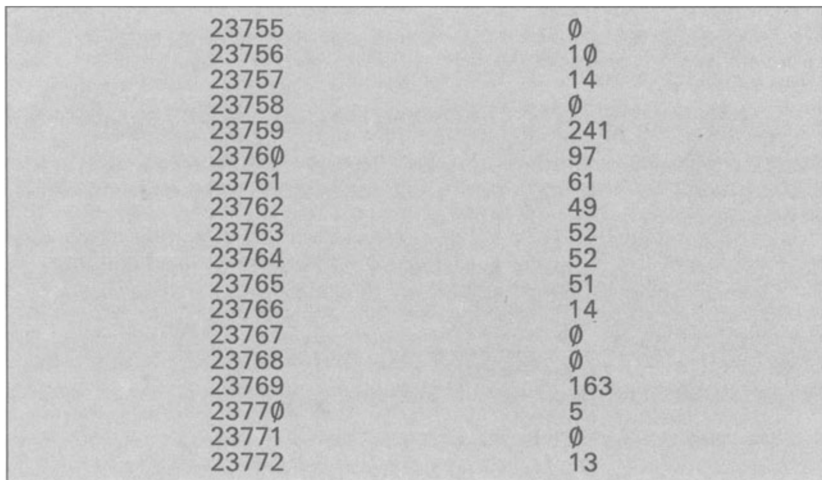
Tabelle 2.2
Die verwendete Codierungsmethode von Programmzeilen.

Speicherzellen	Inhalt
1 und 2	Zeilennummer in umgekehrter Reihenfolge zur Z8ØA Konvention gespeichert
3 und 4	Länge der Zeile ohne die ersten vier Zellen
5	der Code des Befehls
letzte	das Zeichen ENTER, die Zahl 13

Auf der Tabelle fehlt eine Beschreibung der Methode, die benützt wird, um im Programm vorkommende Werte zu speichern. Diese Methode kann folgendermaßen untersucht werden: man ersetzt in Programm P2.4 Zeile 1Ø durch:

1Ø LET a = 1443

Abbildung 2.2 zeigt das Resultat, wenn man das Programm in dieser Form mit RUN fährt.



23755	Ø
23756	1Ø
23757	14
23758	Ø
23759	241
2376Ø	97
23761	61
23762	49
23763	52
23764	52
23765	51
23766	14
23767	Ø
23768	Ø
23769	163
2377Ø	5
23771	Ø
23772	13

Abbildung 2.2
Die Form, in der die Zeile 10 LET a = 1443 im Programmbereich gespeichert wird.

Die Zellen 23755 bis 23758 enthalten dieselben Werte wie vorher. Darauf folgen die Codes für LET, a, = und die vier Ziffern nacheinander, die zusammen die Zahl 1443 ergeben. Der nächste Punkt ist die Zahl 14 in Speicherzelle

23766. Dies ist der Zeichencode, der anzeigt, daß die folgenden fünf Speicherzellen die Zahl 1443 in numerischer Form enthalten. Die Zeile endet in Zelle 23772 mit dem ENTER-Zeichen wie zuvor.

Zahlendarstellung mit fünf Bytes

Fünf Speicherplätze werden benutzt, um jede Zahl, die in einem BASIC-Programm auftaucht (außer den Zeilennummern, wie wir schon gesehen haben), zu speichern. Ganze Zahlen zwischen −65535 und 65535 werden in einer Art gespeichert, die mit der Konvention des Z8ØA übereinstimmt. Für diese Zahlen enthalten die ersten beiden Speicherzellen und die letzte Zelle eine Null; die dritte und vierte Zelle enthalten die Zahl in der Form

Zahl = PEEK dritte Zelle + 256 * PEEK vierte Zelle

So wird zum Beispiel 16553 in fünf Zellen folgendermaßen gespeichert:

Ø   Ø   169   64   Ø

Denn 169 + 256 * 64 = 16553.

Zahlen, die keine ganzen Zahlen sind, werden in der Gleitkommadarstellung gespeichert, und zwar mit dem Exponenten in der ersten Zelle und der Mantisse in den folgenden vier Zellen, d. h.
Zahl = Mantisse * 2 ↑ Exponent

Die erste Zelle der Mantisse wird auch benutzt, um das Vorzeichen der Zahl zu bestimmen. Wenn diese Zelle eine Zahl zwischen Ø und 127 enthält, ist die Zahl positiv, andernfalls negativ.

Programm P2.5 kann verwendet werden, um eine Zahl, die keine ganze Zahl ist, aus den Werten ihrer fünf Komponenten zu rekonstruieren.

Programm P2.5

Dieses Programm rekonstruiert eine Zahl, die keine ganze Zahl ist, aus den Werten ihrer fünf Komponenten.

```
1Ø  PRINT "Geben Sie den Exponenten und die vier Bytes der Mantisse
    ein. Alle Eingaben müssen zwischen Ø und 255 einschließlich lie-
    gen."
2Ø  INPUT e, a, b, c, d
3Ø  PRINT , , "Exponent = "; e
4Ø  PRINT "Mantisse = ", a,, b,, c,, d
5Ø  PRINT , , "Die Zahl = "; (2 * (a < 128) − 1) * 2 ↑ (e − 16Ø) * (((256
    * (a + 128 * (a < 128)) + b) * 256 + c) * 256 + d)
```

Der Variablenbereich

Der Variablenbereich beginnt bei dem Speicherplatz, der in der Systemvariablen VARS gespeichert ist. Diese wiederum ist in 23627 abgelegt. Bei jeder Deklaration einer neuen Variablen, entweder in einem Programm oder von der Tastatur, wird für sie angemessen viel Platz in diesem Bereich bereitgestellt.

Alle Variablennamen müssen mit einem Buchstaben beginnen, wobei Großbuchstaben wie Kleinbuchstaben behandelt werden ("Umkodierung"). Diese Einschränkungen ermöglichen es dem Spectrum, den Zeichencode des vordersten Buchstabens jeder Variablen geschickt zu handhaben; und zwar so, daß er die sechs erlaubten Variablentypen dadurch unterscheiden kann, daß er den Zahlenbereich ansieht, in dem der Code liegt. Alle numerischen Variablen mit einbuchstabigem Namen haben zum Beispiel einen Code, der zwischen 97 und 122 liegt; der Buchstabe a wird durch 97, b durch 98, c durch 99 usw. codiert. Gleichermaßen haben numerische Arrays Codes zwischen 129 und 153; a wird durch 129, b durch 13Ø, c durch 131 usw. codiert. Die Codebereiche sind in Tabelle 2.3 zusammengefaßt. Tabelle 2.3 gibt außerdem die Länge jedes Variablentyps an.

Tabelle 2.3
Variablen, der Bereich von Zeichencodes und die Variablenlängen

Variablentyp	Zeichencode-bereich	Länge im Variablenbereich
numerisch (Name mit einem Zeichen)	97 bis 122	6
numerisch (Name mit mehreren Zeichen)	161 bis 186	5 + Länge des Namens
numerisches Array	129 bis 154	4 + 2*Zahl der Dimensionen + 5*Gesamtanzahl der Elemente
Laufvariable einer FOR-NEXT-Schleife	225 bis 25Ø	18
String	65 bis 9Ø	3 + Länge des Strings
Zeichen-Array	193 bis 218	4 + 2*Zahl der Dimensionen + Gesamtanzahl der Elemente

ROM-Routinen

Einige der Routinen im Teil B verwenden Routinen im ROM wie folgt:

rst 16
Drucken des Akkumulatorinhalts.

call 3976
Einfügen des im Akkumulator gespeicherten Zeichens in diejenige Adresse im RAM, die im Registerpaar hl abgelegt ist.

call 421Ø
Streichen eines Zeichens in derjenigen Adresse im RAM, die in Registerpaar hl abgelegt ist.

call 6326
Setzen des Nullflags und sechsmaliges Inkrementieren des Registerpaars hl, wenn der Akkumulator das NUMBER-Zeichen (14) enthält.

call 6510
Zurückstellen der RAM-Adresse derjenigen Zeile, deren Zeilennummer an die Routine in hl gegeben wurde, in das Registerpaar hl.

Z8ØA-Maschinensprache

Dieses Kapitel beginnt damit, einige wichtige Worte wie *Bit, Byte, Adresse* und *Register* zu erklären. Diese Begriffe werden im weiteren Verlauf des Buches vorausgesetzt. Dann werden die Anzahl und Varianten der Z8ØA-Register untersucht. Dabei werden einige Befehle als Beispiele berücksichtigt. Schließlich wird eine Zusammenfassung des Befehlssatzes gegeben.

Vielleicht ist der schwierigste Aspekt für den Neuling in der Maschinencode-Programmierung, daß er eine Vielzahl von neuen Wörtern und Begriffen in sich aufnehmen muß. Bevor wir in den Hauptteil des Kapitels einsteigen, wollen wir daher einen Befehl untersuchen als Beispiel dafür, was noch vor uns liegt. Man betrachte den folgenden zusammengesetzten Befehl, den man in vielen Routinen in Teil B finden kann:

ld hl, (23627)

Diese Anweisung wird gelesen als *lade* das *Registerpaar hl* mit den *Bytes*, die in den *Adressen* 23627 und 23628 stehen. Jedes der kursiv gedruckten Worte wird in diesem Kapitel ausführlicher erklärt.

Der Befehl wird in Form von drei Dezimalzahlen, nämlich 42, 75, 92, übertragen. Die erste Zahl bedeutet

ld hl, (□□□)

d. h. lade das Registerpaar hl mit dem Inhalt von zwei aufeinanderfolgenden Speicheradressen. Die betreffenden Adressen werden von der zweiten und dritten Zahl durch folgende Rechnung festgelegt:

niederwertige Adresse = erste Zahl + 256 * zweite Zahl

höherwertige Adresse = niederwertige Adresse + 1

oder in diesem Fall

niederwertige Adresse = 75 + 256 * 92 = 23627

höherwertige Adresse = 23627 + 1 = 23628

Das Wort "lade" ist ein anderes Wort für "kopiere"; h und l kann man sich als zwei spezielle Speicherplätze im Z8ØA vorstellen, die Zahlen speichern können. Daher bedeutet der ganze Befehl: kopiere den Inhalt von 23627 in Register l und den Inhalt von 23628 in Register h. Man beachte, daß die niederwertige Adresse die Quelle für l (von engl.: *l*ower address) und die höherwertige Adresse die Quelle für h (von engl.: *h*igher address) ist.

Bits

Ein *Bit* ist die grundlegende Einheit eines Computerspeichers, weil es sich nur in einem von insgesamt zwei Zuständen befinden kann. Die beiden Zustände kann man sich vorstellen als EIN oder AUS; WAHR oder FALSCH; JA oder

NEIN; HINAUF oder HINUNTER; MÄNNLICH oder WEIBLICH oder jedes andere Paar von logisch entgegengesetzten Zuständen. Der Mechanismus, nach dem ein Computerspeicher funktioniert, ist für uns nicht wirklich wichtig; man sollte aber wissen, daß im Spectrum der Zustand eines Bits festgehalten wird, indem ein mikroskopischer Halbleiterschalter entweder EINgeschaltet oder AUSgeschaltet wird, je nach der Situation.

Die übliche Bezeichnung ist es, den einen Zustand als den NULL-Zustand anzusehen und den anderen als den EINS-Zustand. Man betrachtet ein Bit als "gesetzt", wenn es in dem Zustand ist, der EINS entspricht, andernfalls als "nicht gesetzt" oder "zurückgesetzt". Diese Bezeichnung ermöglicht uns, über ein gegebenes Bitmuster unter Verwendung der dazu äquivalenten Binärzahl zu sprechen. Durch die Umwandlung dieser Binärzahl in eine Dezimalzahl kann jedem Bitmuster eine eindeutige positive ganze Dezimalzahl zugeordnet werden.

Man betrachte zum Beispiel 8 Bits, wobei die rechten vier Bits gesetzt sind und die linken vier nicht. Solch ein Bitmuster wird in Tabelle 3.1 veranschaulicht.

Tabelle 3.1
Eine Gruppe von 8 Bits, wobei die vier linken Bits nicht gesetzt und die vier rechten gesetzt sind.

Schalter	Aus	Aus	Aus	Aus	Ein	Ein	Ein	Ein
Gesetzt	nein	nein	nein	nein	ja	ja	ja	ja
Binärmuster	Ø	Ø	Ø	Ø	1	1	1	1
Bitnummer	7	6	5	4	3	2	1	Ø

Das Binärmuster kann in eine Dezimalzahl umgerechnet werden, wenn man sich daran erinnert, daß in einer Binärzahl die am weitesten rechts stehende Spalte die Einerspalte, die links daneben stehende Spalte die Zweierspalte ist; die wiederum nächste links davon stehende Spalte ist die Viererspalte usw. Man verdoppelt bei jedem Schritt nach links. Die zu ØØØØ1111 äquivalente Dezimalzahl ist deshalb

Ø * 128 + Ø * 64 + Ø * 32 + Ø * 16 + 1 * 8 + 1 * 4 + 1 * 2 + 1 * 1 = 15

denn es steht jeweils 1 in der Einer-, Zweier-, Vierer- und Achterspalte und Ø in den restlichen Spalten.

Offensichtlich ist es unbequem, ein Bit als "das am weitesten rechts stehende" oder als "das zweite von links" zu bezeichnen. Deshalb wird die Konvention eingeführt, daß die Bits von rechts her numeriert werden, und zwar beginnt man mit der Nummer Null. Es ist durchaus nicht zufällig, daß bei dieser Konvention die Bitnummer gerade die Potenz ist, in die 2 erhoben werden muß, damit man den Wert der Spalte erhält. Das heißt: $2^{\text{Bitnummer}}$ = Wert der Spalte. Bit 3 erscheint beispielsweise in der Achterspalte, und $2^3 = 8$.

Bytes

Der Z8ØA Mikroprozessor, das Herz des ZX Spectrum, führt immer Operationen mit jeweils 8 Bits aus. (Der Begriff "Operation" überdeckt alle verschiedenen Aufgaben, welche in den Befehlssatz eingearbeitet sind, wie die Addition, die Subtraktion, die Rotation, die logische Verknüpfung AND usw. Die Form dieser Anweisungen wird weiter hinten in diesem Kapitel ausführlich erklärt). So werden Bits normalerweise in Achtergruppen geschlossen gehandhabt, obwohl sie einzeln die grundlegenden Einheiten von Computerspeichern sind. Eine solche Achtergruppe von Bits wird *Byte* (Aussprache: beit) genannt.

Jedes Byte im RAM kann eine natürliche Zahl zwischen Ø und 255 einschließlich aufnehmen, indem es die 8 Bits setzt oder nicht setzt, entsprechend der Binärdarstellung der Zahl. Das Byte aus Tabelle 3.1 enthält beispielsweise die Dezimalzahl 15.

Es gibt 16384 Bytes im Nur-Lese-Speicher (ROM) des ZX Spectrum. Durch den Inhalt dieser Bytes zusammen mit dem elektronischen Aufbau erhält der Computer seinen besonderen Charakter. Der Inhalt wird bei der Herstellung in das ROM eingeprägt und kann danach nicht geändert werden. Aus diesem Grund heißt der Speicher Nur-Lese-Speicher: der Inhalt kann gelesen, aber nicht überschrieben werden.

Der nicht erweiterte Spectrum besitzt weitere 16384 Bytes für den Speicher mit wahlfreiem Zugriff (englisch: random access memory). Der englische Begriff "random access" ("zufälliger Zugriff") ist vielleicht eine etwas falsche Bezeichnung. Er bedeutet nicht, daß der Speicher in zufälliger Weise benutzt wird, sondern, daß jedes Byte zu jeder Zeit sofort erreicht werden kann (d. h. man hat zu ihm "Zugriff"). Diese Einrichtung hebt sich von derjenigen eines Speichers mit sequentiellem Zugriff (englisch: sequential access memory) ab wie z. B. des Bands einer Kassette. In diesem Fall muß man nämlich entlang dem Speichermedium fahren, bis die gewünschte Stelle erreicht ist.

Für den Neuling scheint die Zahl 16384 nicht besonders praktisch als Anzahl von Bytes. Tatsächlich ist sie aber sehr praktisch, da $2^{14} = 16384$ (d. h. 16384 ist gleich 2, vierzehnmal mit sich selbst multipliziert). In der Computerwelt sind Zweierpotenzen "runde Zahlen", genauso wie im Alltag Zehnerpotenzen – Hunderter, Tausender, Millionen – "runde Zahlen" sind. Eine besonders wichtige "runde Zahl" ist 1Ø24 (= 2^{10}). 1Ø24 liegt ausreichend nahe an 1ØØØ, um als Repräsentant dafür die Benutzung des Buchstabens k zu rechtfertigen. (Im metrischen System wird k für tausend verwendet wie in Kilogramm – kg, Kilometer – km usw.) so schreibt man 1Ø24 auch als 1k und 16384 (= 16 x 1Ø24) als 16k.

Adressen

Ein Computer muß jeden Speicherplatz identifizieren können, damit er von der richtigen Zelle oder in die richtige Zelle kopieren kann. Deshalb besitzt jede Speicherzelle eine eindeutig bestimmte *Adresse*. Eine Adresse ist eine positive ganze Zahl, die größer oder gleich null ist.

Viele der Z8ØA-Befehle haben die Form "kopiere den Inhalt der folgenden Adresse in Register oder Registerpaar so-und-so". Die Anweisung

```
ld hl, (23627)
```

die am Anfang dieses Kapitels beschrieben wurde, hat diese Form. Die auf den Befehl folgende Adresse befindet sich in zwei Bytes. Deshalb ist die Anzahl der Zellen, zu denen der Prozessor eindeutig Zugriff hat, durch die Anzahl derjenigen Adressen begrenzt, die in zwei Bytes enthalten sein können. Diese Zahl stimmt überein mit der Zahl der verschiedenen Bitmuster, die von den 16 Bits angenommen werden können, aus denen die zwei Adressenbytes bestehen, d. h. $2^{16} = 65536$.
Eine 2-Byte-Adresse wird interpretiert als:

Adresse = erstes Byte + 256 * zweites Byte

Die beiden Bytes werden manchmal das niederwertige und das höherwertige Byte genannt. Die 2-Byte-Form von 16384 zum Beispiel (der Anfang des RAM im Spectrum) ist niederwertiges Byte = Ø; höherwertiges Byte = 64, denn

Ø + 256 * 64 = 16384.

Die Register des Z8ØA

Ein Computer ändert bei Ausführung eines Programmes den Inhalt eines Speichers nicht direkt, sondern kopiert den Inhalt eines Speicherplatzes in ein *Register*. Danach arbeitet er mit dem Inhalt dieses Registers weiter. In Maschinensprache haben Register eine den Variablen in BASIC vergleichbare Funktion, denn sie werden verwendet, um Zahlen zu speichern, und sie können verwendet werden, um Entscheidungen zu steuern. Sie unterscheiden sich von BASIC-Variablen dadurch, daß ihre Anzahl begrenzt ist und daß sie im Prozessor selbst und nicht im RAM existieren. Außerdem enthalten sie nur ein Byte oder im Fall eines Registerpaares zwei Bytes.

Der Z8ØA ist ein leistungsstarker Mikroprozessor, weil er mehrere Register besitzt und deshalb gleichzeitig mehrere Zahlen enthalten kann. Dadurch verringert sich die Notwendigkeit, zeitaufwendige Transfers zwischen dem Prozessor und dem Speicher vorzunehmen. Die meisten Register haben ein oder sogar mehrere Spezialmerkmale.

Das Akkumulator-Register a

Der Akkumulator ist das wichtigste Register, weil die meisten arithmetischen Anweisungen, z. B. die Addition, und die logischen Anweisungen, z. B. das logische OR (oder), mit dem Inhalt dieses Registers arbeiten. In der Tat verdient es diesen Namen, weil sich die Ergebnisse von mehreren aufeinanderfolgenden Operationen im Register a ansammeln (von engl.: to accumulate).

Einige der Anweisungen, die sich auf den Akkumulator beziehen, verwenden ein zweites Register oder eine Speicheradresse als Datenquelle. Zum Beispiel weist "add a, b" den Prozessor an, den Inhalt von Register b zu dem von Register a zu addieren und das Ergebnis in Register a abzulegen.

Das Flag-Register f

Die meisten Register kommen nur paarweise vor; das heißt, daß sich manche Anweisungen auf zwei Register beziehen. Das Flag-Register oder Register f

und das Register a werden in diesem Sinne als Paar betrachtet, obwohl die Verbindung weit hergeholt ist, denn sie bezieht sich nur auf die Anweisungen "push", "pop" und "exchange".

Das Register f unterscheidet sich ziemlich stark von allen anderen Registern, weil die acht einzelnen Bits im Register als sogenannte Flags (Flaggen) benutzt werden. Solche Flags zeichnen die Folge der Programmausführung auf und steuern sie. Jedes Flag zeigt an, ob eines von zwei logisch entgegengesetzten Ereignissen eingetreten ist. Beispielsweise gibt das Nullflag (engl.: zero flag) an, ob das Ergebnis der letzten Addition, Subtraktion usw. null war. Für die meisten Benutzer sind nur vier der acht Flags interessant. Ihre Merkmale sind in Tabelle 3.2 zusammengefaßt.

Tabelle 3.2
Die vier Flags, welche die meisten Operationen des Z8ØA steuern.

Flag	Mnemonic	Mnemonic, wenn nicht gesetzt	Verwendung
Vorzeichen	M	P	Gesetzt, wenn letztes Ergebnis negativ (vgl.: *m*inus) ist
Null	Z	NZ	Gesetzt, wenn letztes Ergebnis null ist oder wenn bei Vergleich Übereinstimmung vorliegt.
Übertrag	C	NC	Gesetzt, wenn letztes Ergebnis zu groß für ein Byte ist (oder für zwei Bytes bei Operationen auf Registerpaaren).
Parität/ Überlauf	PE	PO	Parität: gesetzt, wenn letztes Ergebnis gerade ist. Überlauf: gesetzt, wenn eine Operation Bit 7 verändert infolge Überlaufs anderer Bits.

Das Vorzeichenflag (englisch: sign flag) ist das einfachste Flag. Nach Konvention enthält Bit 7 eines Bytes das Vorzeichen, wenn dieses Byte als Zahl mit Vorzeichen aufgefaßt wird. Das Bit wird gesetzt, wenn die Zahl negativ ist, sonst nicht. Das Vorzeichenflag zeigt das Vorzeichen des letzten Ergebnisses an.

Das Nullflag wird gesetzt, wenn das Ergebnis der letzten Operation null ist. Es wird auch bei Vergleichsbefehlen verwendet. Diese sind eigentlich Subtraktionsanweisungen, deren Ergebnis nicht weiter berücksichtigt wird.

Das Übertragsflag (englisch: carry flag) zeichnet den Übertrag auf, wenn das Ergebnis einer Addition für das Register zu groß ist und wenn bei einer Subtraktion "geborgt" wird. Es gibt auch einige Rotationsbefehle, in denen die Bits in einem Register nach links oder rechts geschoben werden, wobei Bit 7 und Bit Ø vom oder zum Übertragsflag geschoben werden.

Das Paritäts-/ Überlauf-Flag (englisch: parity/overflow flag) hat eigentlich die Funktion von zwei Flags. Zum einen wird es von arithmetischen Befehlen als Überlauf-Flag verwendet. Es zeigt dann an, ob Bit 7 von einem aus Bit 6 stammenden Übertrag oder "Borgen" beeinflußt wurde. Daher wird es benutzt, um festzustellen, ob das Vorzeichenbit fälschlicherweise verändert wurde. Andererseits verwenden logische Anweisungen dasselbe Flag, um die Parität des Resultats anzugeben. (Die Parität einer Binärzahl ist die Anzahl der [auf eins] gesetzten Bits. Wenn diese Anzahl gerade ist, nennt man die Parität gerade; ist sie ungerade, nennt man die Parität ungerade). Das Flag wird gesetzt, wenn die Parität eines Resultats gerade ist.

Die Wirkung einiger Anweisungen hängt vom aktuellen Zustand einiger Flags ab. Zum Beispiel läßt die Anweisung

jr z, d

den Z8ØA die nächsten d Anweisungen überspringen, wenn das Nullflag gesetzt ist. Andernfalls führt der Prozessor wie üblich die nächstfolgende Anweisung aus. Das Flag-Register ist also wichtig, weil es dem Prozessor ermöglicht, Entscheidungen zu treffen und zu einem anderen Teil des Programms zu springen.

Die Zählregister b und c

Das Register b und zu einem gewissen Grad das Register c, die zusammen ein Registerpaar bilden, stehen für viele Zwecke zur Verfügung. Ihre wichtigste Verwendung ist diejenige als Zähler. Wir haben schon gesehen, wie die Abfolge eines Programms durch das Nullflag in der Anweisung jr z, d gesteuert werden kann. Eine andere Anweisung

djnz d

verwendet auch das Nullflag für Schleifen. Und zwar können damit Schleifen in Maschinencode konstruiert werden, die b als Zähler verwenden, ähnlich wie FOR-NEXT-Schleifen in BASIC.

Auf die obige Anweisung hin dekrementiert der Z8ØA den Inhalt des Registers b, d. h. er verringert den Inhalt um eins. Wenn das Ergebnis null ist, wird die nächstfolgende Anweisung ausgeführt. Ist das Ergebnis nicht null, überspringt die Routine d Anweisungen. Wenn der Programmierer einen negativen Wert für d verwendet, springt der Prozessor an eine frühere Stelle im Programm. Vorausgesetzt, es gibt keine weiteren Verzweigungen, so trifft der Prozessor schließlich wieder auf dieselbe Anweisung. So kann ein Codeteil beliebig oft ausgeführt werden, indem das Register b anfänglich mit einem geeigneten Wert geladen wird und die Distanz d (oft auch Offset genannt, englisch: displacement) passend gewählt wird.

Das Register b enthält nur ein Byte und kann deshalb auf jede Zahl zwischen Ø und maximal 255 gesetzt werden. Deshalb kann der Spectrum mit diesem Mechanismus denselben Codeteil höchstens 255mal durchlaufen.

Es gibt keine vergleichbaren Möglichkeiten für mehr als 255 Durchgänge durch eine Schleife, dafür aber eine begrenzte Zahl von sehr leistungsstarken Anweisungen, die alle 16 Bits des Registerpaars bc als Zähler bis 65535 verwenden. Als Beispiel dient die Anweisung:

cpdr

Wenn der Z8∅A auf sie trifft, so

1) dekrementiert er bc um eins,
2) dekrementiert er den Inhalt von hl (hl ist ein anderes Registerpaar) – siehe unten,
3) vergleicht er den Inhalt des Akkumulators a mit dem Inhalt des Speicherplatzes, dessen Adresse in hl abgelegt ist.

Der Prozessor wiederholt diese Vorgänge bis der Inhalt von a und der Speicherplatzinhalt übereinstimmen oder bis bc = ∅. So kann diese Anweisung verwendet werden, um den Speicher nach einer Adresse zu durchsuchen, in der eine bestimmte Zahl abgelegt ist.

Die Adreßregister de und hl

Die Register d und e haben keine bestimmte Funktion und werden in der Hauptsache als vorübergehende, schnell erreichbare Speicher verwendet. Sie können auch gemeinsam benutzt werden, um eine gerade interessierende Speicherplatzadresse aufzubewahren.

Die Hauptfunktion der Register h und l ist es, gemeinsam eine Speicherplatzadresse zu speichern. Wir haben schon gesehen, wie gewisse leistungsstarke Anweisungen hl zu diesem Zweck gebrauchen. Dabei steht h für höherwertiges Byte, l steht für niederwertiges Byte, und die Adresse wird in folgender Form abgelegt:

Adresse = 256 * h + l

Damit ergeben sich maximal 65536 eindeutige Adressen (d. h. von ∅ bis einschließlich 65535).

Die Indexregister ix und iy

Die Register ix und iy sind 16-Bit-Register und können nur als solche verwendet werden, im Gegensatz zu den Registern bc, de und hl, denen wir bisher begegnet sind. Diese kann man nämlich paarweise als 16-Bit-Register oder einzeln als 8-Bit-Register benutzen. ix und iy werden im allgemeinen ähnlich wie das Registerpaar hl eingesetzt; allerdings benötigen die Anweisungen, die mit ihnen arbeiten, ein Speicherbyte mehr als die entsprechenden hl-Anweisungen. Zum Beispiel ist

```
add hl, bc
```

eine 1-Byte-Anweisung, die den Z8∅A veranlaßt, die Inhalte der Registerpaare hl und bc zu addieren und das Ergebnis in hl abzulegen. Dieselbe Anweisung, die ix verwendet, d. h.

```
add ix, bc
```

ist eine 2-Byte-Anweisung. Die Register ix und iy haben eine weitere Eigenschaft, die hl nicht zur Verfügung steht. Sie können nämlich mit einer Distanz d

benutzt werden. Dies bedeutet, daß eine Anweisung, die sich auf (ix + d) bezieht, nicht den Speicherplatz verwendet, dessen Adresse in ix steht, sondern denjenigen, dessen Adresse die Summe des Wertes in ix und der Zahl d ist. Wegen dieser Eigenschaft werden ix und iy Indexregister genannt.

Der Stapelzeiger sp

Der Stapel ist ein Bereich nahe am oberen Ende des RAM oder ganz oben. Er wird als kurzzeitiger Speicher von Registerpaarinhalten verwendet. Der Stapel wächst nach unten im RAM, wenn auf ihm etwas abgelegt wird, und er "schrumpft" nach oben, wenn er geleert wird. Das untere Ende des Stapels ist festgelegt, und im ZX Spectrum liegt es unmittelbar unter der Speicherzelle, auf welche die Systemvariable RAMTOP zeigt. Das obere Ende des Stapels liegt *unter* dem unteren Ende des Stapels, weil er nach unten wächst und nach oben "schrumpft". Die Adresse der Zelle, die gegenwärtig am oberen Ende des Stapels liegt, befindet sich in Register sp (von englisch: *s*tack *p*ointer – deutsch: Stapelzeiger).

Transfers vom und zum Stapel werden mittels pop- und push-Anweisungen ausgeführt. Zum Beispiel läßt

```
push hl
```

den Prozessor

1) sp dekrementieren,
2) den Inhalt von h in die durch sp angezeigte Stelle kopieren,
3) sp noch einmal dekrementieren, und schließlich
4) den Inhalt von l in die durch sp angezeigte Stelle kopieren.

Die Anweisung pop arbeitet genauso, allerdings in umgekehrter Reihenfolge. Auf diese Weise wird das zuletzt auf dem Stapel abgespeicherte Wertepaar auch wieder zuerst mit pop ausgelesen. Dies ergibt eine einfache und bequeme Methode zur kurzzeitigen Speicherung von Registerinhalten, etwa während eine Subroutine aufgerufen wird. Es gibt keine Schwierigkeiten, vorausgesetzt, die Registerpaare werden in umgekehrter Reihenfolge entfernt, wie sie abgelegt wurden.

Der Befehlszähler pc

Der Befehlszähler pc (von englisch: *p*rogram *c*ounter) ist ein sehr wichtiges 16-Bit-Register, weil es die Speicheradresse des Befehls enthält, der als nächster ausgeführt werden soll.

Die normale Abfolge bei Ausführung eines Befehls ist die folgende:

1) Der Inhalt der durch pc bezeichneten Zelle wird in ein Spezialregister innerhalb des Prozessors kopiert.
2) Falls der Befehl in mehreren Bytes enthalten ist, wird pc inkrementiert und der Inhalt der nächsten Zelle in ein zweites Spezialregister kopiert.
3) Der Befehlszähler pc wird inkrementiert, damit er auf den nächsten auszuführenden Befehl zeigt.
4) Der Befehl, der gerade hineinkopiert wurde, wird ausgeführt.

Eine Sprunganweisung wie djnz d oder jr z, d ändert den normalen Ablauf dadurch, daß pc während Schritt 4) verändert wird. Man beachte, daß diese Änderung nach der Inkrementierung von pc auftritt. Deshalb sollte der Wert einer Distanz d immer relativ zur Position derjenigen Anweisung berechnet werden, die auf die Anweisung mit der Distanz folgt.

Die Wechselregister af', bc', de', hl'

Der Z8ØA besitzt Duplikate der Register a, b, c, d, e, h und l. Diese sind durch einen Strich gekennzeichnet: a' ist beispielsweise das Duplikat von Register a. Es gibt keine Anweisungen, die direkt mit diesen Zweitregistern arbeiten, sondern Austauschbefehle, die zwei oder mehr Register gegen ihre Duplikate auswechseln. Dabei sind dann nur noch die Duplikate in Gebrauch.

Austauschbefehle werden sehr schnell ausgeführt, viel schneller als zum Beispiel push- und pop-Anweisungen. Der eigentliche Inhalt wird nicht wirklich von einem Register in ein anderes kopiert, sondern bestimmte interne Schalter werden geändert, so daß das Strichregister von den folgenden Befehlen verwendet wird, während das ursprüngliche Register ruht.

Über den Befehlssatz

Der Befehlssatz des Z8ØA enthält mehr als 6ØØ Elemente. Sie sind in Anhang A aufgelistet. Da es nur 256 verschiedene Anordnungen von 8 Bits gibt (denn $2^8 = 256$), können nur weniger als die Hälfte der Befehle durch ein Byte dargestellt werden. Die übrigen Befehle benötigen zur Darstellung zwei oder sogar drei Bytes. Das erste Byte eines 2-Byte-Befehls ist entweder 2Ø3, 221, 237 oder 253 (CB, DD, ED oder FD in Hexadezimaldarstellung). Die ersten 2 Bytes eines 3-Byte-Befehls sind entweder 221, 2Ø3 oder 253, 2Ø3 (DD, CB oder FD, CB in Hexadezimaldarstellung).

Auf einige Befehle folgt eine 1-Byte-Distanz d, eine 1-Byte-Ziffer n oder eine 2-Byte-Ziffer oder -adresse nn, auf die sich der Befehl bezieht. Auf diese Art kann ein einziger Befehl bis zu insgesamt vier Bytes belegen. Beispielsweise braucht die schon erwähnte Anweisung

jr nz, d

ein Byte für den Befehl selbst (32 in Dezimal-, 2Ø in Hexadezimalschreibweise) und ein zweites Byte für die Distanz d.

In diesem Kapitel beziehen wir uns auf alle Befehle durch ihr Assemblersprachen-Mnemonic oder ihren Opcode (d. h. Operationscode). Die Mnemonics sind abkürzende Beschreibungen jedes Befehls und nur als Bequemlichkeit für den menschlichen Benutzer gedacht. Der Spectrum erkennt keine Mnemonics, außer über das Hilfsmittel eines Assemblerprogramms.

Einige Konventionen, die befolgt werden, seien hier erwähnt:

1) Man bezieht sich auf einzelne Register durch ihren Buchstaben, zum Beispiel b. Registerpaare werden in alphabetischer Reihenfolge benannt, beispielsweise bc.
2) Eine Distanz d wird als positiv angesehen, wenn sie zwischen Ø und 127 liegt; als negativ, wenn sie zwischen 128 und 255 liegt. Größere oder

kleinere Zahlen sind nicht erlaubt. Der negative Wert wird berechnet, indem man d von 256 abzieht. Zum Beispiel verursacht die unbedingte, relative Sprunganweisung

jr d

einen Sprung vorwärts um 8 Bytes, wenn d = 8, und einen Sprung rückwärts um 8 Bytes, wenn d = 248 (= 256 − 8). Man erinnere sich bei der Berechnung einer Distanz daran, daß ausgehend von der Adresse des ersten Bytes unmittelbar nach der Anweisung gesprungen wird.

3) Eine einfache Byteziffer n liegt zwischen Ø und 255 einschließlich.
4) Eine 2-Byte-Ziffer oder eine Adresse wird durch nn dargestellt und liegt zwischen Ø und 65535 einschließlich. Der Wert wird berechnet, indem man das erste n zum Produkt des zweiten mit 256 addiert.
5) nn in Klammern, d. h. (nn), bedeutet "der Inhalt der Speicherzelle mit Adresse nn", während nn ohne Klammern "die Zahl" nn bedeutet. So heißt

   ld hl, (23627)

   daß das Registerpaar hl mit dem Inhalt der Zellen 23627 und 23628 geladen werden soll. Dagegen heißt

   ld hl, 23627

   daß hl mit der Zahl 23627 geladen werden soll. In ähnlicher Weise steht (hl) für den "Inhalt der Zelle mit der Adresse, die in hl abgelegt ist". Dagegen steht hl ohne Klammern für "die Zahl in hl".
6) Der Bestimmungsort des Ergebnisses eines Befehls wird immer zuerst angegeben. Zum Beispiel heißt

   add a, b

   "addiere den Inhalt von b zum Inhalt von a und lege das Ergebnis in a ab".

Glossar von Maschinencode-Befehlen

Dieser Abschnitt faßt den größten Teil des Z8ØA-Befehlssatzes zusammen. Einige speziellere Befehle, etwa beim Umgang mit Interrupts, wurden ausgelassen.

No Operation **nop**

Der No-Operation-Befehl ist der einfachste Befehl. Wie sein englischer Name sagt, tut der Prozessor nichts, wenn er auf ihn trifft. Er kann sehr nützlich sein bei der Fehlersuche in einer Routine, weil man ihn vorübergehend für einen verdächtigen Befehl einsetzen kann, ohne den Ablauf der übrigen Routine zu verändern. Er kann auch zum Schließen von Lücken verwendet werden, die bei kleinen Änderungen an schon vorhandenen Programmen entstehen. Außerdem kann er die Ausführung verzögern, besonders wenn er in eine geeignete Schleife eingebaut wird. Der Dezimalcode ist Ø.

Laden **ld**

Lade-(englisch: *load*)-Anweisungen werden verwendet, um ein oder zwei Bytes zwischen verschiedenen Registern oder zwischen Registern und dem Speicher zu bewegen. Es gibt mehr als hundert verschiedene Lade-Anweisungen, also mehr als in jeder anderen Klasse von Befehlen. Sie zerfallen in acht Gruppen:

1) Von 8-Bit-Register in Register.
   Der Inhalt aus jedem der Register a, b, c, d, e, h oder l kann wiederum in jedes dieser Register kopiert werden.
2) Von 8-Bit-Speicher in Register.
   (hl), (ix + d) oder (iy + d) können in jedes der Register a, b, c, d, e, h oder l kopiert werden. (bc), (de) oder (nn) können in a kopiert werden.
3) Von 8-Bit-Register in Speicher.
   a, b, c, d, e, h oder l können in (hl), (ix + d) oder (iy + d) kopiert werden. a kann in (bc), (de) oder (nn) kopiert werden.
4) Von Speicher mit unmittelbaren Daten in 8-Bit-Register oder Speicher.
   Eine unmittelbare Zahl ist eine Zahl, die aus dem Programm direkt ausgelesen wird und nicht aus einem Register oder einer anderen Speicheradresse. Eine Zahl n kann in a, b, c, d, e, h, l, (hl), (ix + d) oder (iy + d) geladen werden.
5) Von 16-Bit-Register in Register.
   Der Inhalt von hl, ix oder iy kann in sp geladen werden.
6) Von 16-Bit-Speicher in Register.
   (nn) kann in bc, de, hl, ix, iy oder sp kopiert werden.
7) Von 16-Bit-Register in Speicher.
   bc, de, hl, ix, iy oder sp können in (nn) kopiert werden.
8) Von 16-Bit-Speicher mit unmittelbaren Daten in 16-Bit-Register.
   nn kann in bc, de, hl, ix, iy oder sp geladen werden.

Push und pop **push, pop**

Eine Push-Anweisung kopiert den Inhalt eines gegebenen 16-Bit-Registers in den Stapel und dekrementiert den Stapelzeiger zweimal. Eine Pop-Anweisung arbeitet genau umgekehrt. Deshalb können die beiden Anweisungen verwendet werden, um Registerwerte zu speichern und sie an einer späteren Stelle des Programms wieder zurückzuladen. Die Registerpaare af, bc, de, hl, ix und iy können mit push auf dem Stapel abgelegt und mit pop vom Stapel zurückgeholt werden.

Exchange **ex**

Vertauschungen (englisch: *ex*changes) können zwischen hl und de, hl und (sp), ix und (sp), iy und (sp), af und af' und zwischen bcdehl und bcdehl' (ein einziger Befehl vertauscht alle sechs 8-Bit-Register) vorgenommen werden.

8-Bit-Addition und -Subtraktion **add, sub usw.**

a, b, c, d, e, h, l, (hl), n, (ix + d) und (iy + d) können zu Register a addiert oder von ihm subtrahiert werden, mit oder ohne Übertragsflag. Befehle mit Übertragsflag enden mit c (z. B. adc).

8-Bit-und, -oder, -entweder-oder **and, or, xor**

a, b, c, d, e, h, l, (hl), n, (ix + d) und (iy + d) können durch eine der drei logischen Operationen "und" (and), "einschließendes oder" (or) und "ausschließendes oder" (xor) mit dem Register a kombiniert werden. Dabei setzt "and" im Ergebnis jedes Bit, das in beiden Operanden gesetzt war; "or" setzt jedes Bit, das in einem der beiden oder in beiden Operanden gesetzt war; und "xor" setzt jedes Bit, das entweder im einen oder dem anderen Operanden, aber nicht in beiden, gesetzt war.

Vergleichen **cp**

Der Vergleich (englisch: *c*om*p*are)-Befehl ist wie der Befehl "sub", außer daß nur die Flags und nicht der Inhalt von a berührt werden. a, b, c, d, e, h, l, (hl), n, (ix + d) und (iy + d) können mit dem Akkumulator verglichen werden.

8-Bit-Inkrementierung und -Dekrementierung **inc, dec**

a, b, c, d, e, h, l, (hl), (ix + d) und (iy + d) können inkrementiert oder dekrementiert werden.

16-Bit-Inkrementierung und -Dekrementierung **inc, dec**

bc, de, hl, ix, iy und sp können inkrementiert oder dekrementiert werden.

16-Bit-Addition und -Subtraktion **add, sub usw.**

bc, de, hl, ix können mit oder ohne Übertrag zu hl addiert, oder nur mit Übertrag von hl subtrahiert werden. bc, de, sp und ix können ohne Übertrag zu ix addiert werden, bc, de, sp und iy können ohne Übertrag zu iy addiert werden.

Sprung, Aufruf und Rücksprung

Das Flag-Register f enthält ein Übertragsflag c, ein Paritätsflag p, das gesetzt wird, wenn ein Ergebnis gerade Parität hat, ein Vorzeichenflag s, das gesetzt wird, wenn ein Ergebnis negativ ist, ein Überlauf-Flag v, das bei Überlauf gesetzt wird, und ein Nullflag z, das bei dem Ergebnis "null" gesetzt wird. Diese Flags können verwendet werden, um Sprünge (englisch: jumps), Subroutinenaufrufe (englisch: calls) und Subroutinenrücksprünge (englisch: returns) zu steuern.

1) Sprung **jp oder jr**
Die folgenden Sprünge zur Adresse nn sind möglich: absoluter Sprung (jp); bedingter Sprung bei null (jp z) oder nicht null (jp nz); bedingter Sprung bei Übertrag (jp c) oder keinem Übertrag (jp nc); bedingter Sprung, wenn positiv (jp p) oder negativ (jp m); bedingter Sprung, wenn p/v = 1 (jp pe) oder p/v = Ø (jp po).
Die folgenden relativen Sprünge an eine Adresse d relativ zur aktuellen Position sind möglich, wenn d interpretiert wird als Zahl zwischen −128 und 127: absoluter relativer Sprung (jr); relativer bedingter Sprung bei null (jr z) oder nicht null (jr nz); relativer bedingter Sprung bei Übertrag (jr c) oder keinem Übertrag (jr nc).
Auch zu den Adressen, die in hl, ix oder iy abgelegt sind, können Sprünge gemacht werden (jp (hl), jp (ix), jp (iy)). Der Befehl djnz dekrementiert Register b und springt nach d, wenn b nicht null ist.

2) Aufruf **call**
Dieser Befehl hat eine ähnliche Funktion wie der BASIC-Befehl GOSUB. Wenn die Aufrufbedingung erfüllt ist, geht das Programm zum Befehl in Adresse nn über. Die folgenden Aufrufe können gemacht werden: absoluter Aufruf (call); bedingter Aufruf bei null (call z) oder nicht null (call nz); bedingter Aufruf bei Übertrag (call c) oder keinem Übertrag (call nc); bedingter Aufruf, wenn positiv (call p) oder wenn negativ (call m); bedingter Aufruf, wenn p/v = 1 (call pe) oder p/v = Ø (call po).

3) Rücksprung **ret**
Der Rücksprung-Befehl hat eine ähnliche Funktion wie der BASIC-Befehl RETURN. Rücksprungbedingungen, die jeweils den Aufrufbedingungen entsprechen, stehen zur Verfügung. Es kann auch von einem Interrupt oder einem nicht-maskierbaren Interrupt zurückgesprungen werden (reti und retn).

Bit-Befehle

Die acht Bits in jedem Register werden von rechts nach links mit Ø bis 7 numeriert. Die folgenden Operationen können jeweils mit den Registern a, b, c, d, e, h, l und (hl), (ix + d) und (iy + d) ausgeführt werden.

1) Bit-Test **bit**
Die Bit-Test-Anweisung setzt das Nullflag, wenn das benannte Bit nicht gesetzt ist und umgekehrt. Jedes Bit kann getestet werden.
2) Bit-Setzen **set**
Jedes Bit kann gesetzt werden.
3) Bit-Zurücksetzen **res**
Jedes Bit kann zurückgesetzt werden.
4) Links-Rotieren durch das Übertragsbit **rl**
Bit 7 wird in das Übertragsbit kopiert, das Übertragsbit wird in Bit Ø kopiert, und alle anderen Bits werden um einen Platz nach links verschoben.
5) Rechts-Rotieren durch das Übertragsbit **rr**
Bit Ø wird in das Übertragsbit kopiert, das Übertragsbit wird in Bit 7 kopiert, und alle anderen Bits werden um einen Platz nach rechts verschoben.

6) Zyklisches Links-Rotieren **rlc**
Bit 7 wird in das Übertragsbit und in Bit ∅ kopiert. Alle übrigen Bits werden um einen Platz nach links verschoben.

7) Zyklisches Rechts-Rotieren **rrc**
Bit ∅ wird in das Übertragsbit und in Bit 7 kopiert. Alle anderen Bits werden um einen Platz nach rechts verschoben.

8) Arithmetisches Links-Schieben **sla**
Alle Bits werden um einen Platz nach links verschoben. Bit 7 kommt ins Übertragsbit und der Wert ∅ kommt in Bit ∅.

9) Arithmetisches Rechts-Schieben **sra**
Alle Bits werden um einen Platz nach rechts verschoben. Bit ∅ kommt ins Übertragsbit und der Inhalt von Bit 7 wird in sich selbst kopiert.

1∅) Logisches Rechts-Schieben **srl**
Wie arithmetisches Rechts-Schieben, außer daß in Bit 7 der Wert ∅ kommt.

Links-Dezimal-Rotation **rld**

Die Akkumulatorbits ∅ bis 3 werden in die Bits ∅ bis 3 von (hl) kopiert; währenddessen werden die Bits ∅ bis 3 von (hl) in die Bits 4 bis 7 von (hl) kopiert; gleichzeitig werden die Bits 4 bis 7 von (hl) in die Akkumulatorbits ∅ bis 3 kopiert.

Rechts-Dezimal-Rotation **rrd**

Die Akkumulatorbits ∅ bis 3 werden in die Bits 4 bis 7 von (hl) kopiert; gleichzeitig werden die Bits 4 bis 7 von (hl) in die Bits ∅ bis 3 von (hl) kopiert; währenddessen werden die Bits ∅ bis 3 von (hl) in die Akkumulatorbits ∅ bis 3 kopiert.

Akkumulator-Operationen

1) Den Akkumulator komplementieren **cpl**
Jedes Bit mit Wert 1 bekommt den Wert ∅. Jedes Bit mit Wert ∅ bekommt den Wert 1.

2) Den Akkumulator negieren **neg**
a wird komplementiert, und zu a wird eins addiert.

3) Das Übertragsflag komplementieren **ccf**
Dieser Befehl setzt das Übertragsflag, wenn es nicht gesetzt ist; andernfalls löscht er es.

4) Den Übertrag setzen **scf**
Diese Anweisung setzt das Übertragsflag.

5) Dezimalanpassung des Akkumulators **daa**
Korrigiert a nach einer bcd-Addition und -Subtraktion.
(Anm. d. Übers.: bcd steht für binary coded decimals, d. h. binär codierte Dezimalzahlen.)

Restart **rst**

Speichert den Befehlszähler auf dem Stapel und springt zu Speicherzelle 8 * n.

n	Opcode	Mnemonic	springt zu
Ø	C7	RSTØ	Ø
1	CF	RST8	8
2	D7	RST1ØH	16
3	DF	RST18H	24
4	E7	RST2ØH	32
5	EF	RST28H	4Ø
6	F7	RST3ØH	48
7	FF	RST38H	56

Blockbehandlung

Diese zusammengesetzten Befehle sind dafür bestimmt, Daten zu bewegen oder im Speicher nach Daten zu suchen.

1) Blockladen mit Inkrementieren **ldi**
Dieser Befehl lädt ein Byte aus (hl) in (de); er inkrementiert hl und de und dekrementiert bc.
2) Wiederholtes Blockladen mit Inkrementieren **ldir**
Diese Anweisung lädt ein Byte aus (hl) in (de); sie inkrementiert hl und de und dekrementiert bc. Dies wird so lange wiederholt, bis bc = Ø.
3) Blockladen mit Dekrementieren **ldd**
Dieser Befehl lädt ein Byte aus (hl) in (de) und dekrementiert hl, de und bc.
4) Wiederholtes Blockladen mit Dekrementieren **lddr**
Diese Anweisung lädt ein Byte aus (hl) in (de) und dekrementiert hl, de und bc. Dies wird solange wiederholt bis bc = Ø.
5) Vergleichen mit Inkrementieren **cpi**
Dieser Befehl vergleicht a mit (hl). Er inkrementiert hl und dekrementiert bc.
6) Wiederholtes Vergleichen mit Inkrementieren **cpir**
Diese Anweisung vergleicht a mit (hl). Sie inkrementiert hl und dekrementiert bc. Dies wird solange wiederholt bis a = (hl) oder bc = Ø.
7) Vergleichen mit Dekrementieren **cpd**
Dieser Befehl vergleicht a und (hl); er dekrementiert hl und bc.
8) Wiederholtes Vergleichen mit Dekrementieren **cpdr**
Diese Anweisung vergleicht a und (hl); sie dekrementiert hl und bc. Dies wird solange wiederholt bis a = (hl) oder bc = Ø.

Teil B

4 Einleitung

Die vierzig Maschinencode-Routinen in Teil B werden zur Erleichterung in einer Standardform aufgelistet. Diese Einleitung erklärt die Form und beschreibt ein BASIC-Programm, mit dem man die Routinen in den Speicher laden kann.

Länge

Diese Rubrik bezieht sich auf die Länge der Routine in Bytes.

Anzahl der Variablen

Die Ausführung einiger Routinen kann gesteuert werden durch Ändern der Werte einer oder mehrerer Variablen, die über den Druckerpuffer an die Routine übergeben werden.

Prüfsumme

Jede Routine wird als Folge von natürlichen Zahlen dargestellt, die in aufeinanderfolgende Speicherzellen eingePOKEt werden sollen. Die Prüfsumme (d. h. die Summe aller Zahlen, welche die Routine bilden) wird angegeben, damit der Benutzer sicherstellen kann, daß er die Routine richtig geladen hat. (Falls sich Fehler nicht gegenseitig aufheben)

Arbeitsweise

Die Aufgabe, welche die Routine ausführt, wird in dieser Rubrik kurz erklärt.

Variablen

Zu jeder Variablen werden in dieser Rubrik ihr Name, ihre Länge und ihre Zelle im Druckerpuffer definiert. Eine Variable, die ein Byte lang ist, muß eine positive ganze Zahl zwischen Ø und 255 einschließlich sein. Sie wird von BASIC oder der Tastatur übergeben mittels:

POKE Zelle, Wert

Eine 2-Byte-Variable wird mit zwei Befehlen übergeben:

POKE Zelle, Wert − 256 * INT (Wert/256)

POKE Zelle + 1, INT (Wert/256)

Die verwendeten Zellen sind im Druckerpuffer.

Aufruf

Routinen werden mittels der USR-Funktion aufgerufen, die in einen Befehl eingearbeitet werden muß. Wenn die Maschinencode-Routine nach Abschluß keinen Wert an BASIC zurückgibt, wird der RAND-Befehl empfohlen, wie in

RAND USR Adresse

Wenn der Wert im Registerpaar bc an BASIC zurückgegeben werden soll, wird entweder

LET A = USR Adresse

oder

PRINT USR Adresse

empfohlen, je nachdem, ob der zurückgegebene Wert in einer BASIC-Variablen gespeichert werden soll oder ob er auf dem Bildschirm gedruckt werden soll.

Fehlerüberwachung

In dieser Rubrik werden die Tests erklärt, welche die Routine vornimmt, um zu überprüfen, ob nicht-logische oder gegensätzliche Variablenwerte usw. vorkommen.

Kommentare

Einfache Varianten der Hauptroutinen werden in dieser Rubrik erklärt.

Maschinencode Listing

Die Routine wird in Assemblersprache dargestellt, wobei die Absolutform in der dritten Spalte die Überschrift "einzugebende Zahlen" trägt. Um die Routine zu laden, werden die Zahlen in der dritten Spalte nacheinander in den Speicher eingePOKEt. Alle Zahlen sind in Dezimaldarstellung gegeben.

Wie es funktioniert

In dieser Rubrik wird mit Hilfe von Verweisen auf das Maschinencode Listing erklärt, wie die Routine funktioniert.

Maschinencode-Ladeprogramm

Fast alle Maschinencode-Routinen in diesem Band sind *verschiebbar*, d. h. sie funktionieren normal, egal wo sie sich im RAM befinden. Wenn eine Routine nicht verschiebbar ist, erklärt der Abschnitt "Kommentare" wie sie geändert werden muß, wenn sie eine für sie nicht vorgesehene Startadresse erhalten soll.

Wir haben in Teil A, Kapitel 2 gesehen, daß der Spectrum verschiedene Teile des RAM für verschiedene Funktionen verwendet, und daß der Bereich zwischen den Zellen, auf welche die Systemvariablen RAMTOP und UDG zeigen, zur Speicherung von Maschinencode-Routinen vorgesehen ist.

Programm BP kann verwendet werden, um eine Maschinencode-Routine zu laden, abzuändern und zu bewegen. Der Benutzer kann damit den RAMTOP-Zeiger neu festsetzen, um mehr Platz für eine Routine zu schaffen; er kann damit eine Routine über die Tastatur eingeben; er kann vorwärts und rückwärts in der Routine gehen, um einen Fehler zu korrigieren und um Teile der Routine einzufügen oder zu streichen.

Wenn das Programm mit RUN gefahren wird, druckt es die niedrigste Adresse, an der eine Routine gespeichert werden kann, d. h. eine Adresse über RAMTOP, und sie druckt aus, wieviel Platz zwischen dieser Adresse und dem Ende des RAM zur Verfügung steht.

In der Ausführung mit 16K ist die niedrigste Adresse 32600, es sei denn der Benutzer hat die Systemvariable RAMTOP verändert. Entsprechend ist in der Ausführung mit 48K normalerweise die niedrigste Adresse 65368.

Die 168 Bytes am Ende des RAM sind im Normalfall für Graphikzeichen, die vom Benutzer selbst definiert werden, reserviert. Allerdings kann der Benutzer diesen Bereich mit Hilfe des Programms überschreiben, wenn er es will. Als Alternative kann er eine neue niedrigstmögliche Adresse wählen, die das Programm dann unter Verwendung von CLEAR in den RAMTOP-Zeiger gibt. Das Programm nimmt keine Adresse unter 27000 an, weil die Routine dann in den Bereich eindringen würde, den das Programm selbst benötigt. Das Programm fragt nach der Adresse, an der die Routine anfangen soll. So kann der Benutzer Platz für mehrere Routinen reservieren und sie dann jeweils einzeln laden.

Nachdem dem Benutzer die Möglichkeit gegeben wurde, seine Wahl nochmals zu ändern, druckt das Programm das Hauptdisplay. Abbildung BF1 zeigt die Form des Displays, wenn die Routine "Bildschirm-Inversion" beginnend in Zelle 32000 geladen wurde. In der ersten Spalte steht die Adresse, in der zweiten der Inhalt der Adresse und in der dritten die Prüfsumme. Die Routine "Bildschirm-Inversion" ist 18 Bytes lang und ihre Prüfsumme ist 1613. Sie besetzt deshalb die Zellen 32000 bis 32017, und die Prüfsumme für Zelle 32017, d. h. die Summe der Inhalte von den Zellen 32000 bis 32017, ist 1613.

Wenn das Hauptdisplay gezeigt wird, fällt dem Benutzer gleich eine bestimmte Zelle auf, weil ihr Dezimalinhalt blinkt. Sie heißt die *aktuelle* Zelle. Zu Beginn ist sie die gewählte Startadresse der Routine. Der Benutzer gibt eine ganze Zahl zwischen 0 und 255 einschließlich ein, welche vom Programm in die aktuelle Zelle eingePOKEt wird. Dann wird die folgende Zelle zur aktuellen Zelle. Auf diese Art kann man eine ganze Routine an ihren Platz einPOKEn, wobei bei jedem Schritt das Hauptdisplay auf den neuesten Stand gebracht wird und falls notwendig mit SCROLL verschoben wird.

Der Benutzer kann statt einer Zahl eine der Möglichkeiten in Tabelle BT1 eingeben. Dadurch werden Korrekturen ermöglicht.

Tabelle BT1
Verfügbare Möglichkeiten zum Editieren von Maschinencode.

Code	Möglichkeit
b	Aktuelle Zelle um eine Adresse zurückrücken.
b *Zahl*	Aktuelle Zelle um *Zahl* viele Adressen zurückrücken.
f	Aktuelle Zelle um eine Adresse vorwärtsrücken.
f *Zahl*	Aktuelle Zelle um *Zahl* viele Adressen vorwärtsrücken.
i *Zahl*	*Zahl* viele Bytes, die alle null enthalten, bei der aktuellen Zelle einfügen.
d *Zahl*	*Zahl* viele Bytes an der aktuellen Zelle weglassen.
t	Programm beenden.

Bemerkung: "i" bzw. "d" können u. U. mehrere Minuten dauern.

Programm BP Maschinencode-Ladeprogramm

```
100  GO SUB 8100
200  REM * * * * * Berechnen des verfügbaren Speichers
210  LET min = 1 + PEEK 23730 + 256 * PEEK 23731
220  LET p = PEEK 23732 + 256 * PEEK 23733
230  LET t = p − min + 1
400  REM * * * * * Startadresse erhalten
410  PRINT "Niedrigstmoeglicher □ Start = □"; min „"Anzahl □
     verfuegbare □ Bytes □ = □";t
420  INPUT "Wollen Sie die niedrigste Start-adresse aendern
     (Y oder N) ? ";Z$
430  IF Z$ = "Y" OR Z$ = "y" THEN GO TO 7000
440  INPUT "Geben Sie die Startadresse ein, bei der das Laden des
     Maschinen-codes beginnen soll ";a
450  IF a < min OR a > p THEN BEEP .2,24: GO TO 440
500  GO SUB 8100
510  LET t = t − a + min
520  PRINT "Sie □ koennen □ bis □ zu □";t; "□ Bytes", "ver-
     wenden"
530  LET u = PEEK 23675 + 256 * PEEK 23676
```

```
54Ø  IF a < u AND u < p THEN PRINT' "Wenn □Sie □ mehr □ als
     □";u - a; "□ Bytes", "verwenden, □ ueberschreiben □ Sie",
     "den □ Bereich □ der □ selbstdefinier-","ten □ Graphikzei-
     chen."
55Ø  IF a > = u THEN PRINT "Sie □ ueberschreiben □ den □
     Bereich", "der □ selbstdefinierten □ Zeichen."
56Ø  INPUT "Ist □ das □ in □ Ordnung □ (Y □ oder □ N)?";
     LINE Z$
57Ø  IF Z$ = "N" OR Z$ = "n" THEN GOT TO 7ØØØ
58Ø  IF Z$ < > "Y" AND Z$ < > "y" THEN BEEP .2,24: GO TO 56Ø
7ØØ  REM * * * * * Weitermachen und Laden
71Ø  LET l = a
75Ø  GO SUB 82ØØ
76Ø  INPUT "Geben Sie die Zahl b, f, i, d oder t ein ";Z$
77Ø  IF Z$ = "" THEN BEEP .2,24: GO TO 76Ø
78Ø  LET a$ = CHR$ (CODE Z$ (1) - 32 * (Z$ (1) > "£"))
79Ø  GO TO 8ØØ + 2ØØ * (a$ = "B") + 3ØØ * (a$ = "F") + 4ØØ * (a$ =
     "I") + 5ØØ * (a$ = "D") + 6ØØ * (a$ = "T")
8ØØ  LET x = VAL Z$
81Ø  IF l > p THEN BEEP .2,24: GO TO 75Ø
82Ø  IF x < Ø OR x > 255 OR x < > INT x THEN BEEP .2,24: GO TO
     76Ø
83Ø  POKE l, x
84Ø  LET l = l + 1 : IF l > p THEN LET l = p
85Ø  GO TO 74Ø
1ØØØ REM * * * * * Vorwärts gehen
1Ø1Ø LET l = l - 1
1Ø2Ø IF LEN Z$ > 1 THEN LET l = l + 1 - VAL Z$ (2 TO)
1Ø3Ø IF l < a THEN LET l = a
1Ø4Ø GO TO 74Ø
11ØØ REM * * * * * Rückwärts gehen
```

```
1110  LET l = l + 1
1120  IF LEN Z$ > 1 THEN LET l = l - 1 + VAL Z$ (2 TO )
1130  IF l > p THEN LET l = p
1140  GO TO 740
1200  REM * * * * * Einschieben
1210  IF LEN Z$ = 1 THEN LET n = 1: GO TO 1225
1220  LET n = VAL Z$ (2 TO ): IF n < 1 OR n > p - l OR n <> INT n
      THEN BEEP .2,24: GO TO 740
1225  CLS : GO SUB 8100: PRINT TAB 4; "Einschieben ist im Gange"
1230  FOR j = p TO l + n STEP - 1
1240  POKE j, PEEK (j - n)
1250  NEXT j
1260  FOR j = l TO l + n - 1
1270  POKE j, 0
1280  NEXT j
1290  GO TO 740
1300  REM * * * * * Streichen
1310  IF LEN Z$ = 1 THEN LET n = 1: GO TO 1330
1320  LET n = VAL Z$ (2 TO ): IF n < 1 OR n > p - l OR n <> INT n
      THEN BEEP .2,24: GO TO 740
1330  CLS : GO SUB 8100: PRINT TAB 5; "Streichen ist im Gange"
1340  FOR j = l TO p - n
1350  POKE j, PEEK (j + n)
1360  NEXT j
1370  IF j > = p THEN POKE p, 0
1380  GO TO 740
1400  REM * * * * * Ende Programm
1401  PRINT AT 21,7; "Programm ist beendet"
1410  STOP
7000  REM * * * * * RAMTOP neu festsetzen
7010  INPUT "Geben Sie die neue Startadresse ein ";a
```

```
7020  IF a < 27000 OR a > p THEN BEEP .2,24: GO TO 7010
7030  CLEAR a - 1
7040  RUN
7999  STOP
8100  CLS
8110  PRINT TAB 3; "Maschinencode-Ladeprogramm" "
8120  RETURN
8200  REM * * * * * Speicher drucken
8210  GO SUB 8100
8220  PRINT "Adresse □□□□ Inhalt □□□□ Pruefsumme
8230  LET c = 0
8240  LET s = l - 8: IF s < a THEN LET s = a: GO TO 8280
8250  FOR j = a TO s - 1
8260  LET c = c + PEEK j
8270  NEXT j
8280  LET f = s + 17: IF f > p THEN LET f = p
8290  FOR j = s TO f
8300  LET c = c + PEEK j
8310  PRINT AT j - s + 3,1; j; TAB 12; PEEK j; TAB 22; c
8320  NEXT j
8400  LET pos = l - s + 3
8410  PRINT AT pos, 12; FLASH 1; PEEK l
8420  RETURN
```

Maschinencode-Ladeprogramm		
Adresse	Inhalt	Prüfsumme
32ØØØ	33	33
32ØØ1	Ø	33
32ØØ2	64	97
32ØØ3	1	98
32ØØ4	Ø	98
32ØØ5	24	122
32ØØ6	22	144
32ØØ7	255	399
32ØØ8	122	521
32ØØ9	15Ø	671
32Ø1Ø	119	79Ø
32Ø11	35	825
32Ø12	11	836
32Ø13	12Ø	956
32Ø14	177	1133
32Ø15	32	1165
32Ø16	247	1412
32Ø17	2Ø1	1613

Abbildung BF1
Das vom Maschinencode-Ladeprogramm produzierte Display, wenn die Routine "Bildschirm-Inversion" in Zelle 32000 geladen wurde.

5 Verschiebe-Routinen (Scroll)

Verschieben von Attributen nach links

Länge: 23
Anzahl der Variablen: 1
Prüfsumme: 1574

Arbeitsweise

Diese Routine verschiebt die Attribute aller Bildschirmzeichen um ein Zeichen nach links.

Variablen

Name	Länge	Speicherzelle	Kommentar
new attr	1	23296	Das Attribut für die am weitesten rechts stehende Spalte

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ist sehr nützlich, wenn man Text- und Graphikbereiche hell hervorheben will. Wenn man nur die obersten 22 Zeilen verschieben will, so muß man 24* ersetzen durch 22.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 22528	33 Ø 88
	ld a, (23296)	58 Ø 91
	ld c, 24	14 24*
next line	ld b, 31	6 31
next char	inc hl	35
	ld e, (hl)	94
	dec hl	43

Marke	Assemblersprache	einzugebende Zahlen
	ld (hl),e	115
	inc hl	35
	djnz, next char	16 249
	ld (hl),a	119
	inc hl	35
	dec c	13
	jr nz, next line	32 242
	ret	2Ø1

Wie es funktioniert

Die Routine lädt die Adresse des Attributbereichs in das Registerpaar hl. Dann lädt sie den Wert des Attributs, das in die rechte Spalte kommen soll, in den Akkumulator. Die Anzahl der Zeilen, die verschoben werden sollen, wird in Register c geladen, so daß es als Zeilenzähler verwendet werden kann. Das Register b wird auf die Anzahl der Zeichen pro Zeile minus eins gesetzt, damit es als Zähler benutzt werden kann.

Die Routine inkrementiert hl, damit es auf das nächste Attribut zeigt, und lädt dieses in Register e. Dann dekrementiert sie hl und poket es mit dem Wert in e. hl wird nochmals inkrementiert, damit es auf das nächste Attribut zeigt. Das Register b wird dekrementiert, und wenn keine Null darin steht, springt die Routine zurück zu "next char" (deutsch: nächstes Zeichen). Das Registerpaar hl zeigt nun auf die rechte Spalte. Diese wird jetzt mit dem Wert im Akkumulator gepoket. Die Routine inkrementiert hl, damit es auf den Anfang der nächsten Zeile zeigt. Der Zeilenzähler in Register c wird dekrementiert. Wenn der Wert des Ergebnisses nicht null ist, springt die Routine zurück zu "next line" (deutsch: nächste Zeile).

Die Routine springt dann zu BASIC zurück.

Verschieben von Attributen nach rechts

Länge: 23
Anzahl der Variablen: 1
Prüfsumme: 1847

Arbeitsweise

Diese Routine verschiebt die Attribute aller Bildschirmzeichen um ein Zeichen nach rechts.

Variablen

Name	Länge	Speicherzelle	Kommentar
new attr	1	23296	Das Attribut für die am weitesten links stehende Spalte

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ist nützlich, wenn man Text- und Graphikbereiche hell hervorheben will. Wenn man nur die obersten 22 Zeilen verschieben will, so muß man 24* ersetzen durch 22.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 23295	33 255 9∅
	ld a, (23296)	58 ∅ 91
	ld c, 24	14 24*
next line	ld b, 31	6 31
next char	dec hl	43
	ld e, (hl)	94
	inc hl	35
	ld (hl), e	115
	dec hl	43
	djnz, next char	16 249
	ld (hl), a	119
	dec hl	43
	dec c	13
	jr nz, next line	32 242
	ret	2∅1

Wie es funktioniert

Die Routine lädt die Adresse des letzten Bytes des Attributbereichs in das Registerpaar hl. Sie lädt den Wert des Attributs für die am weitesten links stehende Spalte in den Akkumulator. Die Anzahl der Zeilen, die verschoben werden sollen, wird in Register c geladen; somit kann es als Zeilenzähler verwendet werden. Das Register b wird auf die Anzahl der Zeichen pro Zeile minus eins gesetzt, damit man es als Zähler benutzen kann.

Die Routine dekrementiert hl, so daß es auf das vorangehende Attribut zeigt. Dann lädt sie den Wert dieses Attributs in Register e. Darauf inkrementiert sie hl und poket es mit dem Wert in Register e. hl wird abermals dekrementiert, damit es auf das vorangehende Attribut zeigt. Der Zähler in Register b wird dekrementiert, und wenn darin keine Null steht, springt die Routine zurück zu "next char".

Das Registerpaar hl zeigt nun auf die am weitesten links stehende Spalte;

diese wird mit dem Wert im Akkumulator gepoket. Die Routine dekrementiert hl, damit es auf das rechte Ende der vorangehenden Zeile zeigt. Der Zeilenzähler wird dekrementiert, und wenn er nicht null ist, springt die Routine zurück zu "next line".

Die Routine springt dann zu BASIC zurück.

Verschieben von Attributen nach oben

Länge: 21
Anzahl der Variablen: 1
Prüfsumme: 1591

Arbeitsweise

Diese Routine verschiebt die Attribute aller Bildschirmzeichen um ein Zeichen nach oben.

Variablen

Name	Länge	Speicherzelle	Kommentar
new attr	1	23296	Das Attribut für die unterste Zeile

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ist nützlich, wenn man Text- und Graphikbereiche hell hervorheben will. Sofern man nur die obersten 22 Zeilen verschieben will, so muß man 224* ersetzen durch 16Ø.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 2256Ø	33 32 88
	ld de, 22528	17 Ø 88
	ld bc, 736	1 224* 2
	ldir	237 176

Marke	Assemblersprache	einzugebende Zahlen
	ld a, (23296)	58 ∅ 91
	ld b, 32	6 32
next char	ld (de), a	18
	inc de	19
	djnz next char	16 252
	ret	2∅1

Wie es funktioniert

Die Routine lädt die Adresse der zweiten Attributzeile in hl, die Adresse der ersten Zeile in de und die Anzahl der zu bewegenden Bytes in bc.

Die bc Bytes, die bei hl beginnen, werden mit dem Befehl "ldir" in de hineinkopiert. Als Ergebnis zeigt dann de auf die unterste Attributzeile. Der Akkumulator wird mit dem Code des Attributs geladen, das in die unterste Zeile kommen soll. Die Routine lädt Register b mit der Anzahl der Zeichen in einer Zeile, so daß es als Zähler dienen kann.

Das Registerpaar de wird mit dem Wert im Akkumulator gepoket. Dann wird de inkrementiert, damit es auf das nächste Byte zeigt. Die Routine dekrementiert den Zähler, und wenn er nicht null ist, springt sie zurück zu "next char".

Die Routine springt dann zu BASIC zurück.

Verschieben von Attributen nach unten

Länge: 21
Anzahl der Variablen: 1
Prüfsumme: 2∅57

Arbeitsweise

Diese Routine verschiebt die Attribute aller Bildschirmzeichen um ein Zeichen nach unten.

Variablen

Name	Länge	Speicherzelle	Kommentar
new attr	1	23296	Das Attribut für die oberste Zeile

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ist nützlich, wenn man Text- und Graphikbereiche hell hervorheben will. Sofern man nur die obersten 22 Zeilen verschieben will, muß man folgende Änderungen vornehmen:

223* ersetzen durch 159

255** ersetzen durch 191

224*** ersetzen durch 16Ø

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 23263	33 223* 9Ø
	ld de, 23295	17 255** 9Ø
	ld bc, 736	1 224*** 2
	lddr	237 184
	ld a, (23296)	58 Ø 91
	ld b, 32	6 32
next char	ld (de),a	18
	dec de	27
	djnz next char	16 252
	ret	2Ø1

Wie es funktioniert

Die Routine lädt die Adresse des letzten Attributs in der 23. Zeile in das Registerpaar hl. Sie lädt die Adresse des letzten Attributs in der 24. Zeile in de und die Anzahl der Bytes, die bewegt werden sollen, in bc. Dann verschiebt der "lddr"-Befehl die bc Bytes, die bei hl enden, so daß sie nun bei de enden. Das Ergebnis ist, daß in de die Adresse des letzten Attributs der ersten Zeile abgelegt ist.

Der Akkumulator wird dann mit dem Wert des Attributs für die oberste Zeile geladen. Die Routine lädt Register b mit der Anzahl der Bytes in der obersten Zeile, damit man es als Zähler verwenden kann. Das Registerpaar de wird mit dem Wert im Akkumulator gepoket, und de wird dekrementiert, so daß es auf das vorangehende Byte zeigt. Die Routine dekrementiert den Zähler, und wenn er nicht null ist, springt sie zu "next char".

Dann springt die Routine zu BASIC zurück.

Verschieben um ein Zeichen nach links

Länge: 21
Anzahl der Variablen: Ø
Prüfsumme: 1745

Arbeitsweise

Diese Routine verschiebt den Inhalt der Displaydatei um ein Zeichen nach links.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ist nützlich, wenn man den Bildschirm als "Fenster" verwendet, das gerade einen kleinen Bereich einer größeren Displayfläche zeigt. Das "Fenster" wird mit Verschiebe-(Scroll-)Routinen bewegt.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 16384	33 Ø 64
	ld d,l	85
	ld a, 192	62 192
next line	ld b, 31	6 31
next byte	inc hl	35
	ld e, (hl)	94
	dec hl	43
	ld (hl),e	115
	inc hl	35
	djnz next byte	16 249
	ld (hl),d	114
	inc hl	35
	dec a	61
	jr nz, next line	32 242
	ret	2Ø1

Wie es funktioniert

Die Routine lädt die Adresse der Displaydatei in Registerpaar hl und setzt das Register d auf null. Die Anzahl der Bildschirmzeilen wird in den Akkumulator geladen. Register b wird auf die Anzahl der Zeichen pro Zeile minus eins gesetzt, da dies die Anzahl der zu kopierenden Bytes ist.

Die Routine inkrementiert hl, damit es auf das nächste Byte zeigt, und lädt seinen Wert in Register e. hl wird dekrementiert und mit dem Wert in e gepoket. Dann wird hl inkrementiert, damit es das nächste Byte adressiert. Andererseits wird der Zähler in Register b dekrementiert. Wenn er nicht null ist, springt die Routine zurück zu "next byte".

Wenn in Register b null steht, ist das letzte Byte der Zeile kopiert worden, und hl zeigt auf das am weitesten rechts stehende Byte. Dies wird mit null gepoket, und hl wird inkrementiert, damit es auf die nächste Zeile zeigt. Die Routine dekrementiert den Zeilenzähler im Akkumulator, und wenn er nicht null ist, springt sie zu "next line".

Es erfolgt ein Rücksprung zu BASIC.

Verschieben um ein Zeichen nach rechts

Länge: 22
Anzahl der Variablen: Ø
Prüfsumme: 1976

Arbeitsweise

Diese Routine verschiebt den Inhalt der Displaydatei um ein Zeichen nach rechts.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ist nützlich, wenn man den Bildschirm als "Fenster" verwendet, das gerade einen kleinen Bereich einer größeren Displayfläche zeigt. Das "Fenster" wird mit Verschiebe-(Scroll-) Routinen bewegt.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 22527	33 255 87
	ld d, Ø	22 Ø
	ld a, 192	62 192
next line	ld b, 31	6 31
next byte	dec hl	43
	ld e, (hl)	94
	inc hl	35
	ld (hl),e	115
	dec hl	43

Marke	Assemblersprache	einzugebende Zahlen
	djnz next byte	16 249
	ld (hl),d	114
	dec hl	43
	dec a	61
	jr nz, next line	32 242
	ret	2Ø1

Wie es funktioniert

Die Routine lädt die Adresse des letzten Bytes der Displaydatei in Registerpaar hl und setzt Register d auf null. Dann lädt sie die Anzahl der Bildschirmzeilen in den Akkumulator. Register b wird auf die Anzahl der Zeichen pro Zeile minus eins gesetzt, damit man es als Zähler verwenden kann.

Das Registerpaar hl wird dekrementiert, damit es auf das vorangehende Byte zeigt, und sein Wert wird in Register e geladen. Dann wird hl inkrementiert und mit dem Wert in e gepoket. Die Routine dekrementiert darauf hl, so daß es auf das vorangehende Byte zeigt, und dekrementiert den Zähler in Register b ebenfalls. Wenn in Register b keine Null steht, springt die Routine zurück zu "next byte".

Wenn dagegen in Register b null steht, zeigt hl auf das am weitesten links stehende Byte der Zeile. Dieses wird mit null gepoket, und hl wird dekrementiert, damit es auf die vorangehende Zeile zeigt. Dann dekrementiert die Routine den Zähler im Akkumulator, und wenn darin nicht null steht, springt sie zu "next line".

Die Routine springt dann zu BASIC zurück.

Verschieben um ein Zeichen nach oben

Länge: 68
Anzahl der Variablen: Ø
Prüfsumme: 6328

Arbeitsweise

Diese Routine verschiebt den Inhalt der Displaydatei um acht Pixel nach oben.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Keine

Maschinenencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 16384	33 ∅ 64
	ld de, 16416	17 32 64
save	push hl	229
	push de	213
	ld c, 23	14 23
next line	ld b,32	6 32
copy byte	ld a, (de)	26
	ld (hl),a	119
	ld a,c	121
	and 7	23∅ 7
	cp 1	254 1
	jr nz, next byte	32 2
	sub a	151
	ld (de), a	18
next byte	inc hl	35
	inc de	19
	djnz copy byte	16 241
	dec c	13
	jr z, restore	4∅ 19
	ld a,c	121
	and 7	23∅ 7
	cp ∅	254 ∅
	jr z, next block	4∅ 22
	cp 7	254 7
	jr nz, next line	32 225
	push de	213
	ld de, 1792	17 ∅ 7
	add hl, de	25
	pop de	2∅9
	jr next line	24 217
restore	pop de	2∅9
	pop hl	225
	inc d	2∅
	inc h	36
	ld a,h	124
	cp 72	254 72
	jr nz, save	32 2∅4
	ret	2∅1
next block	push hl	229
	ld hl, 1792	33 ∅ 7
	add hl, de	25
	ex de,hl	235
	pop hl	225
	jr next line	24 198

Wie es funktioniert

Die Adresse der Displaydatei wird in das Registerpaar hl geladen; die Adresse des Bytes, das acht Zeilen weiter unten steht, wird in de geladen. Dann speichert die Routine hl und de auf dem Stapel. Darauf lädt sie Register c mit der Anzahl der "PRINT-Zeilen" auf dem Bildschirm minus eins. Die Anzahl der Bytes in einer Displayzeile wird in Register b geladen, damit es als Zähler verwendet werden kann.

Der Akkumulator wird mit dem Byte, das von de adressiert wird, geladen und in hl eingepoket. Der Akkumulator wird mit dem Inhalt des Registers c geladen, und wenn dieses 1, 9 oder 17 enthält, wird de mit null gepoket. Dann inkrementiert die Routine hl und de, so daß sie auf die nächsten Bytes zeigen. Sie dekrementiert des weiteren den Zähler in Register b, und wenn keine Null darin steht, springt sie zu "copy byte" (deutsch: Byte kopieren).

Der Zeilenzähler in Register c wird dann dekrementiert. Wenn es null enthält, springt die Routine zu "restore" (deutsch: wiederherstellen). Wenn 8 oder 16 in c stehen, springt sie zu "next block". Sofern in c nicht 7 oder 15 stehen, macht sie eine Schleife zu "next line". 1792 wird zu hl addiert, so daß hl auf den nächsten Bildschirmblock zeigt. Dann springt die Routine zu "next line".

Bei "restore" werden de und hl vom Stapel zurückgeholt, und zu beiden wird 256 addiert. So zeigen de und hl auf eine Zeile tiefer als in der vorangehenden Schleife. Sobald 18432 in hl steht, springt die Routine zu BASIC zurück, sonst springt sie zu "save" (deutsch: sichern, retten). Bei "next block" wird 1792 zu de addiert, so daß de auf den nächsten Bildschirmblock zeigt. Die Routine macht dann eine Schleife zu "next line".

Verschieben um ein Zeichen nach unten

Länge: 73
Anzahl der Variablen: Ø
Prüfsumme:7987

Arbeitsweise

Diese Routine verschiebt den Inhalt der Displaydatei um acht Pixel nach unten.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Keine

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 22527	33 255 87
	ld de, 22495	17 223 87
save	push hl	229
	push de	213
	ld c, 23	14 23
next line	ld b,32	6 32
copy byte	ld a, (de)	26
	ld (hl),a	119
	ld a,c	121
	and 7	23Ø 7
	cp 1	254 1
	jr nz, next byte	32 2
	sub a	151
	ld (de),a	18
next byte	dec hl	43
	dec de	27
	djnz copy byte	16 241
	dec c	13
	jr z, restore	4Ø 21
	ld a,c	121
	and 7	23Ø 7
	cp Ø	254 Ø
	jr z, next block	4Ø 24
	cp 7	254 7
	jr nz, next line	32 225
	push de	213
	ld de, 1792	17 Ø 7
	and a	167
	sbc hl,de	237 82
	pop de	2Ø9
	jr next line	24 215
restore	pop de	2Ø9
	pop hl	225
	dec d	21
	dec h	37
	ld a,h	124
	cp 79	254 79
	ret z	2ØØ
	jr save	24 2Ø1
next block	push hl	229
	ld hl, 1792	33 Ø 7
	ex de, hl	235
	and a	167
	sbc hl,de	237 82

Marke	Assemblersprache	einzugebende Zahlen
	ex de, hl	235
	pop hl	225
	jr next line	24 193

Wie es funktioniert

Die Routine lädt die Adresse des letzten Bytes der Displaydatei in Registerpaar hl und die Adresse des Bytes acht Zeilen weiter oben in de. Dann speichert sie hl und de auf dem Stapel. Register c wird dann mit der Anzahl der "PRINT-Zeilen" auf dem Bildschirm minus eins geladen. Die Anzahl der Bytes in einer Displayzeile wird in Register b geladen; damit kann es als Zähler dienen.

Der Akkumulator wird mit dem von de adressierten Byte geladen, und dieses wird in hl eingepoket. Dann wird der Akkumulator mit dem Inhalt des Registers c geladen, und wenn dieses 1, 9 oder 17 enthält, wird de mit null gepoket. Daraufhin dekrementiert die Routine hl und de, damit sie auf die vorangehenden Displaybytes zeigen. Nun dekrementiert sie den Zähler in Register b, und wenn keine Null darin steht, springt sie zu "copy byte".

Der Zeilenzähler in Register c wird dekrementiert, und wenn er null ist, springt die Routine zu "restore". Falls 8 oder 16 in c stehen, springt sie zu "next block". Sofern in c weder 7 noch 15 stehen, macht die Routine eine Schleife zu "next line". 1792 wird von hl abgezogen, so daß hl auf den vorangehenden Bildschirmblock zeigt. Danach springt die Routine zu "next line".

Bei "restore" werden de und hl vom Stapel zurückgeholt, und von beiden wird 256 subtrahiert. Nun zeigen de und hl auf eine Zeile höher als in der vorhergehenden Schleife. Wenn in hl die Zahl 2Ø479 steht, springt die Routine zu BASIC zurück, sonst springt sie zu "save". Bei "next block" wird 1792 von de subtrahiert, so daß de auf den vorangehenden Bildschirmblock zeigt. Die Routine macht dann eine Schleife zu "next line".

Verschieben um ein Pixel nach links

Länge: 17
Anzahl der Variablen: Ø
Prüfsumme: 1828

Arbeitsweise

Diese Routine verschiebt den Inhalt der Displaydatei um ein Pixel nach links.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ergibt eine glattere Bewegung als das "Verschieben um ein Zeichen nach links". Allerdings muß man sie achtmal aufrufen, wenn man die Anzeige um ein ganzes Zeichen verschieben will.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 22527	33 255 87
	ld c, 192	14 192
next line	ld b, 32	6 32
	or a	183
next byte	rl (hl)	2Ø3 22
	dec hl	43
	djnz next byte	16 251
	dec c	13
	jr nz, next line	32 245
	ret	2Ø1

Wie es funktioniert

Die Adresse des letzten Bytes der Displaydatei wird in Registerpaar hl, die Anzahl der Zeilen in der Displaydatei in Register c geladen, damit man einen Zähler zur Verfügung hat. Das Register b soll ebenfalls als Zähler Verwendung finden und wird deshalb mit der Anzahl der Bytes in einer Zeile geladen. Das Übertragsflag wird dann auf null gesetzt.

Dann rotiert die Routine dasjenige Byte, das von hl adressiert wird, um ein Bit nach links, wobei das Übertragsflag in das am weitesten rechts stehende Bit kopiert wird. Außerdem wird das am weitesten links stehende Bit in das Übertragsflag kopiert. Die Routine dekrementiert Registerpaar hl, weil es auf das nächste Byte zeigen soll. Sie dekrementiert ebenfalls den Zähler in Register b. Wenn darin keine Null steht, macht sie eine Schleife zu "next byte". Die Zeilennummer wird dekrementiert, und wenn sie nicht null ist, springt die Routine zurück zu "next line".

Dann springt die Routine zu BASIC zurück.

Verschieben um ein Pixel nach rechts

Länge: 17
Anzahl der Variablen: Ø
Prüfsumme: 155Ø

Arbeitsweise

Diese Routine verschiebt den Inhalt der Displaydatei um ein Pixel nach rechts.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ergibt eine glattere Bewegung als das "Verschieben um ein Zeichen nach rechts". Allerdings muß man sie achtmal aufrufen, wenn man die Anzeige um ein ganzes Zeichen verschieben will.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 16384	33 Ø 64
	ld c, 192	14 192
next line	ld b, 32	6 32
	or a	183
next byte	rr (hl)	2Ø3 3Ø
	inc hl	35
	djnz next byte	16 251
	dec c	13
	jr nz, next line	32 245
	ret	2Ø1

Wie es funktioniert

Die Routine lädt die Adresse der Displaydatei in das Registerpaar hl und die Anzahl der Displayzeilen in Register c, damit es als Zeilenzähler verwendet werden kann. Die Anzahl von Bytes in einer Zeile wird in Register b geladen, da es als Zähler dienen soll. Darauf wird das Übertragsflag auf null gesetzt. Dann rotiert die Routine das von hl adressierte Byte um ein Bit nach rechts, wobei das Übertragsflag in das am weitesten links stehende Bit kopiert wird und das am weitesten rechts stehende Bit in das Übertragsflag kopiert wird. Das Registerpaar hl wird inkrementiert, damit es auf das nächste Byte zeigt; dann wird der Zähler in Register b dekrementiert. Wenn darin keine Null steht, springt die Routine zurück zu "next byte". Sie dekrementiert den Zeilenzähler, und wenn er ungleich null ist, springt sie zu "next line" zurück.

Es erfolgt ein Rücksprung zu BASIC.

Verschieben um ein Pixel nach oben

Länge: 91
Anzahl der Variablen: Ø
Prüfsumme: 9228

Arbeitsweise

Diese Routine verschiebt den Inhalt der Displaydatei um ein Pixel nach oben.

Aufruf

RAND USR Adresse

Fehlerübewachung

Keine

Kommentare

Keine

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 16384	33 Ø 64
	ld de, 1664Ø	17 Ø 65
	ld c, 192	14 192
next line	ld b, 32	6 32
copy byte	ld a, (de)	26
	ld (hl),a	119
	ld a,c	121
	cp 2	254 2
	jr nz, next byte	32 2
	sub a	151
	ld (de), a	18
next byte	inc de	19
	inc hl	35
	djnz copy byte	16 243
	push de	213
	ld de, 224	17 224 Ø
	add hl,de	25
	ex (sp),hl	227
	add hl,de	25
	ex de,hl	235
	pop hl	225

Marke	Assemblersprache	einzugebende Zahlen
	dec c	13
	ld a,c	121
	and 7	23Ø 7
	cp Ø	254 Ø
	jr nz, subtract	32 1Ø
	push de	213
	ld de, 2Ø16	17 224 7
	and a	167
	sbc hl,de	237 82
	pop de	2Ø9
	jr next block	24 14
subtract	cp 1	254 1
	jr nz, next block	32 1Ø
	push hl	229
	ex de,hl	235
	ld de, 2Ø16	17 224 7
	and a	167
	sbc hl,de	237 82
	ex de,hl	235
	pop hl	225
next block	ld a,c	121
	and 63	23Ø 63
	cp Ø	254 Ø
	jr nz, add	32 6
	ld a,7	62 7
	add a, h	132
	ld h,a	1Ø3
	jr next line	24 187
add	cp1	254 1
	jr nz, next line	32 183
	ld a,7	62 7
	add a,d	13Ø
	ld d,a	87
	ld a,c	121
	cp 1	254 1
	jr nz, next line	32 174
	ret	2Ø1

Wie es funktioniert

Die Routine lädt die Adresse der Displaydatei in das Registerpaar hl und die Adresse des ersten Bytes der zweiten Displayzeile in das Registerpaar de. Sie lädt Register c mit der Anzahl der Displayzeilen und Register b mit der Anzahl von Bytes in einer Zeile, da es als Zähler verwendet werden soll.

Der Akkumulator wird mit dem von de adressierten Byte geladen. Dieses wird in den durch hl adressierten Speicherplatz geladen. Die Routine lädt den Akkumulator mit dem Inhalt des Registers c. Wenn darin der Wert zwei steht, zeigt de auf die unterste Bildschirmzeile, und dann wird es mit null gepoket. Darauf inkrementiert die Routine de und hl, damit sie auf die nächsten Bytes

zeigen. Hingegen dekrementiert sie den Zähler in Register b, und wenn er nicht null ist, springt die Routine zu "copy byte".

Sie addiert 224 sowohl zu Registerpaar hl als auch zu de, so daß sie auf die nächste Displayzeile zeigen. Der Zeilenzähler in Register c wird dekrementiert. Wenn der Wert in c kein Vielfaches von acht ist, springt die Routine zu "subtract". Sie subtrahiert 2Ø16 von hl und springt zu "next block". Dies bezweckt, daß hl auf die nächste Menge von acht Zeilen zeigt.

Bei "subtract" springt die Routine zu "next block", falls der Wert (c − 1) kein Vielfaches von acht ist. Andernfalls subtrahiert sie 2Ø16 von de, so daß de auf die nächste Menge von acht Zeilen zeigt. Bei "next block" addiert die Routine 1792 zu hl und springt zu "next line", falls c ein Vielfaches von 64 ist, so daß hl auf den nächsten Block von 64 Zeilen zeigt. Bei "add" wird 1792 zu de addiert, wenn (c − 1) ein Vielfaches von 64 ist; somit zeigt de auf den nächsten Block von 64 Zeilen. Wenn in c keine Eins steht, springt die Routine zu "next line", andernfalls springt sie zu BASIC zurück.

Verschieben um ein Pixel nach unten

Länge: 9Ø
Anzahl der Variablen: Ø
Prüfsumme: 9862

Arbeitsweise

Diese Routine verschiebt den Inhalt der Displaydatei um ein Pixel nach unten.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Keine

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 22527	33 255 87
	ld de, 22271	17 255 86
	ld c, 192	14 192
next line	ld b, 32	6 32
copy byte	ld a, (de)	26
	ld (hl),a	119
	ld a,c	121

Marke	Assemblersprache	einzugebende Zahlen
	cp 2	254 2
	jr nz, next byte	32 2
	sub a	151
	ld (de),a	18
next byte	dec de	27
	dec hl	43
	djnz, copy byte	16 243
	push de	213
	ld de, 224	17 224 Ø
	and a	167
	sbc hl,de	237 82
	ex (sp),hl	227
	and a	167
	sbc hl, de	237 82
	ex de, hl	235
	pop hl	225
	dec c	13
	ld a, c	121
	and 7	23Ø 7
	cp Ø	254 Ø
	jr nz, add	32 8
	push de	213
	ld de, 2Ø16	17 224 7
	add hl,de	25
	pop de	2Ø9
	jr next block	24 11
add	cp 1	254 1
	jr nz, next block	32 7
	push hl	229
	ld hl, 2Ø16	33 224 7
	add hl,de	25
	ex de,hl	235
	pop hl	225
'next block	ld a,c	121
	and 63	23Ø 63
	cp Ø	254 Ø
	jr nz, subtract	32 6
	ld a,h	124
	sub 7	214 7
	ld h,a	1Ø3
	jr next line	24 188
subtract	cp 1	254 1
	jr nz, next line	32 184
	ld a,d	122
	sub 7	214 7
	ld d,a	87
	ld a,c	121
	cp 1	254 1
	jr nz, next line	32 175
	ret	2Ø1

Wie es funktioniert

Die Adresse des letzten Bytes der Displaydatei wird in das Registerpaar hl geladen, die Adresse des Bytes eine Zeile über dem letzten Byte wird in das Registerpaar de geladen. Die Routine lädt die Anzahl der Displayzeilen in Register c. Außerdem lädt sie die Anzahl der Bytes in einer Zeile in Register b, damit es als Zähler eingesetzt werden kann.

Die Routine lädt den Akkumulator mit dem von de adressierten Byte. Dieses wird in den durch hl adressierten Speicherplatz geladen. Der Akkumulator wird mit dem Inhalt des Registers c geladen. Wenn dieses den Wert zwei enthält, zeigt de auf die oberste Zeile des Bildschirms, und deshalb wird es mit null gepoket. de und hl werden dekrementiert, damit sie auf die vorangehenden Bytes zeigen. Der Zähler in Register b wird ebenfalls dekrementiert, und wenn darin keine Null steht, springt die Routine zu "copy byte".

Die Routine subtrahiert 224 sowohl von hl als auch von de, so daß beide Registerpaare auf die vorangehende Displayzeile zeigen. Sie dekrementiert den Zeilenzähler in Register c. Wenn der Wert in c kein Vielfaches von acht ist, springt die Routine zu "add". Dann addiert sie 2Ø16 zu hl und springt zu "next block". Dies bezweckt, daß hl auf den vorangehenden Block von acht Zeilen zeigt.

Bei "add" springt die Routine zu "next block", sofern der Wert von (c − 1) kein Vielfaches von acht ist. Dann addiert sie 2Ø16 zu de, so daß de auf die vorangehende Menge von acht Zeilen zeigt. Bei "next block" wird 1792 von hl subtrahiert, falls c ein Vielfaches von 64 ist, so daß hl auf den vorangehenden Block von 64 Zeilen zeigt, und außerdem springt die Routine zu "next line". Bei "subtract" wird 1792 von de subtrahiert, wenn (c − 1) ein Vielfaches von 64 ist, so daß de auf den vorangehenden Block von 64 Zeilen zeigt. Wenn in c keine Eins steht, springt die Routine zu "next line", andernfalls springt sie zu BASIC zurück.

Display-Routinen

Verschmelzung von Bildern

Länge: 21
Anzahl der Variablen: 1
Prüfsumme: 17Ø9

Arbeitsweise

Diese Routine verschmelzt ein Bild, das im RAM gespeichert wurde (mittels der Routine "Vergrößern und Kopieren des Bildschirms" an anderer Stelle in diesem Buch), mit der aktuellen Bildschirmanzeige. Die Attribute werden nicht verändert.

Variablen

Name	Länge	Speicherzelle	Kommentar
screen store	2	23296	RAM-Adresse des gespeicherten Bildes

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Um Bilder zu verschmelzen, sollte man die Routine wie aufgelistet verwenden. Man kann allerdings interessante Effekte erzielen, wenn man die Anweisung "or (hl) 182" ersetzt durch "xor (hl) 174" oder "and (hl) 166".

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 16384	33 Ø 64
	ld de, (23296)	237 91 Ø 91
	ld bc, 6144	1 Ø 24
next byte	ld a, (de)	26
	or (hl)	182
	ld (hl),a	119

Marke	Assemblersprache	einzugebende Zahlen
	inc hl	35
	inc de	19
	dec bc	11
	ld a, b	120
	or c	177
	jr nz, next byte	32 246
	ret	201

Wie es funktioniert

Die Routine lädt die Adresse der Displaydatei in Registerpaar hl, die Länge des Displays in Registerpaar bc, so daß es als Zähler dienen kann.

Der Akkumulator wird mit dem durch de adressierten Byte geladen, und dieses und das nächste Byte der Displaydatei werden mit "or" verbunden (siehe Glossar). Das Ergebnis wird dann in das Display zurückgeladen.

Die Routine rückt hl und de zur nächsten Position vor und dekrementiert den Zähler. Wenn er nicht null ist, macht sie eine Schleife zurück und wiederholt den Prozeß mit dem nächsten Byte.

Bildschirm-Inversion

Länge: 18
Anzahl der Variablen: 0
Prüfsumme: 1613

Arbeitsweise

Diese Routine invertiert die ganze Displaydatei, d. h. wenn ein Punkt eingeschaltet ist, wird er ausgeschaltet, ist er ausgeschaltet, wird er eingeschaltet.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine kann in Spielprogrammen verwendet werden; man kann damit eine effektive Explosion erzeugen. Der Effekt wird erhöht, wenn man diese Routine mehrmals aufruft, wobei noch irgendeine Geräuschart dazukommen kann.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 16384	33 Ø 64
	ld bc, 6144	1 Ø 24
	ld d, 255	22 255
next byte	ld a,d	122
	sub (hl)	15Ø
	ld (hl),a	119
	inc hl	35
	dec bc	11
	ld a,b	12Ø
	or c	177
	jr nz, next byte	32 247
	ret	2Ø1

Wie es funktioniert

Die Adresse der Displaydatei wird in Registerpaar hl geladen, ihre Länge wird in bc geladen. Die Routine setzt Register d auf 255. Jedesmal wenn sie zu "next byte" zurückspringt, wird der Akkumulator aus d geladen. Hier hätte man auch die Anweisung "ld a, 255" verwenden können, aber das würde ungefähr doppelt so lange dauern als der Befehl "ld a, d". Der Wert des in hl gespeicherten Bytes wird vom Akkumulator subtrahiert, und das Ergebnis wird dann in dasselbe Byte zurückgeladen, wodurch es invertiert wird.

Die Routine inkrementiert hl, damit es auf das nächste Byte zeigt; außerdem dekrementiert sie den Zähler bc. Wenn er nicht null ist, springt die Routine zu "next byte" zurück. Wenn der Zähler null ist, springt sie zu BASIC zurück.

Vertikale Zeichen-Inversion

Länge: 2Ø
Anzahl der Variablen: 1
Prüfsumme: 1757

Arbeitsweise

Diese Routine invertiert ein Zeichen in vertikaler Richtung, zum Beispiel wird ein Aufwärtspfeil zu einem Abwärtspfeil und umgekehrt.

Variablen

Name	Länge	Speicherzelle	Kommentar
chr. start	2	23296	Adresse der Zeichendaten im RAM

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ist in Spielen wie "Minefield" und "Puckman" nützlich, weil Zeichen ihre Richtung ändern können, ohne daß man mehr als ein Zeichen braucht.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23296)	42 Ø 91
	ld d,h	84
	ld e,l	93
	ld b,8	6 8
next byte	ld a, (hl)	126
	inc hl	35
	push af	245
	djnz next byte	16 251
	ld b,8	6 8
replace	pop af	241
	ld (de),a	18
	inc de	19
	djnz replace	16 251
	ret	2Ø1

Wie es funktioniert

Die Routine lädt die Adresse der Zeichendaten im RAM in das Registerpaar hl. Dies kopiert sie dann in de. Register b setzt sie auf 8, weil es als Zähler verwendet werden soll.

Für jedes Byte wird der Akkumulator mit dem aktuellen Wert geladen. hl wird inkrementiert, damit es auf das nächste Byte zeigt, und der Akkumulator wird auf dem Stapel abgelegt. Der Zähler wird dekrementiert. Wenn er nicht null ist, springt die Routine zurück und wiederholt den Vorgang mit dem nächsten Byte. Sie lädt Register b erneut mit 8, damit es wieder als Zähler dienen kann.

Für jedes Byte wird der Akkumulator vom Stapel geholt und in den durch de adressierten Speicherplatz geladen. Die Routine inkrementiert de, damit es auf das nächste Byte zeigt; andererseits dekrementiert sie den Zähler. Wenn dieser nicht null ist, springt sie zurück zu "replace" (deutsch: ersetzen). Dann springt sie zu BASIC zurück.

Horizontale Zeichen-Inversion

Länge: 19
Anzahl der Variablen: 1
Prüfsumme: 1621

Arbeitsweise

Diese Routine invertiert ein Zeichen in horizontaler Richtung, zum Beispiel wird ein Linkspfeil zu einem Rechtspfeil und umgekehrt.

Variablen

Name	Länge	Speicherzelle	Kommentar
chr. start	2	23296	RAM-Adresse der Zeichendaten

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Keine

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zeichen
	ld hl, (23296)	42 Ø 91
	ld a, 8	62 8
next byte	ld b, 8	6 8
next pixel	rr (hl)	2Ø3 3Ø
	rl c	2Ø3 17
	djnz next pixel	16 25Ø
	ld (hl),c	113
	inc hl	35
	dec a	61
	jr nz, next byte	32 243
	ret	2Ø1

Wie es funktioniert

Die Routine lädt die Adresse der Zeichendaten im RAM in das Registerpaar hl und die Anzahl der zu invertierenden Bytes in den Akkumulator. Sie lädt Register b mit der Anzahl der Bits in einem Byte, damit es als Zähler verwendet werden kann.

Das durch hl adressierte Byte wird nach rechts rotiert, so daß das am weitesten rechts stehende Bit in das Übertragsflag kopiert wird. Register c wird nach links rotiert, so daß das Übertragsflag in das am weitesten rechts stehende Bit kopiert wird. Die Routine dekrementiert den in Register b gespeicherten Zähler. Wenn der Zähler nicht null ist, springt sie zu "next pixel" zurück. Sie poket das invertierte Byte, das in Register c gespeichert ist, in seine ursprüngliche Adresse zurück.

Die Routine inkrementiert hl, damit es auf das nächste Byte zeigt, und dekrementiert den Akkumulator. Sofern im Akkumulator keine null steht, springt die Routine zurück zu "next byte".

Dann springt die Routine zu BASIC zurück.

Zeichen-Rotation im Uhrzeigersinn

Länge: 42
Anzahl der Variablen: 1
Prüfsumme: 3876

Arbeitsweise

Diese Routine rotiert ein Zeichen um 9Ø° im Uhrzeigersinn, beispielsweise wird ein Aufwärtspfeil zu einem Rechtspfeil.

Variablen

Name	Länge	Speicherzelle	Kommentar
chr. start	2	23296	Adresse der Zeichendaten im RAM

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine ist sowohl bei Spielen als auch bei ernsthafter Arbeit mit dem Computer, zum Beispiel bei der Beschriftung von Graphen, nützlich.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23296)	42 Ø 91
	ld e, 128	3Ø 128
next bit	push hl	229
	ld c, Ø	14 Ø
	ld b,1	6 1
next byte	ld a, e	123
	and (hl)	166
	cp Ø	254 Ø
	jr z, not set	4Ø 3
	ld a,c	121
	add a,b	128
	ld c,a	79
not set	sla b	2Ø3 32
	inc hl	35
	jr nc, next byte	48 242
	pop hl	225
	push bc	197
	srl e	2Ø3 59
	jr nc, next bit	48 231
	ld de,7	17 7 Ø
	add hl,de	25
	ld b,8	6 8
replace	pop de	2Ø9
	ld (hl), e	115
	dec hl	43
	djnz replace	16 251
	ret	2Ø1

Wie es funktioniert

Jedes Zeichen besteht aus einer Gruppe von 8 x 8 Pixeln; jedes einzelne Pixel kann eingeschaltet (= 1) oder ausgeschaltet (= Ø) sein. Man betrachte ein beliebiges Bit B_2 von Byte B_1 in Abbildung B1. Die Daten in Zelle (B_2, B_1) in der Matrix sind

$$\begin{pmatrix} N_1 & N_3 \\ N_2 & N_4 \end{pmatrix}$$

Dabei gilt:
N_1 = das Byte, bei dem Pixel (B_2, B_1) nach Rotation eingeschoben wird.
N_2 = das Bit in N_1, bei dem es eingeschoben wird.
N_3 = der Wert, den das Bit im Moment darstellt.
N_4 = der Wert des Bits N_2.

Jedes Byte des rotierten Zeichens wird eins nach dem anderen durch Addition der Werte aller Bits N_2, die im neuen Byte sein werden, aufgebaut.

Die Routine lädt die RAM-Adresse des ersten Bytes des Zeichens in Register hl. Sie lädt Register e mit dem Wert des Bytes, bei dem Bit 7 gesetzt ist

und die Bits Ø bis 6 nicht gesetzt sind, d. h. mit 128. Dann speichert sie Register hl auf dem Stapel. Register c wird mit null geladen; zu ihm werden Daten addiert, so daß sich der neue Wert des im Aufbau befindlichen Bytes ergibt. Register b wird mit dem Wert des Bytes geladen, bei dem Bit Ø gesetzt ist, und die Bits 1 bis 7 nicht gesetzt sind, d. h. mit 1.

Der Akkumulator wird mit dem Inhalt des Registers e, d. h. mit (N_3), geladen, dieser durch "and" mit dem Byte, dessen Adresse in hl gespeichert ist, verbunden.

7	6	5	4	3	2	1	Ø	Byte (B_1)
1 128 Ø 1	2 64 Ø 1	3 32 Ø 1	4 16 Ø 1	5 8 Ø 1	6 4 Ø 1	7 2 Ø 1	8 1 Ø 1	1
1 128 1 2	2 64 1 2	3 32 1 2	4 16 1 2	5 8 1 2	6 4 1 2	7 2 1 2	8 1 1 2	2
1 128 2 4	2 64 2 4	3 32 2 4	4 16 2 4	5 8 2 4	6 4 2 4	7 2 2 4	81 2 4	3
1 128 3 8	2 64 3 8	3 32 3 8	4 16 3 8	5 8 3 8	6 4 3 8	7 2 3 8	8 1 3 8	4
1 128 4 16	2 64 4 16	3 32 4 16	4 16 4 16	5 8 4 16	6 4 4 16	7 2 4 16	8 1 4 16	5
1 128 5 32	2 64 5 32	3 32 5 32	4 16 5 32	5 8 5 32	6 4 5 32	7 2 5 32	8 1 5 32	6
1 128 6 64	2 64 6 64	3 32 6 64	4 16 6 64	5 8 6 64	6 4 6 64	7 2 6 64	8 1 6 64	7
1 128 7 128	2 64 7 128	3 32 7 128	4 16 7 128	5 8 7 128	6 4 7 128	7 2 7 128	8 1 7 128	8

Bit (B_2)

Abbildung B1
Schlüssel zur Routine "Zeichenrotation im Uhrzeigersinn"

Wenn das Ergebnis null ist, springt die Routine zu "not set" (deutsch: nicht gesetzt), weil das Pixel, das von e und hl adressiert wird, ausgeschaltet ist. Wenn es eingeschaltet ist, wird der Akkumulator mit dem momentanen Wert des Bytes, nämlich (N_1), geladen. Register b, d. h. (N_4), wird zum Akkumulator

addiert, und dieser wird in c geladen. Register b wird dann so eingestellt, daß es auf das nächste Bit von N_1 zeigt. Die Routine erhöht hl, damit es auf das nächste Byte, d. h. (B_1), zeigt. Wenn das Byte N_1 nicht vollständig ist, springt die Routine zurück zu "next byte".

Registerpaar hl wird vom Stapel zurückgeholt, damit es wieder auf das erste Byte des Zeichens zeigt. Die Routine speichert bc auf dem Stapel, wobei in ihm der Wert des letzten Bytes steht, welches in c noch vervollständigt werden muß. Das Register e wird so eingestellt, daß es das nächste Bit jedes Bytes adressiert. Wenn die Rotation noch nicht vollständig ist, springt die Routine zu "next bit" zurück.

Sie lädt de mit 7 und addiert dieses zu hl, so daß hl auf das letzte Datenbyte zeigt. Register b wird mit der Anzahl derjenigen Bytes geladen, die vom Stapel zurückgeholt werden müssen. Für jedes Byte wird der neue Wert in e kopiert, und dies wird in hl eingepoket. Die Routine dekrementiert hl, damit es auf das nächste Byte zeigt. Sie dekrementiert ebenfalls den Zähler in Register b. Wenn er nicht null ist, springt die Routine zu "replace".

Dann springt die Routine zu BASIC zurück.

Attributsänderung

Länge: 21
Anzahl der Variablen: 2
Prüfsumme: 1952

Arbeitsweise

Diese Routine ändert die Attribute aller Bildschirmzeichen in einer festgesetzten Weise. Zum Beispiel kann die Vordergrundfarbe (INK) geändert werden; der ganze Bildschirm kann auf "Blinken" (FLASH) umgestellt werden usw.

Variablen

Name	Länge	Speicherzelle	Kommentar
data saved	1	23296	Nicht zu ändernde Attributbits
new data	1	23297	Ins Attribut einzuschiebende neue Bits

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Bei jedem einzelnen Zeichen können einzelne Bits der Attribute durch die Maschinencode-Befehle "and" und "or" geändert werden.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 22528	33 0 88
	ld bc, 768	1 0 3
	ld de, (23296)	237 91 0 91
next byte	ld a, (hl)	126
	and e	163
	or d	178
	ld (hl), a	119
	inc hl	35
	dec bc	11
	ld a,b	120
	or c	177
	jr nz, next byte	32 246
	ret	201

Wie es funktioniert

Die Adresse des Attributbereichs wird in das Registerpaar hl geladen. Die Anzahl der Zeichen im Display wird in Registerpaar bc geladen. Die Routine lädt Register d mit dem Wert "new data", Register e mit dem Wert "data saved".

Die Routine lädt den Akkumulator mit dem von hl adressierten Byte, und sie stellt die Bits entsprechend den Werten der Register d und e ein. Das Ergebnis wird in hl zurückgepoket. hl wird inkrementiert, damit es auf das nächste Byte zeigt, und der Zähler in bc wird dekrementiert. Wenn in bc keine Null steht, springt die Routine zu "next byte".

Die Routine springt dann zu BASIC zurück.

Attributswechsel

Länge: 22
Anzahl der Variablen: 2
Prüfsumme: 1825

Arbeitsweise

Diese Routine durchsucht den Attributbereich nach einem bestimmten Wert und ersetzt ihn, jedesmal wenn er auftritt, durch einen anderen Wert.

Variablen

Name	Länge	Speicherzelle	Kommentar
old value	1	23296	Wert des zu ersetzenden Bytes
new value	1	23297	Neuer Wert des ersetzten Bytes

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine kann man verwenden, um Graphikzeichen- und Textbereiche hell hervorzuheben.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, 22528	33 Ø 88
	ld bc, 768	1 Ø 3
	ld de, (23296)	237 91 Ø 91
next byte	ld a, (hl)	126
	cp e	187
	jr nz, no change	32 1
	ld (hl), d	114
no change	inc hl	35
	dec bc	11
	ld a,b	12Ø
	or c	177
	jr nz, next byte	32 245
	ret	2Ø1

Wie es funktioniert

Die Adresse des Attributbereichs wird in Registerpaar hl geladen. Die Anzahl der Bildschirmzeichen wird in bc geladen. Die Routine lädt Register e mit "old value" (deutsch: alter Wert), Register d mit "new value" (deutsch: neuer Wert).

Das von Registerpaar hl adressierte Byte lädt sie in den Akkumulator. Wenn im Akkumulator der Wert des Registers e steht, wird das von hl adressierte Byte mit dem Inhalt von Register d gepoket. Dann inkrementiert die Routine hl, damit es auf das nächste Byte zeigt. Außerdem dekrementiert sie den Zähler in bc. Wenn in bc keine Null steht, springt sie zu "next byte".

Die Routine springt dann zu BASIC zurück.

Ausfüllen eines Gebiets

Länge: 263
Anzahl der Variablen: 2
Prüfsumme: 26647

Arbeitsweise

Diese Routine "bedeckt" eine Bildschirmfläche, die von einer Zeile von Pixeln am Rand des Bildschirms begrenzt wird.

Variablen

Name	Länge	Speicherzelle	Kommentar
x-co-ord	1	23296	x-Koordinate der Anfangsposition
y-co-ord	1	23297	y-Koordinate der Anfangsposition

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn die y-Koordinate größer als 175 ist oder wenn POINT (x, y) = 1, springt die Routine sofort zu BASIC zurück.

Kommentare

Diese Routine ist nicht verschiebbar; die Startadresse ist 31955. Wenn man diese Routine auf eine andere Adresse kopieren will, so verwende man die Methode, die in der Routine "Umnumerieren" dargestellt wird. Sofern als Startadresse dieser Routine 31955 und als Startadresse von "Umnumerieren" 32218 verwendet wird, können beide Routinen gleichzeitig im RAM gespeichert werden. Man braucht einen großen RAM-Bereich, wenn man unregelmäßig geformte Flächen ausfüllen will. Wenn dieser nicht zur Verfügung steht, kann es passieren, daß die Routine zusammenbricht.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23296)	42 Ø 91
	ld a,h	124
	cp 176	254 176
	ret nc	2Ø8
	call subroutine	2Ø5 143* 125*
	and (hl)	166
	cp Ø	254 Ø
	ret nz	192
	ld bc, 65536	1 255 255
	push bc	197
right	ld hl, (23296)	42 Ø 91
	call subroutine	2Ø5 143* 125*
	and (hl)	166
	cp Ø	254 Ø
	jr nz, left	32 9
	ld hl, (23296)	42 Ø 91
	inc l	44
	ld (23296), hl	34 Ø 91
	jr nz, right	32 236
left	ld de, Ø	17 Ø Ø
	ld hl, (23296)	42 Ø 91
	dec l	45
	ld (23296), hl	34 Ø 91
plot	ld hl, (23296)	42 Ø 91
	push hl	229
	call subroutine	2Ø5 143* 125*
	or (hl)	182
	ld (hl),a	119
	pop hl	225
	ld a,h	124
	cp 175	254 175
	jr z, down	4Ø 44
	ld a,e	123
	cp Ø	254 Ø
	jr nz, reset	32 16
	inc h	36
	call subroutine	2Ø5 143* 125*
	and (hl)	166
	cp Ø	254 Ø
	jr nz, reset	32 7
	ld hl, (23296)	42 Ø 91
	inc h	36
	push hl	229
	ld e,l	3Ø 1
reset	ld hl, (23296)	42 Ø 91
	ld a,e	123
	cp 1	254 1
	jr nx, down	32 15

Marke	Assemblersprache	einzugebende Zahlen
	inc h	36
	call subroutine	2∅5 143* 125*
	and (hl)	166
	cp ∅	254 ∅
	jr z, down	4∅ 6
	ld e, ∅	3∅ ∅
	jr down	24 2
long jump	jr right	24 167
down	ld hl, (23296)	42 ∅ 91
	ld a,h	124
	cp ∅	254 ∅
	jr z, next pixel	4∅ 4∅
	ld a,d	122
	cp ∅	254 ∅
	jr nz, restore	32 16
	dec h	37
	call subroutine	2∅5 143* 125*
	and (hl)	166
	cp ∅	254 ∅
	jr nz, restore	32 7
	ld hl, (23296)	42 ∅ 91
	dec h	37
	push hl	229
	ld d,l	22 1
restore	ld a,d	122
	cp 1	254 1
	jr nz, next pixel	32 14
	ld hl, (23296)	42 ∅ 91
	dec h	37
	call subroutine	2∅5 143* 125*
	and (hl)	166
	cp ∅	254 ∅
	jr z, next pixel	4∅ 2
	ld d, ∅	22 ∅
next pixel	ld hl, (23296)	42 ∅ 91
	ld a,l	125
	cp ∅	254 ∅
	jr z, retrieve	4∅ 12
	dec l	45
	ld (23296),hl	34 ∅ 91
	call subroutine	2∅5 143* 125*
	and (hl)	166
	cp ∅	254 ∅
	jr z, plot	4∅ 129
retrieve	pop hl	225
	ld (23296),hl	34 ∅ 91
	ld a, 255	62 255
	cp h	188
	jr nz, long jump	32 177
	cp l	189

Marke	Assemblersprache	einzugebende Zahlen
	jr nz, long jump	32 174
	ret	2∅1
subroutine	push bc	197
	push de	213
	ld a, 175	62 175
	sub h	148
	ld h,a	1∅3
	push hl	229
	and 7	23∅ 7
	add a, 64	198 64
	ld c,a	79
	ld a,h	124
	rra	2∅3 31
	rra	2∅3 31
	rra	2∅3 31
	and 31	23∅ 31
	ld b, a	71
	and 24	23∅ 24
	ld d,a	87
	ld a,h	124
	and 192	23∅ 192
	ld e,a	95
	ld h,c	97
	ld a,l	125
	rra	2∅3 31
	rra	2∅3 31
	rra	2∅3 31
	and 31	23∅ 31
	ld l,a	111
	ld a,e	123
	add a,b	128
	sub d	146
	ld e,a	95
	ld d, ∅	22 ∅
	push hl	229
	push de	213
	pop hl	225
	add hl,hl	41
	add hl,hl	41
	add hl,hl	41
	add hl,hl	41
	add hl,hl	41
	pop de	2∅9
	add hl,de	25
	pop de	2∅9
	ld a,e	123
	and 7	23∅ 7
	ld b, a	71
	ld a, 8	62 8
	sub b	144

Marke	Assemblersprache	einzugebende Zahlen
	ld b,a	71
	ld a,1	62 1
rotate	add a,a	135
	djnz rotate	16 253
	rra	2Ø3 31
	pop de	2Ø9
	pop bc	193
	ret	2Ø1

Wie es funktioniert

Diese Routine zeichnet waagrechte Zeilen von nebeneinanderliegenden Pixeln, "RUNS" genannt, zwischen Bereiche, die von aufleuchtenden Pixeln begrenzt werden. Jeder RUN wird durch "Stapeln" aufbewahrt beziehungsweise gespeichert; und zwar werden die Koordinaten des am weitesten rechts stehenden RUN-Pixels auf dem Stapel abgelegt. Die Routine trägt dann jeden RUN ein, wobei sie bei den angegebenen Anfangskoordinaten beginnt. Dabei vermerkt sie die Positionen jeglicher uneingetragener RUNS darüber oder darunter. Ist ein RUN beendet, wird das letzte vermerkte Koordinatenpaar vom Stapel zurückgeholt und der entsprechende RUN eingetragen. Dieser Prozeß wird wiederholt, bis es keine uneingetragenen RUNS mehr gibt.

Abbildung B2 stellt das Verfahren dar. Die Quadrate bedeuten aufleuchtende Pixel, X kennzeichnet die Startposition innerhalb der zu bedeckenden Fläche und * markiert die am weitesten rechts stehenden RUN-Pixel.

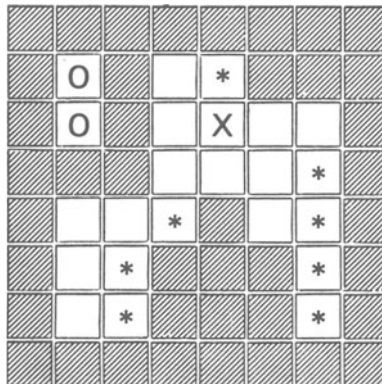


Abbildung B2
*Eine Darstellung des Verfahrens zum Ausfüllen eines Gebietes. Graue Quadrate leuchten schon auf und definieren das Gebiet, das bedeckt werden soll. X ist die Startposition, ein * ist jeweils der Anfang eines RUNS und die Felder mit O werden nicht ausgefüllt.*

Die Routine bedeckt die waagrechte Zeile, in der die Startposition liegt, und speichert die Anfangspositionen der RUNS in den Zeilen unmittelbar darüber und darunter auf dem Stapel. Als nächstes bedeckt sie die Zeile darüber und dann diejenige darunter. Im letzten Fall vermerkt sie, daß zwei weitere RUNS in der Zeile darunter anfangen usw. Jede Position innerhalb der zu bedeckenden Fläche kann als Startposition gewählt werden. Man beachte jedoch, daß die zwei mit Nullen markierte Pixel nie erreicht werden, weil sie von der Fläche, die ausgefüllt wird, getrennt sind.

Die vorgegebene y-Koordinate wird in Register h geladen, die x-Koordinate in Register l. Wenn der Wert der y-Koordinate größer als 175 ist, springt die Routine zu BASIC zurück. Dann wird die "subroutine" aufgerufen, welche die Speicheradresse des Bits (x, y) ausgibt. Wenn dieses Bit "eingeschaltet" ist, springt die Routine zu BASIC zurück.

Die Zahl 65535 wird auf dem Stapel abgelegt und markiert dadurch den ersten gespeicherten Wert. Wird zu einem späteren Zeitpunkt eine Zahl vom Stapel zurückgeholt, so wird sie als Koordinatenpaar verwendet. Ist die Zahl allerdings 65535 selbst, springt die Routine zu BASIC zurück, weil sie dann fertig ist.

Die y-Koordinate wird in Register h, die x-Koordinate in Register l geladen. Die "subroutine" wird aufgerufen, welche die Adresse des Bits (x, y) in hl stellt. Wenn dieses Bit gesetzt ist, springt die Routine zu "left" (deutsch: links). Andernfalls inkrementiert sie die x-Koordinate und, sofern x ungleich 256 ist, springt sie zu "right" (deutsch: rechts).

Bei "left" wird de auf null gesetzt. Die Register d und e werden als Flags benutzt, d für abwärts, e für aufwärts. Die x-Koordinate wird dekrementiert. Die Subroutine wird aufgerufen, und der Punkt (x, y) wird gezeichnet. Wenn die y-Koordinate 175 ist, springt die Routine zu "down" (deutsch: abwärts). Ist das "Aufwärts-Flag" auf eins gesetzt, springt sie zu "reset". Wenn das Bit (x, y + 1) ausgeschaltet ist, werden die Werte von x und y + 1 auf dem Stapel gespeichert, und das "Aufwärts-Flag" wird auf eins gesetzt.

Wenn bei "reset" das "Aufwärts-Flag" auf null steht, springt die Routine zu "down". Steht das Bit (x, y + 1) auf eins, wird das "Aufwärts-Flag" auf null gesetzt. Wenn bei "down" die y-Koordinate null ist, springt die Routine zu "next pixel". Ist das "Abwärts-Flag" auf eins gesetzt, springt sie zu "restore". Sofern das Bit (x, y −1) ausgeschaltet ist, werden die Werte von x und y − 1 auf dem Stapel gespeichert und das "Abwärts-Flag" auf eins gesetzt.

Bei "restore" springt die Routine zu "next pixel", wenn das "Abwärts-Flag" auf null steht. Ist das Bit (x, y − 1) eingeschaltet, wird das "Abwärts-Flag" auf null gesetzt. Bei "next pixel" springt die Routine zu "retrieve" (deutsch: zurückholen), sofern die x-Koordinate null ist. Die x-Koordinate wird dekrementiert, und wenn das neue Bit (x, y) ausgeschaltet ist, wird zu "plot" (deutsch: aufzeichnen) gesprungen. Bei "retrieve" wird eine x- und eine y-Koordinate vom Stapel zurückgeholt. Sind x und y beide gleich 255, springt die Routine zu BASIC zurück, da das Gebiet schon vollständig überdeckt wurde. Andernfalls springt die Routine zu "right" zurück.

Die Subroutine muß die Adresse des Bits (x, y) im Speicher berechnen. In BASIC wäre diese Adresse

```
16384 + INT (Z/8) + 256 * (Z – 8 * INT (Z/8))
+32 * (64 * INT (Z/64) + INT (Z/8 – 8 * INT (Z/64))
wobei Z = 175 – Y
```

Die Registerpaare bc und de werden auf dem Stapel abgelegt. Der Akkumulator wird mit 175 geladen, und die y-Koordinate wird davon abgezogen. Die

Routine stellt das Resultat in Register h zurück. Dann speichert sie hl auf dem Stapel. Sie setzt die linken fünf Bits des Akkumulators auf null und zählt die Zahl 64 dazu. Das Ergebnis wird in Register c kopiert. Wenn dies mit 256 multipliziert wird, ergibt sich 16384 + 256 * (Z – 8 * INT (Z/8)). Der Akkumulator wird mit Z geladen, und dies wird durch acht geteilt; das Resultat, INT (Z/8), wird in Register b gestellt. Wenn man die drei am weitesten rechts stehenden Bits auf null setzt, erhält man das Ergebnis 8 * INT (Z/64); dieses wird in Register d geladen.

Die Routine lädt Z in den Akkumulator und setzt die sechs am weitesten rechts stehenden Bits auf null. Dies ergibt den Wert 64 * INT (Z/64). Er wird in Register e geladen. Der Wert in Register c wird in h kopiert. Die Routine lädt den Akkumulator mit der x-Koordinate, teilt dies durch acht und kopiert das Ergebnis in l hinein.

Der Akkumulator wird dann mit dem Wert in e geladen, und der Inhalt von b wird dazu addiert. Davon subtrahiert die Routine den Wert in d. Sie kopiert das Resultat in de. Das Registerpaar hl wird auf dem Stapel abgelegt und dann mit dem Wert in de geladen. Dies wird mit 32 multipliziert, de wird vom Stapel zurückgeholt und zu hl addiert. Damit steht in hl jetzt die Adresse des Bits (x, y).

Die Routine lädt den Akkumulator mit dem ursprünglichen Wert von x. Wenn man die linken fünf Bits auf null setzt, erhält man den Wert x – 8 * INT (x/8). Das Register b wird dann mit acht minus dem Wert des Akkumulators geladen, damit es als Zähler dienen kann. Der Akkumulator wird auf eins gesetzt, und dies wird (b – 1)-mal mit zwei multipliziert.

Zu diesem Zeitpunkt sollte nur ein einziges Bit im Akkumulator gesetzt sein, was dem Bit (x, y), welches von hl adressiert wird, entspricht. de und bc werden dann vom Stapel zurückgeholt, und die Subroutine springt in die Hauptroutine zurück.

Zeichnen von Figuren

Länge: 196
Anzahl der Variablen: 2
Prüfsumme: 2Ø278

Arbeitsweise

Diese Routine zeichnet eine Figur irgendeiner Größe auf den Bildschirm.

Variablen

Name	Länge	Speicherzelle	Kommentar
X start	1	23296	X-Koordinate des ersten Pixels
Y start	1	23297	Y-Koordinate des ersten Pixels

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn A$ nicht existiert, Länge null hat oder keine Information über die Figur enthält, kehrt die Routine sofort zu BASIC zurück. Dies geschieht auch, wenn "Y start" größer als 175 ist.

Kommentare

Dies ist eine nützliche Methode zum Speichern von Figuren, die schnell auf dem Bildschirm gezeichnet werden sollen.

Die Routine wird folgendermaßen verwendet:
(i) LET A$ = "Information über die Figur"
(ii) POKE 23296, X-Koordinate des ersten Pixels
(iii) POKE 23297, Y-Koordinate des ersten Pixels
(iv) RAND USR Adresse

Die Information über die Figur ist eine Zeichenkette, wobei die Zeichen folgende Bedeutung haben:
"Ø" Punkt zeichnen
"5" X-Koordinate verringern
"6" Y-Koordinate verringern
"7" Y-Koordinate erhöhen
"8" X-Koordinate erhöhen

Andere Zeichen werden übergangen.

Die Routine ist auf "zyklisches Umhüllen" eingerichtet, d. h. wenn die X-Koordinate aus dem Bildschirm links herausfällt, taucht sie ganz rechts wieder auf usw.

Will man erreichen, daß die Routine einen anderen String als A$ verwendet, so muß man 65* ersetzen durch den Code des Großbuchstabens des Stringnamens.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23627)	42 75 92
next variable	ld a, (hl)	126
	cp 128	254 128
	ret z	2ØØ
	bit 7,a	2Ø3 127
	jr nz, for next	32 23
	cp 96	254 96
	jr nc, number	48 11

Marke	Assemblersprache	einzugebende Zahlen
	cp 65	254 65*
	jr z, found	4Ø 35
string	inc hl	35
	ld e, (hl)	94
	inc hl	35
	ld d, (hl)	86
add	add hl,de	25
	jr increase	24 5
number	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
increase	inc hl	35
	jr next variable	24 225
for next	cp 224	254 224
	jr c, next bit	56 5
	ld de, 18	17 18 Ø
	jr add	24 236
next bit	bit 5,a	2Ø3 111
	jr z, string	4Ø 228
next byte	inc hl	35
	bit 7,(hl)	2Ø3 126
	jr z, next byte	4Ø 251
	jr number	24 228
found	inc hl	35
	ld c, (hl)	78
	inc hl	35
	ld b, (hl)	7Ø
	inc hl	35
	ex de,hl	235
	ld a, (23297)	58 1 91
	cp 176	254 176
	ret nc	2Ø8
again	ld hl, (23296)	42 Ø 91
	ld a,b	12Ø
	or c	177
	ret z	2ØØ
	dec be	11
	ld a, (de)	26
	inc de	19
	cp 48	254 48
	jr nz, not plot	32 78
	push bc	197
	push de	213
	ld a, 175	62 175
	sub h	148
	ld h,a	1Ø3
	push hl	229
	and 7	23Ø 7

Marke	Assemblersprache	einzugebende Zahlen
	add a,64	198 64
	ld c,a	79
	ld a,h	124
	rra	2∅3 31
	rra	2∅3 31
	rra	2∅3 31
	and 31	23∅ 31
	ld b,a	71
	and 24	23∅ 24
	ld d,a	87
	ld a,h	124
	and 192	23∅ 192
	ld e,a	95
	ld h,c	97
	ld a,l	125
	rra	2∅3 31
	rra	2∅3 31
	rra	2∅3 31
	and 31	23∅ 31
	ld l,a	111
	ld a,e	123
	add a,b	128
	sub d	146
	ld e,a	95
	ld d, ∅	22 ∅
	push hl	229
	push de	213
	pop hl	225
	add hl,hl	41
	add hl,hl	41
	add hl,hl	41
	add hl,hl	41
	add hl,hl	41
	pop de	2∅9
	add hl,de	25
	pop de	2∅9
	ld a,e	123
	and 7	23∅ 7
	ld b,a	71
	ld a, 8	62 8
	sub b	144
	ld b,a	71
	ld a,1	62 1
rotate	add a,a	135
	djnz rotate	16 253
	rra	2∅3 31
	pop de	2∅9
	pop bc	193
	or (hl)	182
	ld (hl),a	119

Marke	Assemblersprache	einzugebende Zahlen
here	jr again	24 165
not plot	cp 53	254 53
	jr nz, down	32 1
	dec l	45
down	cp 54	254 54
	jr nz, up	32 8
	dec h	37
	ld a,h	124
	cp 255	254 255
	jr nz, save	32 19
	ld h, 175	38 175
up	cp 55	254 55
	jr nz, right	32 8
	inc h	36
	ld a,h	124
	cp 176	254 176
	jr nz, save	32 7
	ld h, ∅	38 ∅
right	cp 56	254 56
	jr nz, save	32 1
	inc l	44
save	ld (23296),hl	34 ∅ 91
	jr here	24 215

Wie es funktioniert

Die Adresse des Strings A$ wird gefunden durch Anwendung einer umgearbeiteten Version des ersten Teils der Routine "Instr$".

Die Länge des Strings wird in bc geladen, und die Adresse des ersten Zeichens von A$ wird in de geladen. Die Routine setzt den Akkumulator auf den Wert "Y start", und wenn dieser größer als 175 ist, springt sie zu BASIC zurück. Die Y-Koordinate wird in Register h geladen, die X-Koordinate in Register l. Wenn der Wert des Registerpaars bc null ist, springt die Routine zu BASIC zurück, weil das Ende des Strings erreicht wurde. Sie dekrementiert bc und zeigt damit an, daß mit einem anderen Zeichen gearbeitet wurde. Das nächste Zeichen wird in den Akkumulator geladen, und de wird inkrementiert, damit es auf das nächste Byte zeigt. Wenn im Akkumulator nicht 48 steht, springt die Routine zu "not plot". Der Punkt (X, Y) wird mit Hilfe der "subroutine" aus der Routine "Ausfüllen eines Gebiets" gezeichnet. Die Routine springt dann zurück zu "again" (deutsch: nochmals).

Bei "not plot" dekrementiert die Routine die X-Koordinate, sofern im Akkumulator 53 steht. Bei "down" springt sie, falls im Akkumulator nicht 54 steht, zu "up" (deutsch: aufwärts). Sie dekrementiert die Y-Koordinate, und wenn dann −1 darin steht, setzt sie die Y-Koordinate auf 175.

Bei "up" springt die Routine zu "right", sofern im Akkumulator nicht 55 steht. Sie inkrementiert die Y-Koordinate. Wenn diese 176 ist, wird die Y-Koordinate auf null gesetzt. Bei "right" inkrementiert sie die X-Koordinate, falls im Akkumulator 56 steht. Bei "save" werden die X- und die Y-Koordinate in den Speicher gebracht und die Routine macht eine Schleife zu "here".

Vergrößern und Kopieren des Bildschirms

Länge: 335
Anzahl der Variablen: 8
Prüfsumme: 33663

Arbeitsweise

Diese Routine kopiert einen Displayausschnitt auf eine andere Fläche auf dem Bildschirm. Dabei wird die Kopie in x- und y-Richtung vergrößert.

Variablen

Name	Länge	Speicherzelle	Kommentar
upper y co-ord	1	23296	y-Koordinate der obersten Zeile
lower y co-ord	1	23297	y-Koordinate der untersten Zeile
right x co-ord	1	23298	x-Koordinate der am weitesten rechts stehenden Spalte
left x co-ord	1	23299	x-Koordinate der am weitesten links stehenden Spalte
horizontal scale	1	233ØØ	Vergrößerung in x-Richtung
vertical scale	1	233Ø1	Vergrößerung in y-Richtung
new left co-ord	1	233Ø2	x-Koordinate der am weitesten links stehenden Spalte der Fläche, auf die kopiert werden soll
new lower co-ord	1	233Ø3	y-Koordinate der untersten Zeile der Fläche, auf die kopiert werden soll

Aufruf

RAND USR Adresse

Fehlerüberwachung

Die Routine springt sofort zu BASIC zurück, wenn eine der folgenden Bedingungen erfüllt ist:
(i) "horizontal scale" = Ø
(ii) "vertical scale" = Ø
(iii) "upper y co-ord" größer als 175
(iv) "new lower co-ord" größer als 175
(v) "lower y co-ord" größer als "upper y co-ord"
(vi) "left x co-ord" größer als "right x co-ord"

Da die Routine nicht zu lang werden soll, wird allerdings nicht überprüft, ob der zu kopierende Auschnitt auch tatsächlich auf den Bildschirm paßt. Wenn das nicht geht, kann es passieren, daß die Routine zusammenbricht. Die Routine benötigt einen großen zusätzlichen RAM-Bereich, und wenn dieser nicht verfügbar ist, ist ein Zusammenbruch ebenfalls möglich.

Kommentare

Diese Routine ist nicht verschiebbar wegen der "Plot/Point" Subroutine. Sie steht in Adresse 65Ø33 und kann daher nur auf Maschinen mit 48K-RAM gefahren werden. Die Routine kann in eine neue Position im Speicher gebracht werden mittels des Verfahrens, das bei der Routine "Umnumerieren" angegeben ist. Wenn jedoch große Bildschirmflächen kopiert werden sollen, braucht man viel zusätzlichen Platz im RAM, und deshalb sollte die Startadresse so hoch wie möglich sein.

Wenn die kopierte Displayfläche dieselbe Größe wie das Original haben soll, muß man die Variablen für die Maßstäbe (englisch: scales), d. h. "horizontal scale" und "vertical scale", auf eins setzen. Um die Größe zu verdoppeln, muß man sie mit zwei laden, um die Größe zu verdreifachen, mit drei usw.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld ix, 23296	221 33 Ø 91
	ld a, 175	62 175
	cp (ix + Ø)	221 19Ø Ø
	ret c	216
	cp (ix + 7)	221 19Ø 7
	ret c	216
	sub a	151
	cp (ix + 4)	221 19Ø 4
	ret z	2ØØ
	cp (ix + 5)	221 19Ø 5
	ret z	2ØØ
	ld hl, (23296)	42 Ø 91
	ld b, 1	69
	ld a,l	125
	sub h	148
	ret c	216
	ld (23298),a	5Ø Ø 91
	ld e,a	95
	ld hl, (23298)	42 2 91
	ld c,l	77
	ld a,l	125
	sub h	148
	ret c	216
	ld (23298),a	5Ø 2 91
	push bc	197
	ld l,a	111

Marke	Assemblersprache	einzugebende Zahlen
	ld h, Ø	38 Ø
	inc hl	35
	push hl	229
	pop bc	193
	inc e	28
add	dec e	29
	jr z, remainder	4Ø 3
	add hl,bc	9
	jr add	24 25Ø
remainder	ld a,l	125
	and 15	23Ø 15
	ld b,a	71
	pop hl	225
	ld c,l	77
	jr nz, save	32 2
full	ld b,16	6 16
save	push hl	229
	call subroutine	2Ø5 13* 255*
	and (hl)	166
	jr z, off	4Ø 2
	ld a, 1	62 1
off	pop hl	225
	rra	2Ø3 31
	rl e	2Ø3 19
	rl d	2Ø3 18
	ld a,l	125
	cp (ix + 3)	221 19Ø 3
	jr z, next row	4Ø 6
	dec l	45
next bit	djnz save	16 231
	push de	213
	jr full	24 226
next row	ld l,c	1Ø5
	ld a,h	124
	cp (ix + 1)	221 19Ø 1
	jr z, copy	4Ø 3
	dec h	37
	jr next bit	24 241
copy	push de	213
	ld b, Ø	6 Ø
	ld h,b	96
	ld l,b	1Ø4
reset	ld (233Ø6),hl	34 1Ø 91
	ld a,b	12Ø
	or a	183
	jr nz, retrieve	32 3
	pop de	2Ø9
	ld b, 16	6 16
retrieve	sub a	151
	dec b	5

Marke	Assemblersprache	einzugebende Zahlen
	rr d	2Ø3 26
	rr e	2Ø3 27
	rl a	2Ø3 23
	push de	213
	push bc	197
	push af	245
	ld h,1	38 1
loop	ld, l,1	46 1
preserve	ld (233Ø4),hl	34 8 91
	ld a, (233Ø7)	58 11 91
	ld hl, Ø	33 Ø Ø
	ld de, (233Ø1)	237 91 5 91
	ld d,l	85
multiply	or a	183
	jr z, calculate	4Ø 6
	add hl,de	25
	dec a	61
	jr multiply	24 249
long jump	jr reset	24 2Ø8
calculate	ld a, (233Ø3)	58 7 91
	add a,l	133
	ld hl, (233Ø4)	42 8 91
	add a,l	133
	dec a	61
	push af	245
	ld a, (233Ø6)	58 1Ø 91
	ld hl, Ø	33 Ø Ø
	ld de, (233ØØ)	237 91 4 91
	ld d, l	85
repeat	or a	183
	jr z, continue	4Ø 4
	add hl,de	25
	dec a	61
	jr repeat	24 249
continue	ld a, (233Ø2)	58 6 91
	add a,l	133
	ld hl, (233Ø5)	42 9 91
	add a, l	133
	dec a	61
	ld l,a	111
	pop af	241
	ld h,a	1Ø3
	pop af	241
	push af	245
	or a	183
	jr nz, Plot	32 7
	call subroutine	2Ø5 13* 255*
	cpl	47
	and (hl)	166
	jr Poke	24 4

Marke	Assemblersprache	einzugebende Zahlen
Plot	call subroutine	2∅5 13* 255*
	or (hl)	182
Poke	ld (hl),a	119
	ld hl, (233∅4)	42 8 91
	inc l	44
	ld a, (233∅1)	58 5 91
	inc a	6∅
	cp l	189
	jr nz, preserve	32 165
	inc h	36
	ld a, (233∅∅)	58 4 91
	inc a	6∅
	cp h	188
	jr nz, loop	32 155
	pop af	241
	pop bc	193
	pop de	2∅9
	ld hl, (233∅6)	42 1∅ 91
	Inc l	44
	ld a, (23298)	58 2 91
	inc a	6∅
	cp l	189
	jr nz, long jump	32 164
	ld l, ∅	46 ∅
	inc h	36
	ld a, (23296)	58 ∅ 91
	inc a	6∅
	cp h	188
	jr nz, long jump	32 154
	ret	2∅1
subroutine	push bc	197
	push de	213
	ld a, 175	62 175
	sub h	148
	ld h,a	1∅3
	push hl	229
	and 7	23∅ 7
	add a, 64	198 64
	ld c,a	79
	ld a,h	124
	rra	2∅3 31
	rra	2∅3 31
	rra	2∅3 31
	and 31	23∅ 31
	ld b,a	71
	and 24	23∅ 24
	ld d,a	87
	ld a,h	124
	and 192	23∅ 192
	ld e,a	95

Marke	Assemblersprache	einzugebende Zahlen
	ld h, c	97
	ld a,l	125
	rra	2∅3 31
	rra	2∅3 31
	rra	2∅3 31
	and 31	23∅ 31
	ld l,a	111
	ld a,e	123
	add a,b	128
	sub d	146
	ld e,a	95
	ld d, ∅	22 ∅
	push hl	229
	push de	213
	pop hl	225
	add hl,hl	41
	add hl,hl	41
	add hl,hl	41
	add hl,hl	41
	add hl,hl	41
	pop de	2∅9
	add hl,de	25
	pop de	2∅9
	ld a,e	123
	and 7	23∅ 7
	ld b,a	71
	ld a,8	62 8
	sub b	144
	ld b,a	71
	ld a, 1	62 1
rotate	add a,a	135
	djnz rotate	16 253
	rra	2∅3 31
	pop de	2∅9
	pop bc	193
	ret	2∅1

Wie es funktioniert

Die Adresse des Druckerpuffers wird in ix geladen, damit man einen Zeiger auf die Variablen zur Verfügung hat. Wenn die obere y-Koordinate ("upper y co-ord") oder die neue untere y-Koordinate ("new lower co-ord") größer als 175 ist, springt die Routine zu BASIC zurück. Wenn der horizontale Maßstab oder der vertikale Maßstab null ist, springt die Routine ebenfalls zu BASIC zurück.

Die untere y-Koordinate ("lower y co-ord") wird in das Register h geladen, und die obere y-Koordinate wird in Register l geladen. Register l wird dann sowohl in Register b als auch in den Akkumulator geladen. Register h wird vom Akkumulator subtrahiert; die Routine springt zu BASIC zurück, wenn das Ergebnis negativ ist.

Der Wert des Akkumulators wird dann in Speicherzelle 23298 gestellt, damit man einen Zähler zur Verfügung hat. Darauf legt die Routine das Registerpaar bc auf dem Stapel ab.

Das Register hl wird mit dem Wert im Akkumulator geladen, inkrementiert und in Register bc kopiert. Die Routine addiert bc daraufhin e-mal zu hl; das Ergebnis in hl ist die Anzahl der zu kopierenden Bildschirm-Pixel. Der Akkumulator wird mit dem Wert in Register l geladen, und die vier am weitesten links stehenden Bits werden auf null gesetzt. Das Ergebnis wird in Register b kopiert, damit es als Zähler dienen kann.

Das Registerpaar hl wird vom Stapel zurückgeholt, und Register l wird in Register c kopiert. Wenn in Register b eine Null steht, wird es mit sechzehn geladen. Dies ist die Anzahl der Bits in einem Registerpaar. Dann wird die "subroutine" aufgerufen, und der Akkumulator wird mit dem Wert von POINT (l,h) geladen. Die Routine rotiert Registerpaar de nach links und lädt den Wert des Akkumulators in das am weitesten rechts stehende Bit von Register e.

Sofern Register l gleich der linken x-Koordinate ("left x co-ord") ist, springt die Routine zu "next row". Andernfalls wird Register l dekrementiert, gefolgt von Register b. Wenn in Register b keine Null steht, springt die Routine zu "save", um dadurch das nächste Bit in Registerpaar de einzugeben. Wenn in Register b null steht, wird Registerpaar de auf dem Stapel abgelegt, und die Routine springt zu "full" (deutsch: voll).

Bei "next row" wird das Register l mit der rechten x-Koordinate ("right x co-ord") geladen sowie der Akkumulator mit dem Wert in Register h geladen. Wenn der Wert des Akkumulators gleich der unteren y-Koordinate ist, springt die Routine zu "copy", weil der letzte zu kopierende Pixel schon in de eingegeben wurde. Andernfalls wird das Register h dekrementiert, damit es auf die nächste Zeile zeigt. Außerdem macht die Routine eine Schleife zu "next bit".

Bei "copy" wird de auf dem Stapel abgelegt, und die Register b, h und l werden auf null gesetzt, damit sie als Zähler verwendet werden können. Das Registerpaar hl wird in die Adressen 23306/7 gebracht, so daß hl als Zähler für weitere Schleifen benutzt werden kann, ohne daß man den Stapel braucht. Wenn in Register b eine Null steht, wird de vom Stapel zurückgeholt und Register b wird wieder auf sechzehn gesetzt, was die Anzahl der in de gespeicherten Pixel angibt. Register b wird dekrementiert. Dies gibt an, daß ein Informationsbit von de weggeholt werden muß. Die Routine lädt das am weitesten rechts stehende Bit in Register e in den Akkumulator und rotiert Registerpaar de nach rechts. Sie legt de, bc und af auf dem Stapel ab, während einige Rechnungen durchgeführt werden.

Die Register h und l werden mit eins geladen, damit sie als Zähler benutzt werden können. Die Routine bringt hl in die Adressen 23304/5. Sie lädt den Akkumulator mit dem Wert des Bytes in Adresse 23307; dies ist einer der Zähler, die zuvor gespeichert wurden. Das Registerpaar de wird mit dem vertikalen Maßstab geladen. Dies wird dann mit dem Wert im Akkumulator multipliziert, und das Ergebnis wird in hl eingegeben. Dies wird zu der neuen unteren y-Koordinate im Akkumulator geladen. Dann wird das Byte in Adresse 23304 zum Akkumulator addiert und das Ergebnis dekrementiert.

Im Akkumulator steht jetzt die y-Koordinate des nächsten zu zeichnenden Pixels. Sie wird auf dem Stapel gespeichert, während die x-Koordinate durch einen sehr ähnlichen Vorgang berechnet wird. Wenn die x-Koordinate berechnet ist, wird sie in Register l geladen. Die Routine holt die y-Koordinate vom Stapel zurück und lädt sie in Register h. Sie setzt den Akkumulator auf den letzten, im Stapel gespeicherten Wert. Wenn dieser eins ist, sollte der Punkt (x,

y) gezeichnet werden, andernfalls nicht. Die Subroutine wird aufgerufen, und es wird entsprechend weitergearbeitet.

Das Registerpaar hl wird mit den Schleifenzählern, die in den beiden Adressen 23304/5 abgelegt sind, geladen. Die Routine inkrementiert Register l, und wenn darin nicht der Wert (1 + "vertical scale") steht, springt sie zu "preserve" (deutsch: erhalten, aufbewahren). Sie inkrementiert Register h, und wenn darin nicht der Wert (1 + "horizontal scale") steht, springt sie zu "loop" (deutsch: Schleife).

Die Registerpaare af, bc und de werden vom Stapel zurückgeholt, und das Registerpaar hl wird mit der zweiten Menge von Schleifenzählern, die in den Adressen 23306/7 abgelegt sind, geladen. Die Routine inkrementiert Register l und springt zu "reset", wenn das Ergebnis ungleich ("right x co-ord" – "left x co-ord" + 1) ist. Sie setzt Register l auf null, was der ursprüngliche Wert des Schleifenzählers ist. Dann inkrementiert sie Register h und springt zu "reset", wenn das Resultat ungleich ("upper y co-ord" – "lower y co-ord" + 1) ist. Dann springt sie zu BASIC zurück.

Die "subroutine" ist mit derjenigen in der Routine "Ausfüllen eines Gebiets" identisch.

7 Routinen zum Manipulieren von Programmen

Streichen von Programmblöcken

Länge: 42
Anzahl der Variablen: 2
Prüfsumme: 5977

Arbeitsweise

Diese Routine streicht Blöcke eines BASIC-Programms zwischen zwei vom Benutzer angegebenen Zeilen.

Variablen

Name	Länge	Speicherzelle	Kommentar
start line no	2	23296	erste zu streichende Zeile
end line no	2	23298	letzte zu streichende Zeile

Aufruf

RAND USR Adresse

Fehlerüberwachung

Die Routine hält an, ohne etwas vom BASIC-Programm zu streichen, wenn die folgenden Fehler auftreten:
(i) Die letzte Zeilennummer ist kleiner als die erste Zeilennummer.
(ii) Es gibt kein BASIC-Programm zwischen den beiden vorgegebenen Zeilen.
(iii) Eine oder beide vorgegebenen Zeilennummern sind null.

Kommentare

Diese Routine ist beim Streichen eines großen Programmzeilenblocks ziemlich langsam, aber trotzdem geht es viel schneller, als von Hand die Zeilen zu streichen. Geben Sie keine Zeilennummern über 9999 ein!

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23296)	42 Ø 91
	ld de, (23298)	237 91 2 91
	ld a,h	124
	or l	181
	ret z	2ØØ
	ld a,d	122
	or e	179
	ret z	2ØØ
	push de	213
	call 651Ø	2Ø5 11Ø 25
	ex (sp), hl	227
	inc hl	35
	call 651Ø	2Ø5 11Ø 25
	pop de	2Ø9
	and a	167
	sbc hl,de	237 82
	ret z	2ØØ
	ret c	216
	ex de,hl	235
next chr	ld a,d	122
	or e	179
	ret z	2ØØ
	push de	213
	push hl	229
	call 412Ø	2Ø5 24 16
	pop hl	225
	pop de	2Ø9
	dec de	27
	jr next chr	24 243

Wie es funktioniert

Die Registerpaare hl und de werden mit den Anfangs- beziehungsweise Endzeilennummern ("start line no" beziehungsweise "end line no") geladen. Die Routine überprüft die Werte, und wenn einer oder beide null sind, springt sie zu BASIC zurück.

Dann ruft sie die ROM-Routine in Adresse 651Ø auf; diese stellt die Adresse der ersten Zeile bereit. Sie wird nochmals aufgerufen, um die Adresse des Zeichens nach "ENTER" in der letzten Zeile zu finden. Das Registerpaar hl wird auf die Differenz der beiden Adressen gesetzt, und wenn diese null oder negativ ist, springt die Routine zu BASIC zurück.

Die Routine kopiert den Inhalt des Registerpaars hl in de; dies soll als Zähler eingesetzt werden. Wenn der Zähler null ist, so ist die Routine fertig, wenn nicht, wird die ROM-Routine in Adresse 412Ø aufgerufen. Sie streicht ein Zeichen. Die Routine springt dann zurück zu "next chr".

Austauschen von Zeichen

Länge: 46
Anzahl der Variablen: 2
Prüfsumme: 5ØØØ

Arbeitsweise

Diese Routine tauscht ein bestimmtes Zeichen in einem BASIC-Programm, jedesmal wenn es vorkommt, gegen ein anderes aus. Beispielsweise können alle PRINT-Befehle durch LPRINT-Befehle ersetzt werden.

Variablen

Name	Länge	Speicherzelle	Kommentar
chr old	1	23296	zu ersetzendes Zeichen
chr new	1	23297	einzusetzendes Zeichen

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn kein BASIC-Programm im Speicher ist oder wenn eines der vorgegebenen Zeichen einen Code unter 32 hat, springt die Routine zu BASIC zurück.

Kommentare

Diese Routine läuft sehr schnell, aber sie läuft natürlich umso länger, je länger das BASIC-Programm ist.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld, bc, (23296)	237 75 Ø 91
	ld a,31	62 31
	cp b	184
	ret nc	2Ø8
	cp c	185
	ret nc	2Ø8
	ld hl, (23635)	42 83 92
next chr	inc hl	35
	inc hl	35
	inc hl	35

Marke	Assemblersprache	einzugebende Zahlen
check	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	ret nc	2∅8
	add hl,de	25
	inc hl	35
	ld a, (hl)	126
	inc hl	35
	cp 13	254 13
	jr z, next chr	4∅ 237
	cp 14	254 14
	jr nz, compare	32 3
	inc hl	35
	jr next chr	24 23∅
compare	dec hl	43
	cp c	185
	jr nz, check	32 229
	ld (hl), b	112
	jr check	24 226

Wie es funktioniert

Die neuen bzw. alten Zeichen ("chr old" bzw. "chr new") werden in Register b bzw. c geladen. Wenn eines der Zeichen einen Code unter 32 hat, springt die Routine zu BASIC zurück.

Sie lädt die Adresse des BASIC-Programmanfangs in Register hl. Dann erhöht sie das Paar hl und vergleicht es mit der Adresse des Variablenbereichs. Wenn hl nicht kleiner als die Variablenadresse ist, springt die Routine zu BASIC zurück.

Das Paar hl wird inkrementiert, damit es auf das nächste Zeichen zeigt. Der Code dieses Zeichens wird in den Akkumulator geladen, und hl wird nochmals inkrementiert. Wenn der Wert des Akkumulators 13 oder 14 (ENTER oder NUMBER) ist, springt die Routine zu "next chr" zurück und erhöht hl, damit es auf das nächste Zeichen zeigt. Wenn im Akkumulator keine 13 oder 14 steht, vergleicht sie den gespeicherten Wert mit "chr old". Wenn die beiden Zahlen übereinstimmen, wird dieses Zeichen durch "chr new" ersetzt.

Die Routine springt dann zu "check" (deutsch: überprüfen) zurück, um zu überprüfen, ob das Programmende erreicht wurde.

REM-Beseitigung

Länge: 132
Anzahl der Variablen: ∅
Prüfsumme:138∅9

Arbeitsweise

Diese Routine streicht alle REM-Anweisungen eines BASIC-Programms im Speicher.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn kein BASIC-Programm im Speicher ist, springt die Routine zurück, ohne etwas zu tun.

Kommentare

Die verwendete ROM-Routine, die Zeichen streicht, ist nicht sehr schnell. Die Routine läuft deshalb möglicherweise ziemlich lange.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23635)	42 83 92
	jr check	24 31
next line	push hl	229
	inc hl	35
	inc hl	35
	ld c, (hl)	78
	inc hl	35
	ld b, (hl)	7∅
next chr	inc hl	35
	ld a,(hl)	126
	cp 33	254 33
	jr c, next chr	56 25∅
	cp 234	254.234
	jr nz, search	32 26
	inc bc	3
	inc bc	3
	inc bc	3
	inc bc	3
	pop hl	225
delete line	push bc	197
	call 412∅	2∅5 24 16
	pop bc	193
	dec bc	11
	ld a,b	12∅
	or c	177
	jr nz, delete line	32 246

Marke	Assemblersprache	einzugebende Zahlen
check	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	ret nc	2Ø8
	add hl,de	25
	jr next line	24 214
search	inc hl	35
	ld a, (hl)	126
	cp 13	254 13
	jr nz, not enter	32 8
enter found	pop hl	225
	add hl,bc	9
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr check	24 231
not enter	cp 14	254 14
	jr nz, not number	32 7
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr search	24 231
not number	cp 33	254 33
	jr c, search	56 227
	cp 34	254 34
	jr nz, not quote	32 8
find quote	inc hl	35
	ld a, (hl)	126
	cp 34	254 34
	jr nz, find quote	32 25Ø
	jr search	24 215
not quote	cp 58	254 58
	jr nz, search	32 211
	ld d,h	84
	ld e,l	93
find enter	inc hl	35
	ld a, (hl)	126
	cp 13	254 13
	jr z, enter found	4Ø 2Ø9
	cp 33	254 33
	jr c, find enter	56 246
	cp 234	254 234
	jr nz, not quote	32 236
	ld h,d	98
	ld l,e	1Ø7
delete chr	push bc	197
	call 412Ø	2Ø5 24 16

Marke	Assemblersprache	einzugebende Zahlen
	pop bc	193
	dec bc	11
	ld a, (hl)	126
	cp 13	254 13
	jr nz, delete chr	32 245
	pop hl	225
	inc hl	35
	inc hl	35
	ld (hl),c	113
	inc hl	35
	ld (hl),b	112
	dec hl	43
	dec hl	43
	dec hl	43
	jr check	24 160

Wie es funktioniert

Die Adresse des Anfangs des BASIC-Programmbereichs wird in Register hl geladen, und es wird zu derjenigen Routine gesprungen, die überprüft, ob das Ende des Programmbereichs erreicht wurde. Wenn dies der Fall ist, wird zu BASIC zurückgesprungen.

Die Routine springt zu "next line". Zur späteren Verwendung speichert dieser Teil der Routine die Adresse in hl auf dem Stapel. Dann lädt sie bc mit der Länge der angetroffenen BASIC-Zeile. Die Routine "next chr" inkrementiert die Adresse in hl und lädt den Akkumulator mit dem in dieser Adresse gespeicherten Zeichen. Wenn dieses einen Code kleiner als 33 hat, was anzeigt, daß es ein Leer- oder Steuerzeichen ist, springt die Routine zurück und wiederholt diesen Teil noch einmal. Ist das angetroffene Zeichen keine REM-Anweisung, springt sie zu "search" (deutsch: suchen).

Sofern ein REM gefunden wurde, wird das Register bc um vier erhöht, so daß es als Zähler verwendet werden kann. hl wird vom Stapel zurückgeholt. Dann werden mit Hilfe der ROM-Routine in Adresse 4120 bc Zeichen bei Adresse hl gestrichen. Die Routine "fällt durch" bis zur Routine "check".

Wenn die Routine zu "search" springt, inkrementiert sie hl, damit es auf das nächste Zeichen zeigt, und lädt dieses in den Akkumulator. Wenn es ein ENTER-Zeichen ist, holt sie hl vom Stapel zurück und erhöht es, damit es auf den Anfang der nächsten Zeile zeigt. Daraufhin springt die Routine zu "check".

Wenn im Akkumulator das NUMBER-Zeichen (14) steht, wird hl erhöht, so daß es auf das erste Zeichen nach der gespeicherten Zahl zeigt; sodann wiederholt die Routine den Suchprozeß.

Daraufhin werden Zeichen gesucht, deren Codes unter 33 liegen, und wenn die Routine eines findet, springt sie zu "search" zurück. Wenn ein Anführungszeichen (34) (englisch: quote) gefunden wird, springt die Routine, bis sie ein zweites Anführungszeichen findet; erst dann setzt sie die Suche fort. Falls das gefundene Zeichen kein Doppelpunkt ist, der auf eine Zeile mit mehreren Befehlen hinweist, wird die Suche wiederholt. hl wird dann in de kopiert, um die Adresse des Doppelpunkts zu speichern, und dann wird hl inkrementiert, damit es auf das nächste Zeichen zeigt. Wenn dieses Zeichen das ENTER-Zeichen ist,

wird zu "enter found" gesprungen. Ist es ein Steuer- oder Leerzeichen, springt die Routine zurück zu "find enter".

Wenn das Zeichen kein REM-Befehl ist, springt die Routine zu "not quote" zurück. Wenn sie ein REM-Zeichen gefunden hat, lädt sie hl mit der Adresse des letzten angetroffenen Doppelpunkts. Dann werden alle Zeichen von hl bis zum nächsten ENTER-Zeichen gestrichen. Die Routine korrigiert die Zeiger für die Zeile, setzt hl auf den Anfang der Zeile und springt zu "check" zurück.

REM-Erzeugung

Länge: 85
Anzahl der Variablen: 3
Prüfsumme: 9526

Arbeitsweise

Diese Routine erzeugt eine REM-Anweisung in einer spezifizierten Zeile mit einer vorgegebenen Anzahl von Zeichen. Diese werden vom Benutzer gewählt.

Variablen

Name	Länge	Speicherzelle	Kommentar
line number	2	23296	Zeile, bei der REM eingefügt werden soll
number char.	2	23298	Zeichenzahl nach REM
char. code	1	23300	Code der Zeichen nach REM

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn die vorgegebene Zeilenzahl null oder größer als 9999 ist oder wenn eine Zeile mit der gleichen Nummer schon existiert, springt die Routine zu BASIC zurück.

Kommentare

Diese Routine überprüft nicht, ob genügend Speicherplatz für das Einfügen der neuen Zeile vorhanden ist. Bevor man die Routine fährt, sollte man dies also überprüfen. Dies geschieht durch Aufrufen der Routine "Freier Speicherplatz", die an anderer Stelle in diesem Buch steht.

Die Zeichen, die nach REM eingefügt werden, sollten am besten Codes haben, die größer als 31 sind, da die Steuerzeichen (Ø–31) möglicherweise die LIST-Routine im ROM durcheinanderbringen.

Die ROM-Routine, die zum Einschieben von Zeichen aufgerufen wird, ist ziemlich langsam, so daß die REM-Erzeugung unter Umständen lange läuft.

Die REM-Anweisung, die durch diese Routine erzeugt wird, kann verwendet werden, um Maschinencode oder Daten zu speichern, die an einen bestimmten Ort gebracht werden müssen.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23296)	42 Ø 91
	ld, a,h	124
	or l	181
	ret z	2ØØ
	ld de, 1ØØØØ	17 16 39
	and a	167
	sbc hl,de	237 82
	ret nc	2Ø8
	add hl,de	25
	push hl	229
	call 651Ø	2Ø5 11Ø 25
	jr nz, create	32 2
	pop hl	225
	ret	2Ø1
create	ld bc, (23298)	237 75 2 91
	push bc	197
	push bc	197
	ld a, 13	62 13
	call 3976	2Ø5 136 15
	inc hl	35
	pop bc	193
next chr	push bc	197
	ld a,b	12Ø
	or c	177
	jr z, insert REM	4Ø 11
	ld a, (233ØØ)	58 4 91
	call 3976	2Ø5 136 15
	inc hl	35
	pop bc	193
	dec bc	11
	jr next chr	24 24Ø
insert REM	pop bc	193
	ld a, 234	62 234
	call 3976	2Ø5 136 15
	inc hl	35
	pop bc	193
	inc bc	3
	inc bc	3

Marke	Assemblersprache	einzugebende Zahlen
	ld a,b	12∅
	push bc	197
	call 3976	2∅5 136 15
	pop bc	193
	inc hl	35
	ld a,c	121
	call 3976	2∅5 136 15
	inc hl	35
	pop bc	193
	ld a,c	121
	push bc	197
	call 3976	2∅5 136 15
	pop bc	193
	inc hl	35
	ld a,b	12∅
	jp 3976	195 136 15

Wie es funktioniert

Das Registerpaar hl wird mit der vorgegebenen Zeilennummer geladen. Dies wird mit null verglichen, und wenn die Zahlen übereinstimmen, springt die Routine zu BASIC zurück. Dasselbe geschieht, wenn hl eine Zahl enthält, die größer als 9999 (die höchstmögliche Zeilennummer) ist.

Eine ROM-Routine, die in hl die Adresse derjenigen Zeile zurückstellt, deren Nummer vorher in hl war, wird aufgerufen. Wenn das Nullflag gesetzt ist, existiert dort schon eine Zeile. Deshalb springt die Routine zu BASIC zurück.

Wenn das Nullflag nicht gesetzt ist, springt die Routine zu "create" (deutsch: erzeugen). Sie lädt bc mit der Anzahl der Zeichen, die nach "REM" eingefügt werden sollen, und speichert diese Zahl auf dem Stapel. Der Akkumulator wird mit 13 geladen; das ist der Code für das ENTER-Zeichen. Dann wird die ROM-Routine in Adresse 3976 aufgerufen, die das ENTER-Zeichen einschiebt. Das Register bc wird vom Stapel zurückgeholt. Nachdem die Routine bc wieder auf den Stapel zurückgespeichert hat, testet sie bc, um herauszufinden, ob noch mehr Zeichen eingeschoben werden müssen. Wenn nicht, springt die Routine zu "insert REM" (deutsch: REM einschieben). Wenn ein weiteres Zeichen eingeschoben werden muß, wird der Akkumulator mit dem vorgegebenen Code geladen und die ROM-Routine in 3976 wird verwendet, um es einzuschieben. Die Routine dekrementiert den Zähler bc und springt zurück, um abzufragen, ob bc null ist. Sobald die Routine "insert REM" erreicht hat, fügt sie ein REM-Zeichen mittels derselben ROM-Routine ein. bc wird dann mit der Länge der neuen Zeile geladen, und die Zeiger für diese Zeile werden erzeugt. Die Routine holt dann die Zeilennummer vom Stapel zurück und fügt sie ein. Danach springt die Routine zu BASIC zurück.

Kompaktprogramm

Länge: 71
Anzahl der Variablen: ∅
Prüfsumme: 7158

Arbeitsweise

Diese Routine streicht alle überflüssigen Steuer- und Leerzeichen in einem BASIC-Programm. Dadurch wird der verfügbare RAM-Bereich vergrößert.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn kein BASIC-Programm im Speicher ist, springt die Routine sofort zu BASIC zurück.

Kommentare

Bei dieser Routine geht man davon aus, daß schon alle REM-Anweisungen aus dem BASIC-Programm entfernt wurden. Allerdings bricht der Computer nicht zusammen, wenn dies nicht so ist, und man die Routine trotzdem laufen läßt. Die Laufzeit der Routine ist proportional zur Länge des BASIC-Programms im Speicher.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zeichen
	ld hl, (23635)	42 83 92
next line	inc hl	35
	inc hl	35
check	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	ret nc	2Ø8
	add hl,de	25
length	push hl	229
	ld c, (hl)	78
	inc hl	35
	ld b, (hl)	7Ø
next byte	inc hl	35
load	ld a, (hl)	126
	cp 13	254 13
	jr nz, number	32 8
restore	pop hl	225
	ld (hl),c	113
	inc hl	35
	ld (hl),b	112
	add hl,bc	9
	inc hl	35
	jr next line	24 227

Marke	Assemblersprache	einzugebende Zahlen
number	cp 14	254 14
	jr nz, quote	32 7
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr next byte	24 231
quote	cp 34	254 34
	jr nz, control	32 12
find quote	inc hl	35
	ld a, (hl)	126
	cp 34	254 34
	jr z, next byte	4Ø 221
	cp 13	254 13
	jr z, restore	4Ø 223
	jr find quote	24 244
control	cp 33	254 33
	jr nc, next byte	48 211
	push bc	197
	call 412Ø	2Ø5 24 16
	pop bc	193
	dec bc	11
	jr load	24 2Ø4

Wie es funktioniert

Die Routine lädt die Adresse des BASIC-Programms in Registerpaar hl. Dann inkrementiert sie hl zweimal, so daß es auf die zwei Bytes zeigt, welche die Länge der nächsten Zeile enthalten. Das Registerpaar de wird mit der Adresse des Variablenbereichs geladen. Wenn hl nicht kleiner als de ist, springt die Routine zu BASIC zurück, weil das Programmende erreicht wurde.

Die Routine speichert die Adresse in hl auf dem Stapel, lädt bc mit der Länge der aktuellen Zeile und inkrementiert hl, damit es auf das nächste Byte in der Zeile zeigt. Das Byte in hl wird dann in den Akkumulator geladen. Wenn der Akkumulator nicht 13 enthält, springt die Routine zu "number".

Bis die Routine "restore" erreicht, muß sie das Ende der aktuellen Zeile gefunden haben. Die Adresse des Zeichenzählers wird vom Stapel in hl geladen; die aktuelle Länge wird eingefügt. Die Routine addiert die Zeilenlänge zu hl, inkrementiert hl und springt zu "next line" zurück.

Wenn die Routine "number" erreicht, überprüft sie, ob im Akkumulator das NUMBER-Zeichen (14) steht. Wenn ja, wird hl um fünf erhöht, so daß die folgende Zahl unverändert bleibt. Dann springt die Routine zu "next byte".

Wenn im Akkumulator nicht der Code für ein Anführungszeichen steht, springt die Routine zu "control". Wenn sie ein Anführungszeichen gefunden hat, macht sie eine Schleife bis das Zeilenende oder ein anderes Anführungszeichen gefunden wird. Im ersten Fall springt sie zu "restore", im zweiten Fall zu "next byte".

Bei "control" wird überprüft, ob das Zeichen einen Code kleiner als 33 hat. Wenn nicht, springt die Routine zu "next byte".

Sobald die Routine ein Leerzeichen oder Steuerzeichen gefunden hat, wird die ROM-Routine in Adresse 4120 aufgerufen, um es zu streichen. Der Zeichenzähler, der in bc abgelegt ist, wird dekrementiert. Dann springt die Routine zu "load".

Laden von Maschinencode in DATA-Anweisungen

Länge: 179
Anzahl der Variablen: 2
Prüfsumme: 19181

Arbeitsweise

Diese Routine erzeugt eine DATA-Anweisung in Zeile 1 eines BASIC-Programms und füllt sie dann mit Daten, die mit PEEK aus dem Speicher geholt wurden.

Variablen

Name	Länge	Speicherzelle	Kommentar
data start	2	23296	Adresse, aus der kopiert wird
data length	2	23298	Anzahl der zu kopierenden Bytes

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn die Anzahl der zu kopierenden Bytes null ist oder wenn es schon eine Zeile 1 gibt, springt die Routine sofort zu BASIC zurück. Die Routine überprüft nicht, ob genügend Speicherplatz für die Zeile zur Verfügung steht; dies muß von Hand geschehen.

Die Routine braucht zehn Bytes pro Datenbyte und fünf weitere Bytes für Zeilennummern, Zeiger usw. Man sollte jedoch immer berücksichtigen, daß die verwendete ROM-Routine einen großen Arbeitsspeicher benutzt. Wenn nicht genügend Speicherplatz zur Verfügung steht, werden die Zeilenzeiger nicht richtig gesetzt und das BASIC-Listing verfälscht.

Kommentare

Die Laufzeit dieser Routine ist proportional zur Länge des zu kopierenden Speicherteils.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld de, (23296)	237 91 Ø 91
	ld bc, (23298)	237 75 2 91
	ld a,b	12Ø
	or c	177
	ret z	2ØØ
	ld hl, (23635)	42 83 92
	ld a, (hl)	126
	cp Ø	254 Ø
	jr nz, continue	32 6
	inc hl	35
	ld a, (hl)	126
	cp l	254 1
	ret z	2ØØ
	dec hl	43
continue	push hl	229
	push bc	197
	push de	213
	sub a	151
	call 3976	2Ø5 136 15
	ex de,hl	235
	ld a,l	62 1
	call 3976	2Ø5 136 15
	ex de,hl	235
	call 3976	2Ø5 136 15
	ex de,hl	235
	call 3976	2Ø5 136 15
	ex de,hl	235
	ld a, 228	62 228
	call 3976	2Ø5 136 15
	ex de,hl	235
next byte	pop de	2Ø9
	ld a, (de)	26
	push de	213
	ld c,47	14 47
hundreds	inc c	12
	ld b, 1ØØ	6 1ØØ
	sub b	144
	jr nc, hundreds	48 25Ø
	add, a,b	128
	ld b,a	71
	ld a,c	121
	push bc	197
	call 3976	2Ø5 136 15
	ex de,hl	235
	pop bc	193
	ld a,b	12Ø
	ld c,47	14 47
tens	inc c	12

Marke	Assemblersprache	einzugebende Zahlen
	ld b,1∅	6 1∅
	sub b	144
	jr nc, tens	48 25∅
	add a,b	128
	ld b,a	71
	ld a,c	121
	push bc	197
	call 3976	2∅5 136 15
	pop bc	193
	ex de,hl	235
	ld a,b	12∅
	add a,48	198 48
	call 3976	2∅5 136 15
	ex de,hl	235
	ld a,14	62 14
	ld b, 6	6 6
next zero	push bc	197
	call 3976	2∅5 136 15
	pop bc	193
	ex de,hl	235
	sub a	151
	djnz next zero	16 247
	pop de	2∅9
	push hl	229
	dec hl	43
	dec hl	43
	dec hl	43
	ld a, (de)	26
	ld (hl),a	119
	pop hl	225
	inc de	19
	pop bc	193
	dec bc	11
	ld a,b	12∅
	or c	177
	jr z, enter	4∅ 1∅
	push bc	197
	push de	213
	ld a, 44	62 44
	call 3976	2∅5 136 15
	ex de, hl	235
	jr next byte	24 173
enter	ld a, 13	62 13
	call 3976	2∅5 136 15
	pop hl	225
	ld bc, ∅	1 ∅ ∅
	inc hl	35
	inc hl	35
	ld d,h	84

Marke	Assemblersprache	einzugebende Zahlen
	ld e,l	93
	inc hl	35
pointers	inc hl	35
	inc bc	3
	ld a, (hl)	126
	cp 14	254 14
	jr nz, end?	32 12
	inc bc	3
	inc bc	3
	inc bc	3
	inc bc	3
	inc bc	3
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr pointers	24 237
end?	cp 13	254 13
	jr nz pointers	32 233
	ld a,c	121
	ld (de),a	18
	inc de	19
	ld a,b	12Ø
	ld (de),a	18
	ret	2Ø1

Wie es funktioniert

Die Startadresse der zu kopierenden Bytes wird in Registerpaar de geladen. Die Anzahl der zu kopierenden Bytes wird in Registerpaar bc geladen. Wenn in bc eine Null steht, springt die Routine sofort zu BASIC zurück.

Die Adresse des BASIC-Programms wird in Registerpaar hl geladen. Das Byte aus der Adresse, die in hl gespeichert ist, wird in den Akkumulator geladen. Dies ist das höherwertige Byte der Zeilennummer. Wenn darin keine Null steht, gibt es noch keine Zeile 1, und deshalb springt die Routine zu "continue" (deutsch: fortfahren). Falls im höherwertigen Byte null steht, lädt sie den Akkumulator mit dem niederwertigen Byte. Wenn dieses auf eins steht, gibt es die Zeile schon; also springt die Routine zu BASIC zurück. Die Routine speichert die Adresse des höherwertigen Bytes der Zeilennummer auf dem Stapel sowie die Anzahl der zu kopierenden Bytes, gefolgt von der Adresse der Daten.

Der Akkumulator wird dann mit null geladen, dem höherwertigen Byte der neuen Zeilennummer. Die ROM-Routine in Adresse 3976 wird aufgerufen und schiebt dann das im Akkumulator abgelegte Zeichen an der in hl gespeicherten Adresse ein. Die Routine setzt hl auf den Wert, den es vor dieser Operation hatte. Sie lädt den Akkumulator mit eins und schiebt dies dreimal ein. Die erste Eins ist für das niederwertige Byte der Zeilennummer, die nächsten beiden Einsen entsprechen der Zeilenlänge. Der Akkumulator wird dann mit dem Code des DATA-Zeichens geladen; dies wird eingeschoben.

Die Adresse des nächsten Datenbytes wird vom Stapel zurückgeholt und in de geladen. Der Akkumulator wird mit diesem Byte geladen. Die Routine legt dann de wieder auf dem Stapel ab. Sie lädt Register c mit dem Code für das Zeichen "Ø" minus eins. Register c wird inkrementiert, und Register b mit 1ØØ geladen. Register b wird vom Akkumulator subtrahiert, und wenn das Ergebnis nicht negativ ist, macht die Routine eine Schleife zurück zu "hundreds" (deutsch: Hunderter).

Register b wird einmal zum Akkumulator addiert, so daß im Akkumulator eine positive Zahl steht. Die Routine lädt dann diesen Wert in Register b. Sie lädt den Inhalt von c in den Akkumulator und speichert dann bc auf dem Stapel ab. Die ROM-Routine in Adresse 3976 schiebt dann das Zeichen, das im Akkumulator abgelegt ist, an der Adresse, die in hl gespeichert ist, ein. Das Registerpaar bc wird vom Stapel zurückgeholt; außerdem wird der Akkumulator mit dem Wert aus Register b geladen. Der oben beschriebene Prozeß wird mit b = 1Ø wiederholt. Der Akkumulator wird dann um 48 erhöht, und das entstehende Zeichen wird eingeschoben.

Die obige Routine hat den Dezimalwert des vorgefundenen Datenbytes in den DATA-Befehl eingeschoben. Nun muß die binäre Darstellung eingeschoben werden. Dies wird durch das NUMBER-Zeichen, chr 14, markiert, welches zuerst, gefolgt von fünf Nullen, eingegeben wird. Der Wert des zu kopierenden Bytes wird eingepoket und ersetzt die dritte Null. Die Routine inkrementiert dann de, damit es auf das nächste Datenbyte zeigt. Die Anzahl der zu kopierenden Bytes wird aus dem Stapel in bc geladen, und dieses wird dekrementiert. Wenn das Ergebnis null ist, springt die Routine zu "enter". Andernfalls legt sie die Registerpaare bc und de auf den Stapel zurück, fügt ein Komma im DATA-Befehl ein und springt zu"next byte".

Bei "enter" wird ein ENTER-Zeichen eingeschoben, um das Ende des DATA-Befehls zu markieren. Die Adresse des Zeilenanfangs wird in hl geladen, und bc wird null gesetzt. hl wird erhöht, damit es auf das niederwertige Byte des Zeichenzählers zeigt. Diese neue Adresse wird in de kopiert. Die Routine inkrementiert hl, damit es auf das höherwertige Byte des Zeichenzählers zeigt. Dann inkrementiert sie hl und bc und lädt den Akkumulator mit dem Zeichen aus der in hl gespeicherten Adresse.

Wenn im Akkumulator 14 steht, wurde eine Zahl gefunden. Also werden sowohl hl als auch bc um fünf erhöht, damit sie auf das erste Zeichen nach der Zahl zeigen. Dann springt die Routine zu "pointers" (deutsch: Zeiger).

Wenn im Akkumulator weder 14 noch 13 steht, springt die Routine zu "pointers" zurück.

Wenn die Routine bis hierher gekommen ist, muß sie das ENTER-Zeichen, welches das Zeilenende markiert, vorgefunden haben. In bc steht nun die Zeilenlänge, und diese wird jetzt in den Zeichenzähler eingepoket, dessen Adresse in de abgelegt ist.

Die Routine springt dann zu BASIC zurück.

Umwandlung von Buchstaben (klein/groß)

Länge: 41
Anzahl der Variablen: Ø
Prüfsumme: 4683

Arbeitsweise

Diese Routine verwandelt alle Kleinbuchstaben in einem BASIC-Programm in Großbuchstaben oder umgekehrt.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn es kein BASIC-Programm im Speicher gibt, springt die Routine sofort zu BASIC zurück.

Kommentare

Wenn man diese Routine abändern will, so daß sie Großbuchstaben in Kleinbuchstaben umwandelt, muß man die markierten Zahlen folgendermaßen ersetzen:

96* durch 64

9Ø** durch 122

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23635)	42 83 92
	ld de, (23627)	237 91 75 92
jump	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
changed	inc hl	35
next byte	and a	167
	sbc hl,de	237 82
	ret nc	2Ø8
	add hl,de	25
	ld a, (hl)	126
	cp 13	254 13

Marke	Assemblersprache	einzugebende Zahlen
	jr z, jump	4Ø 241
	cp 14	254 14
	inc hl	35
	jr z, jump	4Ø 236
	sub 96	214 96*
	jr c, next byte	56 237
	sub 26	214 26
	jr nc, next byte	48 233
	add a,9Ø	198 9Ø**
	dec hl	43
	ld (hl), a	119
	jr changed	24 226

Wie es funktioniert

Die Adresse des BASIC-Programms wird in Registerpaar hl, die Adresse des Variablenbereichs in de geladen, hl wird erhöht, damit die Zeilennummer und der Zeichenzähler übersprungen werden. Wenn hl nicht kleiner als de ist, springt die Routine zu BASIC zurück, weil das Ende des Programms erreicht wurde.

Der Akkumulator wird mit dem in hl gespeicherten Byte geladen. Wenn dieses Byte ein ENTER-Zeichen ist, springt die Routine zurück zu "jump". Wenn das Byte das NUMBER-Zeichen ist, springt die Routine ebenfalls zu "jump" zurück, wobei sie hl schon inkrementiert hat. So werden die fünf Bytes nach dem Zeichen 14 vermieden.

Die Routine zieht vom Akkumulator 96 ab. Wenn das Ergebnis negativ ist, springt sie zu "next byte", weil der Buchstabe kein Kleinbuchstabe sein kann. Dann wird vom Akkumulator 26 abgezogen. Wenn das Ergebnis nicht negativ ist, wird zu "next byte" gesprungen, da der Buchstabe einen für einen Kleinbuchstaben zu hohen Code hat. Dann addiert die Routine 9Ø zum Akkumulator. Das ergibt den Code des entsprechenden Großbuchstabens. Die Routine dekrementiert hl, damit es auf den Buchstaben zeigt, der ersetzt werden soll. Sie poket diese Adresse mit dem Wert im Akkumulator. Dann springt sie zu "changed" (deutsch: geändert).

8 Routinen aus dem Werkzeugkasten

Umnumerieren

Länge: 382
Anzahl der Variablen: 2
Prüfsumme: 41629

Arbeitsweise

Diese Routine numeriert ein BASIC-Programm um, einschließlich aller GOTO, GOSUBs usw.

Variablen

Name	Länge	Speicherzelle	Kommentar
first line no	2	23296	Die Nummer der ersten Zeile, wenn mit RUN gefahren wird
step	2	23298	Die Differenz zwischen aufeinanderfolgenden Zeilennummern

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn die Nummer der ersten Zeile ("first line no") oder die Schrittweite von einer Zeilennummer zur nächsten ("step") null ist, springt die Routine sofort zu BASIC zurück. Wenn im RAM überhaupt kein BASIC-Programm vorhanden ist, springt sie ebenfalls zu BASIC zurück. Zeilennummern, die aus arithmetischen Ausdrücken bestehen (z. B. GOTO 7*A), die einen Dezimalpunkt enthalten (z. B. GOTO 7.8), die kleiner als null (z. B. GOTO –1) oder größer als 9999 sind (z. B. GOTO 2Ø17Ø), werden nicht beachtet. Wenn die Schrittweite zu groß ist, werden möglicherweise Zeilennummern wiederholt, wodurch das Programm falsch wird. Die Routine erhöht die Länge des BASIC-Programms im RAM. Deshalb sollte immer geprüft werden, ob genügend Platz im RAM zur Verfügung steht.

Kommentare

Die Laufzeit dieser Routine ist proportional zur Länge des BASIC-Programms im RAM.

Die Routine ist nicht verschiebbar und sollte normalerweise in Speicherzelle 32218 begonnen werden. Mit folgendem Verfahren kann ihre Position geändert werden:

(i) Man setze X = neue Adresse – 32218
(ii) Man setze H = INT (X/256)
Man setze L = X-256*H
(iii) Für jedes Zahlenpaar im Listing, das mit "*" gekennzeichnet ist,
setze man LI = L + die erste Zahl
setze man HI = H + die zweite Zahl
Wenn LI größer als 255 ist, setze man HI = HI + 1
und LI = LI – 256
Man ersetze das Zahlenpaar durch LI und HI.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23296)	42 ∅ 91
	ld a, h	124
	or l	181
	ret z	2∅∅
	ld hl, (23298)	42 2 91
	ld a,h	124
	or l	181
	ret z	2∅∅
	ld hl, (23635)	42 83 92
	ld de, (23296)	237 91 ∅ 91
next line	call check	2∅5 76* 127*
	jr nc, find GOTO	48 22
	ld b, (hl)	7∅
	ld (hl),d	114
	inc hl	35
	ld c, (hl)	78
	ld (hl),e	115
	inc hl	35
	ld (hl),c	113
	inc hl	35
	ld (hl),b	112
	inc hl	35
	push hl	229
	ld hl, (23298)	42 2 91
	add hl,de	25
	ex de, hl	235
	pop hl	225
	call end of line	2∅5 65* 127*
	jr next line	24 229

Marke	Assemblersprache	einzugebende Zahlen
find GOTO	ld hl, (23635)	42 83 92
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
search	call find	2Ø5 235* 126*
	jp nc, restore	21Ø 184* 126*
	ld d,h	84
	ld e, l	93
	ld b, Ø	6 Ø
next digit	inc b	4
	inc hl	35
	ld a, (hl)	126
	cp 46	254 46
	jr nz, continue	32 3
find next	ex de,hl	235
	jr search	24 236
continue	cp 14	254 14
	jr nz, next digit	32 242
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	ld a, (hl)	126
	cp 58	254 58
	jr z, found	4Ø 4
	cp 13	254 13
	jr nz, find next	32 234
found	ld a, b	12Ø
compare	cp 4	254 4
	jr z, calculate	4Ø 16
	jr nc, find next	48 227
	push de	213
	ld h,d	98
	ld l,e	1Ø7
	push af	245
	ld a, 48	62 48
	call 3976	2Ø5 136 15
	pop af	241
	inc a	6Ø
	pop de	2Ø9
	jr compare	24 236
calculate	ld b,d	66
	ld c,e	75
	push de	213
	ld hl, Ø	33 Ø Ø
	ld de, 1ØØØ	17 232 3
	call add	2Ø5 226* 126*

Marke	Assemblersprache	einzugebende Zahlen
	ld de, 1∅∅	17 1∅∅ ∅
	call add	2∅5 226* 126*
	ld e, 1∅	3∅ 1∅
	call add	2∅5 226* 126*
	ld a, (bc)	1∅
	sub 48	214 48
	ld e, a	95
	add hl,de	25
	ld b,h	68
	ld c,l	77
	ld hl, (23635)	42 83 92
find line	inc hl	35
	inc hl	35
end of prog	call check	2∅5 76* 127*
	jr c, exists	56 3
	pop hl	225
	jr search	24 153
exists	ld a, (hl)	126
	cp c	185
	jr nc, next byte	48 7
	inc hl	35
wrong line	inc hl	35
	call end of line	2∅5 65* 127*
	jr find line	24 235
next byte	inc hl	35
	ld a, (hl)	126
	cp b	184
	jr c, wrong line	56 245
	dec hl	43
	dec hl	43
	ld c, (hl)	78
	dec hl	43
	ld h, (hl)	1∅2
	ld l, c	1∅5
	pop bc	193
	push bc	197
	push hl	229
	ld de, 1∅∅∅	17 232 3
	call insert	2∅5 212* 126*
	ld de, 1∅∅	17 1∅∅ ∅
	call insert	2∅5 212* 126*
	ld e, 1∅	3∅ 1∅
	call insert	2∅5 212* 126*
	ld e, 1	3∅ 1
	call insert	2∅5 212* 126*
	inc bc	3
	sub a	151
	ld (bc),a	2
	inc bc	3
	ld (bc),a	2

Marke	Assemblersprache	einzugebende Zahlen
	inc bc	3
	pop hl	225
	ld a,l	125
	ld (bc),a	2
	inc bc	3
	ld a,h	124
	ld (bc),a	2
	inc bc	3
	sub a	151
	ld (bc),a	2
	pop hl	225
	jp search	195 15* 126*
restore	ld hl, (23635)	42 83 92
following line	inc hl	35
	inc hl	35
	call check	2Ø5 76* 127*
	ret nc	2Ø8
	ld d,h	84
	ld e,l	93
	inc hl	35
	inc hl	35
	call end of line	2Ø5 65* 127*
	push hl	229
	scf	55
	sbc hl,de	237 82
	dec hl	43
	ex de,hl	235
	ld (hl),e	115
	inc hl	35
	ld (hl),d	114
	pop hl	225
	jr following line	24 231
insert	ld a, 48	62 48
subtract	and a	167
	sbc hl,de	237 82
	jr c, poke	56 3
	inc a	6Ø
	jr subtract	24 248
poke	add hl,de	25
	ld (bc),a	2
	inc bc	3
	ret	2Ø1
add	ld a, (bc)	1Ø
	inc bc	3
	sub 47	214 47
repeat	dec a	61
	ret z	2ØØ
	add hl,de	25
	jr repeat	24 251
find	ld a, (hl)	126

Marke	Assemblersprache	einzugebende Zahlen
	call check	2Ø5 76* 127*
	ret nc	2Ø8
	cp 234	254 234
	jr nz not REM	32 13
find ENTER	inc hl	35
	ld a, (hl)	126
	cp 13	254 13
	jr nz, find ENTER	32 25Ø
increase	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr find	24 234
not REM	cp 34	254 34
	jr nz, not string	32 9
next character	inc hl	35
	ld a, (hl)	126
	cp 34	254 34
	jr nz, next character	32 25Ø
	inc hl	35
	jr find	24 221
not string	cp 13	254 13
	jr z, increase	4Ø 232
	call 6326	2Ø5 182 24
	jr z, find	4Ø 212
	cp 237	254 237
	jr z, check digit	4Ø 27
	cp 236	254 236
	jr z, check digit	4Ø 23
	cp 247	254 247
	jr z, check digit	4Ø 19
	cp 24Ø	254 24Ø
	jr z, check digit	4Ø 15
	cp 229	254 229
	jr z, check digit	4Ø 11
	cp 225	254 225
	jr z, check digit	4Ø 7
	cp 2Ø2	254 2Ø2
	jr z, check digit	4Ø 3
	inc hl	35
	jr find	24 181
check digit	inc hl	35
	ld a, (hl)	126
	cp 48	254 48
	jr c, find	56 175
	cp 58	254 58
	jr nc, find	48 171
	ret	2Ø1

Marke	Assemblersprache	einzugebende Zahlen
end of line	ld a, (hl)	126
again	call 6326	2Ø5 182 24
	jr z, again	4Ø 251
	cp 13	254 13
	inc hl	35
	jr nz, end of line	32 245
check	push hl	229
	push de	213
	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	pop de	2Ø9
	pop hl	225
	ret	2Ø1

Wie es funktioniert

Das Registerpaar hl wird mit der ersten Zeilennummer geladen. Wenn diese null ist, springt die Routine zu BASIC zurück. Die Schrittweite von einer Zeilennummer zur nächsten wird dann in hl geladen, und wenn sie null ist, springt die Routine zu BASIC zurück.

Die Routine lädt hl mit der Adresse des BASIC-Programms und setzt de auf die erste Zeilennummer. Die Subroutine "check" wird aufgerufen, und wenn das Ende des BASIC-Programms erreicht wurde, wird zu "find GOTO" gesprungen. Die Routine lädt die alte Nummer der vorgefundenen Zeile in bc und ersetzt die Nummer durch de; dann kopiert sie bc in den Zeichenzähler.

Die Routine speichert hl auf dem Stapel, lädt es mit der Schrittweite und erhöht es um de. Das Resultat, die nächste Zeilennummer, wird in de kopiert. hl wird dann vom Stapel zurückgeholt und die Subroutine "end of line" erhöht es, so daß es auf die nächste Zeile zeigt. Die Routine springt dann zurück zu "next line".

Bei "find GOTO" wird hl mit der Adresse des BASIC-Programms geladen, und diese wird erhöht, damit sie auf das erste Zeichen der ersten Zeile zeigt. Dann wird die Subroutine "find" aufgerufen. Wenn es keine GOTOs, GOSUBs usw. mehr gibt, die noch geändert werden müssen, springt die Routine zu "restore". Andernfalls steht in hl nach Rücksprung von der Subroutine die Adresse der ersten Ziffer nach GOTO, GOSUB usw. Diese wird in de kopiert, und Register b wird auf null gesetzt. Das Register b wird verwendet, um die Ziffern in der folgenden Zahl abzuzählen.

Die Routine inkrementiert Register b und erhöht hl, damit es auf das nächste Zeichen zeigt. Dann lädt sie dieses Zeichen in den Akkumulator. Wenn das Zeichen ein Dezimalpunkt ist, lädt die Routine hl mit de und springt zurück zu "search", um das nächste GOTO zu finden. Wenn das Zeichen kein NUMBER-Zeichen ist, springt die Routine zu "next digit" (deutsch: nächste Ziffer).

hl wird erhöht, damit es auf das Zeichen unmittelbar nach der Gleitkomma-Zahl zeigt. Wenn das kein Doppelpunkt oder ENTER-Zeichen ist, springt die Routine zu "find next", da das zur Diskussion stehende GOTO eine berechnete Zielmarke hat. Der Akkumulator wird mit dem Wert in Register b geladen. Wenn dieser vier ist, springt die Routine zu "calculate" (deutsch:

berechnen); wenn er größer als vier ist, wird zu "find next" zurückgesprungen, da Zeilennummern größer als 9999 ungültig sind.

Dann wird de auf dem Stapel abgelegt und in hl kopiert. Der Akkumulator wird dann auf dem Stapel gespeichert und mit dem Code des Zeichens Null geladen. Dies wird bei der Adresse, die in hl steht, eingeschoben, und zwar mittels der ROM-Routine in Adresse 3976. Der Akkumulator wird vom Stapel zurückgeholt und inkrementiert. Die neue Anzahl von Ziffern in der Zeilennummer steht dann darin. Darauf holt die Routine de vom Stapel zurück und springt zu "compare" (deutsch: vergleichen).

Bei "calculate" wird die Adresse, die in de steht, in bc kopiert und dann auf dem Stapel abgelegt. hl wird mit null, de mit 1000 geladen. Dann wird die Subroutine "add" aufgerufen, um die Tausender in der betreffenden Zeilennummer zu hl zu addieren. Dies wird dann für die Hunderter, Zehner und Einer wiederholt, wobei zum Schluß die Zeilennummer in hl steht. Das Registerpaar bc wird mit dem Ergebnis geladen.

Die Adresse des BASIC-Programms wird in Register hl geladen. Die Subroutine "check" wird aufgerufen. Wenn die Routine das Ende des Programmbereichs erreicht hat, holt sie hl vom Stapel zurück und springt zu "search", weil die Zielmarke des GOTO nicht existiert. Wenn das von hl adressierte Byte kleiner als der Wert des Registers c ist, wird hl erhöht, damit es auf die nächste Zeile zeigt, und die Routine springt zu "find line" (deutsch: Zeile finden). Andernfalls wird hl inkrementiert, damit es auf das nächste Byte derjenigen Zeilennummer zeigt, die gerade getestet wird. Wenn dies kleiner als der Wert in Register b ist, wird zu "wrong line" (deutsch: falsche Zeile) gesprungen.

Wenn die Routine bis hierher gekommen ist, muß sie die Zielmarke des GOTO gefunden haben. hl wird erniedrigt, damit es auf den Zeilenanfang zeigt, und dann mit der neuen Zeilennummer geladen. bc wird mit der Adresse auf dem Stapel geladen. Dann wird hl auf dem Stapel gespeichert. In bc steht nun die Adresse, auf welche die Zeilennummer kopiert werden soll. de wird mit 1000 geladen. Die Subroutine "insert" wird aufgerufen. Sie berechnet die Tausender in hl, addiert 48, womit sie eine lesbare Ziffer erzeugt, und poket den Wert in bc. Dann wird bc erhöht, damit es auf das nächste Zeichen zeigt. Dieser Vorgang wird für die Hunderter, Zehner und Einer wiederholt.

Dann baut die Routine die Binärdarstellung der Zeilennummer auf; sie erhöht bc, damit es auf das Zeichen nach dem NUMBER-Zeichen zeigt. Darauf poket sie die nächsten beiden Bytes mit null. hl wird dann vom Stapel geholt und in die nächsten Bytes gepoket. Das fünfte Byte der Nummer wird mit null gepoket. Die Routine holt hl vom Stapel und springt zu "search", um den Vorgang mit dem nächsten GOTO zu wiederholen.

Bei "restore" wird hl mit der Adresse des BASIC-Programmbereichs geladen und dann zweimal inkrementiert, damit es auf die Zeilenlänge zeigt, wo aber die alte Zeilennummer gespeichert wurde. Die Subroutine "check" wird aufgerufen. Wenn das Ende des BASIC-Programms erreicht wurde, springt die Routine zu BASIC zurück.

Die Adresse in hl wird in de geladen und zweimal inkrementiert. Dann wird die Subroutine "end of line" aufgerufen. In hl befindet sich danach die Adresse des ENTER-Zeichens plus eins, d. h. die Adresse der nächsten Zeile. hl wird auf dem Stapel abgelegt. Die Routine subtrahiert de und das vorher gesetzte Übertragsflag von hl, dann dekrementiert sie hl. Dadurch erhält man die neue Zeilenlänge. hl und de werden ausgetauscht und die neue Zeilenlänge in hl und hl+1 geladen. Die Routine holt hl vom Stapel zurück und springt zu "following line".

Insert: Der Akkumulator wird mit dem Code des Zeichens Null geladen. de wird von hl subtrahiert, und wenn das Ergebnis negativ ist, springt die Routine zu "poke". Andernfalls inkrementiert sie den Akkumulator und macht eine Schleife zu "subtract".

Bei "poke" wird de zu hl addiert, und dies ergibt einen positiven Wert. Die Routine poket bc mit dem Wert im Akkumulator und inkrementiert bc, damit es auf das nächste Byte zeigt. Dann erfolgt der Rücksprung.

Add: Der Akkumulator wird mit dem von bc adressierten Byte geladen, und bc wird inkrementiert, damit es auf das nächste Byte zeigt. Die Routine zieht 47 vom Akkumulator ab. Sie dekrementiert den Akkumulator, und wenn das Ergebnis null ist, springt sie zurück. Andernfalls addiert die Routine de zu hl und springt zu "repeat" (deutsch: wiederholen).

Find: Der Akkumulator wird mit dem von hl adressierten Byte geladen. Dann wird die Subroutine "check" aufgerufen, und wenn das Ende des BASIC-Programmes erreicht wurde, erfolgt der Rücksprung. Wenn das Zeichen im Akkumulator nicht das REM-Zeichen ist, wird zu "not REM" gesprungen. Die Routine inkrementiert hl mehrmals bis sie das Ende der Zeile gefunden hat. Sie erhöht hl um 5, damit es auf das erste Zeichen der nächsten Zeile zeigt, und springt dann zu "find".

Wenn bei "not REM" im Akkumulator nicht der Code des Anführungszeichens steht, springt die Routine zu "not string". Andernfalls wird hl mehrmals inkrementiert, bis ein zweites Anführungszeichen gefunden wird. hl wird nochmals inkrementiert, damit es auf das nächste Zeichen zeigt, und die Routine springt zu "find" zurück.

Bei "not string" wird eine Schleife zu "increase" (deutsch: erhöhen) gemacht, sofern im Akkumulator das ENTER-Zeichen steht. Wenn dagegen das NUMBER-Zeichen darin steht, springt die Routine zu "find". Wenn kein GOSUB-, GOTO-, RUN-, LIST-, RESTORE-, LLIST-, LINE-Befehl gefunden wurde, inkrementiert sie hl und springt zu "find". Die Routine inkrementiert hl und lädt den Akkumulator mit dem nächsten Zeichen. Wenn dieses nicht zwischen 48 und 57 liegt, springt die Routine zu "find". Dann erfolgt der Rücksprung.

End of Line: Der Akkumulator wird mit dem von hl adressierten Byte geladen. Wenn dieses das NUMBER-Zeichen ist, wird hl erhöht und zu "again" gesprungen. hl wird inkrementiert. Wenn im Akkumulator nicht das ENTER-Zeichen steht, springt die Routine zu "end of line". Sie testet, ob das Ende des BASIC-Programms erreicht wurde, und springt dann zurück.

Freier Speicherplatz

Länge: 14
Anzahl der Variablen: 0
Prüfsumme: 1443

Arbeitsweise

Diese Routine gibt in Form von Bytes an, wieviel RAM frei zur Verfügung steht.

Aufruf

PRINT USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine sollte aufgerufen werden, bevor man eine Routine verwendet, die möglicherweise die Programmlänge vergrößert. Damit kann man sicherstellen, daß genügend Platz im RAM frei zur Verfügung steht.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, ∅	33 ∅ ∅
	add hl, sp	57
	ld de, (23653)	237 91 1∅1 92
	and a	167
	sbc hl,de	237 82
	ld b,h	68
	ld c,l	77
	ret	2∅1

Wie es funktioniert

Das Registerpaar hl wird auf null gesetzt und die Adresse des Endes des freien RAM-Bereichs wird dazuaddiert (diese Adresse ist in sp gespeichert). Das Registerpaar de wird mit der Adresse des Anfangs des freien RAM-Bereichs geladen und von hl subtrahiert. Die Routine kopiert hl in bc und springt dann zu BASIC zurück.

Programmlänge

Länge: 13
Anzahl der Variablen: ∅
Prüfsumme: 1544

Arbeitsweise

Diese Routine gibt in Form von Bytes an, wie lange ein BASIC-Programm ist.

Aufruf

PRINT USR Adresse

Fehlerüberwachung

Keine

Kommentare

Keine

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23627)	42 75 92
	ld de, (23635)	237 91 83 92
	and a	167
	sbc hl,de	237 82
	ld b,h	68
	ld c,l	77
	ret	2Ø1

Wie es funktioniert

Die Routine lädt die Adresse des Variablenbereichs in Registerpaar hl. Dann lädt sie die Adresse des BASIC-Programms in de. Darauf wird de von hl subtrahiert, was die Programmlänge ergibt. Die Routine kopiert hl in bc und springt dann zu BASIC zurück.

Zeilenadresse

Länge: 29
Anzahl der Variablen: 1
Prüfsumme: 2351

Arbeitsweise

Diese Routine gibt die Adresse des ersten Zeichens aus, das nach einem REM-Zeichen in einer vorgegebenen Zeile steht.

Variablen

Name	Länge	Speicherzelle	Kommentar
line number	2	23296	Zeilennummer, in der "REM" stehen sollte

Aufruf

LET A = USR Adresse

Fehlerüberwachung

Wenn die vorgegebene Zeile nicht existiert oder wenn sie kein REM-Befehl ist, gibt die Routine den Wert Null aus.

Kommentare

Diese Routine kann verwendet werden, um die Adresse zu finden, in die Maschinencode eingePOKEt werden muß, wenn er in einen REM-Befehl gebracht werden soll.

Bei Aufruf wird die Variable A (man könnte jede beliebige Variable verwenden) auf die Adresse oder, wenn ein Fehler vorliegt, auf null gesetzt. Geben Sie keine Zeilennummern über 9999 ein!

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23296)	42 ∅ 91
	ld a,h	124
	or l	181
	jr z, error	4∅ 5
	call 651∅	2∅5 11∅ 25
	jr z, continue	4∅ 4
error	ld bc, ∅	1 ∅ ∅
	ret	2∅1
continue	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	ld a, 234	62 234
	cp (hl)	19∅
	jr nz, error	32 243
	inc hl	35
	ld b,h	68
	ld c,l	77
	ret	2∅1

Wie es funktioniert

Das Registerpaar hl wird mit der vorgegebenen Zeilennummer geladen. Wenn diese Zahl null ist, wird zu "error" (deutsch: Fehler) gesprungen, andernfalls wird die ROM-Routine in Adresse 651Ø aufgerufen. Beim Rücksprung von dieser Subroutine ist in hl die Adresse der Zeile enthalten. Wenn das Nullflag gesetzt ist, wird zu "continue" gesprungen. Wenn das Nullflag nicht gesetzt ist, gibt es die Zeile nicht, und die Routine "fällt durch" bis "error", wo bc mit null geladen wird und die Routine zu BASIC zurückspringt.

Wenn die Routine "continue" erreicht, wird hl um vier erhöht, damit es auf die erste Anweisung in der vorgegebenen Zeile zeigt. Wenn diese Anweisung nicht den Code 234 hat, springt die Routine zu "error". Wenn die Anweisung ein "REM" ist, wird hl erhöht, damit es auf das nächste Zeichen zeigt. Der Wert von hl wird in bc kopiert, und die Routine springt zu BASIC zurück.

Speicherkopie

Länge: 33
Anzahl der Variablen: 3
Prüfsumme: 4Ø22

Arbeitsweise

Diese Routine kopiert einen Speicherbereich von einer Adresse in eine andere.

Variablen

Name	Länge	Speicherzelle	Kommentar
start	2	23296	Adresse, von der kopiert werden soll
destination	2	23298	Adresse, in die kopiert werden soll
length	2	233ØØ	Anzahl der zu kopierenden Bytes

Aufruf

RAND USR Adresse

Fehlerüberwachung

Keine

Kommentare

Diese Routine kann verwendet werden, um "Zeichentrickfilme" mit folgender Methode zu produzieren:

(i) Man produziere den ersten Bildschirm mit Information.
(ii) Man kopiere das Display über RAMTOP.
(iii) Man wiederhole mit weiteren Bildschirmanzeigen.
(iv) Man kopiere die Bildschirmanzeigen eine nach der anderen in schneller Folge zurück.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23296)	42 ∅ 91
	ld de, (23298)	237 91 2 91
	ld bc, (233∅∅)	237 75 4 91
	ld a,b	12∅
	or c	177
	ret z	2∅∅
	and a	167
	sbc hl,de	237 82
	ret z	2∅∅
	add hl,de	25
	jr c, lddr	56 3
	ldir	237 176
	ret	2∅1
lddr	ex de,hl	235
	add hl,bc	9
	ex de,hl	235
	add hl,bc	9
	dec hl	43
	dec de	27
	lddr	237 184
	ret	2∅1

Wie es funktioniert

Die Adresse des ersten Speicherbytes, das kopiert werden soll, wird in Registerpaar hl geladen. Die Adresse, auf die kopiert werden soll, wird in de, die Anzahl der zu kopierenden Bytes in bc geladen. Wenn bc null ist oder hl = de, springt die Routine zu BASIC zurück. Wenn hl größer als de ist, wird der Speicherteil mit Hilfe der ldir-Anweisung kopiert. Dann springt die Routine zu BASIC zurück.

Wenn de größer als hl ist, wird bc – 1 zu beiden Registerpaaren addiert. Der Speicher wird mit dem lddr-Befehl kopiert, und die Routine springt zu BASIC zurück.

Wert Null für alle Variablen

Länge: 1Ø8
Anzahl der Variablen: Ø
Prüfsumme: 1Ø717

Arbeitsweise

Alle numerischen Variablen bekommen den Wert Null, alle dimensionierten Strings werden mit Leerzeichen gefüllt, und nichtdimensionierten Zeichenfolgen wird Länge Null zugeordnet. (Man nennt sie dann Nullstrings.)

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn es keine Variablen im Speicher gibt, springt die Routine sofort zu BASIC zurück.

Kommentare

Diese Routine ist eine sehr nützliche Hilfe beim Debugging, d. h. bei der Fehlerbeseitigung.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23627)	42 75 92
check	ld a, (hl)	126
	cp 128	254 128
	ret z	2ØØ
	ld de, 1	17 1 Ø
	bit 7,a	2Ø3 127
	jr nz, next bit	32 32
	bit 5,a	2Ø3 111
	jr z, string	4Ø 9
zero	ld b,5	6 5
next byte	inc hl	35
	ld (hl),d	114
	djnz next byte	16 252
	add hl,de	25
	jr check	24 232
string	inc hl	35
	ld c,(hl)	78

Marke	Assemblersprache	einzugebende Zahlen
	ld (hl),d	114
	inc hl	35
	ld b,(hl)	7∅
	ld (hl),d	114
	inc hl	35
delete	ld a,b	12∅
	or c	177
	jr z, check	4∅ 221
	push bc	197
	call 412∅	2∅5 24 16
	pop bc	193
	dec bc	11
	jr delete	24 244
next bit	bit 6,a	2∅3 119
	jr nz, bit 5	32 45
	bit 5,a	2∅3 111
	jr z, array	4∅ 7
number	inc hl	35
	bit 7, (hl)	2∅3 126
	jr z, number	4∅ 251
	jr zero	24 213
array	sub a	151
find length	push af	245
	inc hl	35
	ld c, (hl)	78
	inc hl	35
	ld b, (hl)	7∅
	inc hl	35
	push hl	229
	ld l, (hl)	11∅
	ld h,d	98
	add hl,hl	41
	pop de	2∅9
find elements	inc de	19
	dec bc	11
	dec hl	43
	ld a,h	124
	or l	181
	jr nz, find elements	32 249
	dec bc	11
rub out	inc de	19
	dec bc	11
	pop af	241
	push af	245
	ld (de),a	18
	ld a,b	12∅
	or c	177
	jr nz, rub out	32 247
	pop af	241

Marke	Assemblersprache	einzugebende Zahlen
restore	inc de	19
	ex de,hl	235
	jr check	24 164
bit 5	bit 5,a	2∅3 111
	jr z, string array	4∅ 5
	ld de, 14	17 14 ∅
	jr zero	24 17∅
string array	ld a, 32	62 32
	jr find length	24 21∅

Wie es funktioniert

Die Adresse des Anfangs des Variablenbereichs wird in Registerpaar hl geladen. Das in hl gespeicherte Byte wird in den Akkumulator geladen. Wenn dieses Byte den Wert 128 hat, macht die Routine einen Rücksprung zu BASIC, weil der Code 128 das Ende der Variablen kennzeichnet. Das Register de wird mit dem Wert eins geladen zur späteren Verwendung in der Routine. Wenn Bit 7 im Akkumulator auf eins gesetzt ist, springt die Routine zu "next bit", und dann springt sie zu "string", sofern Bit 5 auf null gesetzt ist.

Wenn "zero" erreicht werden soll, ohne in der Routine vorzuspringen, muß die gefundene Variable eine Zahl mit einem einbuchstabigen Namen sein. Register b wird auf fünf gesetzt, damit es als Zähler verwendet werden kann. Die Routine inkrementiert hl, damit es auf das nächste Byte zeigt und poket dieses mit null. Sie dekrementiert den Zähler, und wenn dieser nicht null ist, springt sie zurück zu "next byte". Dann wird de zu hl addiert, damit es auf die nächste Variable zeigt. Darauf springt die Routine zu "check" zurück.

Wenn die Routine "string" erreicht, inkrementiert sie hl, damit es auf diejenigen Bytes zeigt, welche die Länge der gefundenen Zeichenfolge enthalten. Die alte Länge wird in bc geladen, damit es als Zähler dienen kann, und die neue Länge wird auf null gesetzt. hl wird nochmals inkrementiert, damit es auf den Text der Zeichenfolge zeigt. Wenn der Zähler auf null steht, zeigt hl auf die nächste Variable. Deshalb wird zu "check" zurückgesprungen. Andernfalls wird bc auf dem Stapel abgelegt und die ROM-Routine in Adresse 412∅ aufgerufen, damit sie ein Zeichen streicht. Der Zähler wird dann vom Stapel zurückgeholt, dekrementiert, und es wird zu "delete" (deutsch: streichen) zurückgesprungen.

Bei "next bit" wird Bit 6 des Akkumulators überprüft. Wenn es auf eins steht, wird zu "bit 5" gesprungen, da ein Stringarray oder eine FOR/NEXT Laufvariable gefunden wurde. Wenn es auf null steht und Bit 5 auf null steht, wird zu "array" gesprungen.

Wenn die Routine "number" erreicht, muß die gefundene Variable eine Zahl mit einem mehrbuchstabigen Namen sein. Das Registerpaar hl wird inkrementiert, damit es auf das nächste Byte zeigt. Dies wird wiederholt, bis ein Byte vorgefunden wird, in welchem Bit 7 auf eins gesetzt ist. Wenn dies gefunden ist, springt die Routine zu "zero", um die Variable mit null zu laden.

Wenn die Routine "array" erreicht, wird der Akkumulator mit null geladen, denn dies ist der Wert, auf den die Elemente später gesetzt werden müssen.

Bei "find length" (deutsch: die Länge finden) wird der Akkumulator auf dem Stapel gespeichert und hl inkrementiert, damit es auf die Bytes, welche die Arraylänge enthalten, zeigt. Diese werden in bc kopiert, um als Zähler Verwen-

dung zu finden. hl wird noch einmal inkrementiert, so daß es jetzt auf das Byte zeigt, das die Anzahl der Dimensionen enthält. Dann wird hl auf dem Stapel gespeichert. Die Routine lädt hl mit der Anzahl der Dimensionen und multipliziert es mit zwei. Dann setzt sie de auf die auf dem Stapel gespeicherte Adresse. Registerpaar de wird daraufhin hl-mal inkrementiert, und bc wird (hl + 1)-mal dekrementiert. Dann wird de inkrementiert und bc wieder dekrementiert. Nun zeigt de auf das nächste Element des Arrays, und die Anzahl derjenigen Bytes, die vor dem Erreichen des Endes übrig sind, steht in bc. Der Akkumulator wird vom Stapel zurückgeholt und in de eingepoket. Der Zähler in bc wird dekrementiert, und wenn er nicht null ist, springt die Routine zurück zu "rub out" (deutsch: auslöschen). Die Routine stellt dann den Wert in hl so ein, daß er auf die nächste Variable zeigt, und springt zu "check".

Bei "bit 5" nimmt die Routine einen Test vor, um herauszufinden, ob ein Stringarray vorgefunden wurde. Wenn ja, setzt sie den Akkumulator auf den Code des Leerzeichens und springt zu "find length". Wenn die Routine diesen Punkt erreicht, muß die Variable eine FOR/NEXT Laufvariable sein. de wird auf 14 gesetzt, so daß die Summe davon und (hl + 5) auf die nächste Variable zeigt. Die Routine springt dann zurück zu "zero".

Variablenliste

Länge: 94
Anzahl der Variablen: Ø
Prüfsumme: 1Ø295

Arbeitsweise

Diese Routine listet die Namen aller Variablen auf, die gegenwärtig im Speicher stehen.

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn keine Variablen im Speicher stehen, springt die Routine sofort zu BASIC zurück.

Kommentare

Diese Routine ist eine nützliche Hilfe für Debugging von Programmen, besonders bei langen oder komplizierten Programmen.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	res Ø, (IY + 2)	253 2Ø3 2 134
	ld hl, (23627)	42 75 92
next variable	ld a, 13	62 13
	rst 16	215
	ld a, 32	62 32
	rst 16	215
	ld a, (hl)	126
	cp 128	254 128
	ret z	2ØØ
	bit 7,a	2Ø3 127
	jr z, bit 5	4Ø 62
	bit 6,a	2Ø3 119
	jr z, next bit	4Ø 31
	bit 5,a	2Ø3 111
	jr z, string array	4Ø 9
	sub 128	214 128
	ld de, 19	17 19 Ø
print	rst 16	215
	add hl,de	25
	jr next variable	24 225
string array	sub 96	214 96
	rst 16	215
	ld a, 36	62 36
brackets	rst 16	215
	ld a, 4Ø	62 4Ø
	rst 16	215
	ld a, 41	62 41
pointers	inc hl	35
	ld e, (hl)	94
	inc hl	35
	ld d, (hl)	86
	inc hl	35
	jr print	24 234
next bit	bit 5,a	2Ø3 111
	jr z, array	4Ø 19
	sub 64	214 64
	rst 16	215
next character	inc hl	35
	ld a, (hl)	126
	bit 7,a	2Ø3 127
	jr nz, last character	32 3
	rst 16	215
	jr next character	24 247
last character	sub 128	214 128
jump	de, 6	17 6 Ø
	jr print	24 211
array	sub 32	214 32
	jr brackets	24 216

Marke	Assemblersprache	einzugebende Zahlen
bit 5	bit 5,a	2Ø3 111
	jr nz, jump	32 243
	add a, 32	198 32
	rst 16	215
	ld a, 36	62 36
	jr pointers	24 211

Wie es funktioniert

Bit Ø des Bytes in Adresse 23612 wird gelöscht. Damit will man sicherstellen, daß mit PRINT gedruckte Zeichen im oberen Teil des Bildschirms erscheinen. hl wird mit der Adresse des Variablenbereichs, der Akkumulator mit dem ENTER-Zeichen geladen. Dieses wird mittels der ROM-Routine in Adresse 16 gedruckt. Der Akkumulator wird dann mit dem Code eines Leerzeichens geladen, und dieses wird mit Hilfe derselben Routine gedruckt.

Der Akkumulator wird mit dem Byte geladen, das in der Adresse in hl gespeichert ist. Wenn es den Wert 128 hat, macht die Routine einen Rücksprung zu BASIC, weil das Ende des Variablenbereichs erreicht wurde.

Wenn Bit 7 des Akkumulators auf null steht, springt die Routine zu "bit 5", weil ein String oder eine Zahl, deren Name einbuchstabig ist, vorgefunden wurde. Bit 6 des Akkumulators wird getestet. Wenn es auf null steht, springt die Routine zu "next bit", weil sie ein Array oder eine Zahl gefunden hat, deren Name mehrbuchstabig ist. Wenn Bit 5 des Akkumulators null ist, springt die Routine zu "string array".

Die Routine kommt bis hierher, sofern die gefundene Variable die Laufvariable einer FOR/NEXT-Schleife ist. 128 wird vom Akkumulator abgezogen. Das Ergebnis ist der Code desjenigen Zeichens, das gedruckt werden soll. Registerpaar de wird mit 19 geladen, damit es auf die nächste Variable zeigt, wenn es zu hl addiert wird. Die Routine druckt das Zeichen im Akkumulator, addiert de zu hl und springt zurück zu "next variable".

Wenn die Routine "string array" erreicht, wird 96 vom Akkumulator abgezogen. Das ergibt den Code des Namens des gefundenen Arrays. Dies wird mittels der ROM-Routine gedruckt. Ein Dollarzeichen und eine offene Klammer werden dann gedruckt, und der Akkumulator wird mit dem Code für eine geschlossene Klammer geladen. Die Routine erhöht hl, damit es auf die Bytes zeigt, in denen die Länge des Arrays steht. Sie lädt diese in de, so daß man durch Addition zu hl die Adresse der nächsten Variablen erhält. Dann springt die Routine zu"print", wo die geschlossene Klammer gedruckt wird, und de zu hl addiert wird.

Bei "next bit" wird Bit 5 des Akkumulators getestet. Wenn es auf null steht, springt die Routine zu "array". Steht es dagegen auf eins, wurde eine Zahl gefunden, deren Name mehrbuchstabig ist. Die Routine subtrahiert dann 64 vom Akkumulator und druckt das daraus entstehende Zeichen. Dann macht sie eine Schleife, bei der sie jedes vorgefundene Zeichen druckt, bis sie eines antrifft, in dem Bit 7 auf eins steht. Die Routine zieht 128 vom Zeichencode ab, lädt de mit dem Abstand zur nächsten Variablen und springt zu "print".

Wenn die Routine ein Array findet, subtrahiert sie 32 vom Akkumulator, damit sich der richtige Code ergibt. Dann springt sie zu "brackets" (deutsch: Klammern).

Bei "bit 5" macht die Routine eine Schleife zurück zu "jump", sofern eine Zahl mit einem einbuchstabigen Namen gefunden wurde.

Bis hierher kommt die Routine nur, wenn die vorgefundene Variable ein String ist. Der zu druckende Code ergibt sich, indem man 32 vom Akkumulator subtrahiert. Schließlich lädt die Routine den Akkumulator mit dem Code für ein Dollarzeichen und springt zu"pointers".

Suchen und Auflisten

Länge: 155
Anzahl der Variablen: 2
Prüfsumme: 17221

Arbeitsweise

Diese Routine durchsucht ein BASIC-Programm nach einem vom Benutzer vorgegebenen String und listet jede Zeile auf, in der er vorkommt.

Variablen

Name	Länge	Speicherzelle	Kommentar
data start	2	23296	Adresse des ersten Datenbytes
string length	1	23298	Anzahl von Zeichen im String

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn kein BASIC-Programm im Speicher steht oder der String die Länge Null hat, springt die Routine direkt zu BASIC zurück.

Kommentare

Die Laufzeit dieser Routine ist proportional zur Länge des Strings und zur Länge des BASIC-Programms.

Der String, nach dem gesucht wird, sollte über RAMTOP eingepoket, die Adresse des ersten Bytes in dem String sollte in 23296/7 eingepoket werden. Die Länge des Strings sollte in 23298 gespeichert werden.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	res ∅, (IY + 2)	253 2∅3 2 134
	ld ix, (23296)	221 42 ∅ 91
	ld hl, (23635)	42 83 92
restart	ld a, (23298)	58 2 91
	ld e, a	95
	cp ∅	254 ∅
	ret z	2∅∅
	push hl	229
restore	push ix	221 229
	pop bc	193
	ld d, ∅	22 ∅
	inc hl	35
	inc hl	35
	inc hl	35
check	inc hl	35
	push de	213
	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	add hl,de	25
	pop de	2∅9
	jr c, enter	56 4
	pop hl	225
	ret	2∅1
long jump	jr restart	24 223
enter	ld a, (hl)	126
	cp 13	254 13
	jr nz, number	32 5
	inc hl	35
	pop bc	193
	push hl	229
	jr restore	24 221
number	call 6326	2∅5 182 24
	jr nz, compare	32 8
	dec hl	43
different	push ix	221 229
	pop bc	193
	ld d, ∅	22 ∅
	jr check	24 216
compare	ld a, (bc)	1∅
	cp (hl)	19∅
	jr nz, different	32 245
	inc bc	3
	inc d	2∅
	ld a,d	122
	cp e	187
	jr nz, check	32 2∅6
	ld a, 13	62 13
	rst 16	215

Marke	Assemblersprache	einzugebende Zahlen
	pop hl	225
	push hl	229
	ld b, (hl)	70
	inc hl	35
	ld l, (hl)	110
	ld h, b	96
	ld de, 1000	17 232 3
	ld a, 47	62 47
thousands	inc a	60
	and a	167
	sbc hl,de	237 82
	jr nc, thousands	48 250
	add hl, de	25
	rst 16	215
	ld de, 100	17 100 0
	ld a, 47	62 47
hundreds	inc a	60
	and a	167
	sbc hl, de	237 82
	jr nc, hundreds	48 250
	add hl,de	25
	rst 16	215
	ld de, 10	17 10 0
	ld a, 47	62 47
tens	inc a	60
	and a	167
	sbc hl,de	237 82
	jr nc, tens	48 250
	add hl, de	25
	rst 16	215
	ld a,l	125
	add a, 48	198 48
	rst 16	215
	pop hl	225
	inc hl	35
	inc hl	35
	inc hl	35
next character	inc hl	35
	ld a, (hl)	126
line end	cp 13	254 13
	jr nz, chr 14	32 4
	rst 16	215
	inc hl	35
	jr long jump	24 155
chr 14	call 6326	205 182 24
	jr z, line end	40 243
	cp 32	254 32
	jr c, next character	56 237
	rst 16	215
	jr next character	24 234

Wie es funktioniert

Die Routine löscht Bit Ø des in Adresse 23612 gespeicherten Bytes, um sicherzustellen, daß gedruckte Zeichen im oberen Teil des Bildschirms erscheinen. Dann lädt sie ix mit der Adresse des ersten Datenbytes. Dadurch kann die Adresse in andere Registerpaare geladen werden mit weniger Bezugnahme auf den Druckerpuffer. Die Adresse des BASIC-Programms wird in hl geladen.

Der Akkumulator wird mit der Länge des Strings geladen, und dies wird in Register e kopiert. Wenn seine Länge null ist, springt die Routine unmittelbar zu BASIC zurück. Die Adresse in hl wird auf dem Stapel gespeichert, wobei darin die Speicherposition der gegenwärtig durchsuchten Zeile steht.

Die Datenadresse wird aus ix in bc kopiert, damit man besseren Zugriff zu ihr hat. Register d wird mit null geladen, d. h. der Anzahl derjenigen gefundenen Zeichen, die mit den eingegebenen Daten übereinstimmen. Die Routine erhöht das Registerpaar hl um drei, damit es auf das höherwertige Byte der Zeilenlänge zeigt. Sie inkrementiert hl, weil es auf das nächste Zeichen zeigen soll. Das Registerpaar de wird auf dem Stapel abgelegt.

de wird mit der Adresse des Variablenbereichs geladen, und dies wird von hl abgezogen. Wenn das Ergebnis negativ ist, springt die Routine zu "enter", nachdem sie in hl den alten Wert wiederhergestellt hat und de vom Stapel geholt hat. Wenn das Ergebnis positiv ist, stellt die Routine die ursprüngliche Größe des Stapels wieder her und springt zu BASIC zurück, da das Ende des BASIC-Programms erreicht wurde.

Bei "enter" wird der Akkumulator mit dem in der Adresse in hl gespeicherten Byte geladen. Wenn das nicht das ENTER-Zeichen ist, springt die Routine zu "number". Falls sie das ENTER-Zeichen antrifft, erhöht sie hl, damit es auf den Anfang der nächsten Zeile zeigt. Die Adresse der vorigen Zeile wird vom Stapel entfernt und durch den neuen Wert in hl ersetzt. Dann springt die Routine zu "restore".

Bei "number" wird die ROM-Routine in Adresse 6326 aufgerufen. Wenn das Zeichen im Akkumulator das NUMBER-Zeichen ist, wird hl erhöht. Es zeigt dann auf das erste Zeichen nach der Binärdarstellung der von der ROM-Routine gefundenen Zahl. Wenn die Routine kein NUMBER-Zeichen findet, springt sie zu "compare"; andernfalls dekrementiert sie hl und "fällt durch" bis zu "different" (deutsch: verschieden). bc wird aus ix kopiert; die Anzahl der gefundenen Zeichen wird auf null gesetzt, und die Routine springt zu "check".

Bei "compare" wird der Akkumulator mit dem in der Adresse in bc gespeicherten Byte geladen. Wenn dieses nicht mit dem in der Adresse in hl gespeicherten Byte übereinstimmt, springt die Routine zurück zu "different". bc wird inkrementiert, damit es auf das nächste Datenbyte zeigt, und die Anzahl der gefundenen Zeichen wird erhöht. Wenn dies nicht mit der Länge des Strings übereinstimmt, springt die Routine zurück zu "check".

Der Akkumulator wird mit dem Code des ENTER-Zeichens geladen, und dieser wird unter Verwendung der ROM-Routine in Adresse 16 ausgedruckt. Die Adresse der zu druckenden Zeile wird aus dem Stapel in hl geladen. Die Zeilennummer wird dann über Register b in hl kopiert. 1ØØØ wird in de geladen, und der Code des Zeichens "Ø" minus eins wird in den Akkumulator geladen. Der Akkumulator wird inkrementiert, und de wird mehrmals von hl subtrahiert bis hl negativ ist. Dann addiert die Routine einmal de zu hl, was eine positive Zahl ergibt. Sie druckt dann das Zeichen im Akkumulator.

Der obige Prozeß wird mit de = 1ØØ und de = 1Ø wiederholt. Dann wird

der Rest in den Akkumulator geladen, 48 dazu addiert und das daraus entstehende Zeichen gedruckt.

Die Adresse des Zeilenanfangs wird vom Stapel zurückgeholt und in hl geladen. Dann erhöht die Routine hl, damit es auf das höherwertige Byte der Zeilenlänge zeigt. Sie inkrementiert hl und lädt das Byte, das darinsteht, in den Akkumulator. Wenn dieses Byte nicht das ENTER-Zeichen ist, wird zu "chr 14" gesprungen, sonst wird ENTER gedruckt. Die Routine inkrementiert hl und springt zu "restart" zurück.

Bei "chr 14" wird die ROM-Routine in Adresse 6326 aufgerufen. Wenn das Zeichen im Akkumulator das NUMBER-Zeichen ist, wird hl erhöht, damit es auf das erste Zeichen nach der gefundenen Zahl zeigt. Dieses Zeichen wird in den Akkumulator geladen, und die Routine springt zu "line end". Wenn das Zeichen im Akkumulator einen Code unter 32 hat, springt die Routine zurück zu "next character". Wenn der Code über 31 liegt, druckt sie das gefundene Zeichen und springt zu "next character".

Suchen und Ersetzen

Länge: 85
Anzahl der Variablen: 3
Prüfsumme: 8518

Arbeitsweise

Diese Routine durchsucht ein BASIC-Programm nach einem String und ersetzt ihn, jedesmal wenn er auftritt, durch einen anderen String gleicher Länge.

Variablen

Name	Länge	Speicherzelle	Kommentar
old data start	2	23296	Adresse des zu ersetzenden Strings
string length	1	23298	Länge des zu ersetzenden Strings
new data start	2	23299	Adresse des "Ersatzstrings"

Aufruf

RAND USR Adresse

Fehlerüberwachung

Wenn die Länge des Strings null ist oder wenn es kein BASIC-Programm im Speicher gibt, springt die Routine sofort zu BASIC zurück.

Kommentare

Die Laufzeit dieser Routine hängt von der Länge des Strings und des BASIC-Programms im Speicher ab.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld ix, (23296)	221 42 Ø 91
	ld hl, (23635)	42 83 92
	ld a, (23298)	58 2 91
	ld e, a	95
	cp Ø	254 Ø
	ret z	2ØØ
	dec hl	43
new line	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	jr reset	24 23
check	inc hl	35
	push de	213
	ld de, (23627)	237 91 75 92
	and a	167
	sbc hl,de	237 82
	add hl,de	25
	pop de	2Ø9
	ret nc	2Ø8
	ld a, (hl)	126
	cp 13	254 13
	jr z, new line	4Ø 233
	call 6326	2Ø5 182 24
	jr nz, compare	32 8
	dec hl	43
reset	push ix	221 229
	pop bc	193
	ld d, Ø	22 Ø
	jr check	24 226
compare	ld a, (bc)	1Ø
	cp (hl)	19Ø
	jr nz, reset	32 245
	inc bc	3
	inc d	2Ø
	ld a,d	122
	cp e	187
	jr nz, check	32 216
	push hl	229
	ld d, Ø	22 Ø
	and a	167
	sbc hl,de	237 82

Marke	Assemblersprache	einzugebende Zahlen
	ld d,e	83
	ld bc, (23299)	237 75 3 91
	inc d	20
next char	inc hl	35
	dec d	21
	jr z, finish	40 5
	ld a, (bc)	10
	ld (hl),a	119
	inc bc	3
	jr next char	24 247
finish	pop hl	225
	jr reset	24 215

Wie es funktioniert

Das ix-Register wird mit der Adresse des Strings geladen, nach dem gesucht werden soll. Dies sollte über RAMTOP sein. Die Adresse des Programmbereichs wird in hl geladen und die Länge des Strings wird in den Akkumulator geladen. Die Länge wird auch in Register e kopiert, damit man sie später im Programm verwenden kann. Wenn die Stringlänge null ist, springt die Routine zu BASIC zurück. Sie stellt hl so ein, daß es auf das höherwertige Byte der Zeilenlänge zeigt und springt zu "reset".

Bei "check" wird hl inkrementiert, damit es auf das nächste Zeichen zeigt. de wird auf dem Stapel gespeichert und mit der Adresse des Variablenbereichs geladen. Wenn hl nicht kleiner als de ist, hat die Routine das Programmende erreicht und springt deshalb zu BASIC zurück, nachdem sie de aus dem Stapel geholt hat.

Der Akkumulator wird mit dem von hl adressierten Zeichen geladen. Wenn dies das ENTER-Zeichen ist, macht die Routine eine Schleife zu "new line". Wenn im Akkumulator nicht das NUMBER-Zeichen (Zeichen 14) steht, springt sie zu "compare"; andernfalls erhöht sie hl um fünf, so daß es auf das fünfte Byte der gefundenen Zahl zeigt.

Bei "reset" wird bc mit der Adresse des Strings geladen, der gesucht werden soll. Register d wird auf null gesetzt und enthält dadurch die Anzahl der Zeichen im String, die bis dahin gefunden wurden. Die Routine springt dann zu "check" zurück.

Bei "compare" wird der Akkumulator mit dem Zeichen des Strings geladen, auf das bc zeigt. Wenn dies von dem durch hl adressierten Byte verschieden ist, springt die Routine zu "reset". bc wird inkrementiert und zeigt dadurch auf das nächste Zeichen im String. Der Zähler in Register d wird inkrementiert. Wenn dies mit der Länge des Strings nicht übereinstimmt, springt die Routine zurück zu "check".

Wenn die Routine den String gefunden hat, legt sie hl auf dem Stapel ab, so daß sie von dieser Adresse ausgehend von neuem nach dem String sucht. de wird mit der Länge des Strings geladen, und dies wird von hl subtrahiert. Daraus ergibt sich die Startadresse minus eins. Die Stringlänge wird dann in d geladen, damit es als Zähler dienen kann. bc wird mit der Startadresse des neuen Strings geladen und Register d wird inkrementiert. Register hl wird ebenfalls inkrementiert, und zeigt dadurch auf die nächste Speicherzelle. Der Zähler wird dekre-

mentiert. Wenn im Zähler null steht, holt die Routine hl vom Stapel zurück und springt zu"reset", um die nächste Stelle, an welcher der String auftritt, zu finden. Der Akkumulator wird mit demjenigen Zeichen geladen, auf das bc zeigt, und dies wird in hl eingepoket. Die Routine inkrementiert bc, damit es auf das nächste Zeichen zeigt und springt zu "next char" zurück.

ROM-Suche

Länge: 58
Anzahl der Variablen: 3
Prüfsumme: 6533

Arbeitsweise

Diese Routine durchsucht den ROM nach einem Bytemuster, das vom Benutzer vorgegeben wurde.

Variablen

Name	Länge	Speicherzelle	Kommentar
search start	2	23296	Anfang des Speicherbereiches, der durchsucht werden soll
string length	1	23298	Anzahl der Bytes im String
data start	2	23299	Adresse des Strings im RAM

Aufruf

PRINT USR Adresse

Fehlerüberwachung

Wenn die Länge des Strings null ist, springt die Routine sofort zu BASIC zurück und gibt die Adresse des Datenanfangs aus. Wenn der String nicht gefunden wird, gibt sie den Wert 65535 aus.

Kommentare

Diese Routine kann beim Schreiben von Maschinencode-Programmen verwendet werden, um Subroutinen im ROM zu finden, sofern der Benutzer schon weiß, wie Teile dieser Routine abgefaßt sind.

Da für den größten Teil des Spectrum ROM nur der ZX81 umgearbeitet wurde, können Programme, die ursprünglich für den ZX81 geschrieben wur-

den, leicht angepaßt werden. Zum Beispiel ruft die Routine "Zeilenadresse" die ROM-Routine in Adresse 651∅ auf. Beim ZX81 beginnt die Routine in Adresse 252∅. Wenn man diese Routine disassembliert, ergibt sich

```
push hl
ld hl, program
ld d,h*
ld e,l*
pop bc*
call ∅9EA
```

Die drei mit einem Stern gekennzeichneten Bytes sind beim Spectrum gleich und können mit Hilfe der Such-Routine gefunden werden. In der Tat gibt die Routine 6514 aus, d. h. die Summe von 4 und der Startadresse der verlangten ROM-Routine.

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	ld hl, (23296)	42 ∅ 91
	ld de, (23298)	237 91 2 91
restart	ld bc, (23299)	237 75 3 91
	ld a,e	123
	cp ∅	254 ∅
	ret z	2∅∅
	push hl	229
	ld d, ∅	22 ∅
compare	ld a, (bc)	1∅
	cp (hl)	19∅
	jr z, match	4∅ 25
	pop hl	225
	inc hl	35
	push de	213
	push hl	229
	ld hl, 16384	33 ∅ 64
	ld, ∅	22 ∅
	and a	167
	sbc hl,de	237 82
	inc hl	35
	pop de	2∅9
	and a	167
	sbc hl,de	237 82
	ex de,hl	235
	pop de	2∅9
	jr nz, restart	32 22∅
	ld bc, 65535	1 255 255
	ret	2∅1

Marke	Assemblersprache	einzugebende Zahlen
match	inc d	2Ø
	ld a,d	122
	cp e	187
	jr nz, next byte	32 2
	pop bc	193
	ret	2Ø1
next byte	inc hl	35
	inc bc	3
	jr compare	24 216

Wie es funktioniert

Das Registerpaar hl wird mit der Adresse der ersten Speicherzelle geladen, die überprüft werden soll. Will man herausfinden, wo der String zum ersten Mal im ROM auftritt, so sollte hl auf null gesetzt werden. Das Register e wird mit der Anzahl der Bytes im String geladen, nach dem gesucht wird. Die Routine lädt die Adresse des Strings im RAM, die vom Benutzer eingegeben wurde, in Registerpaar bc. Sie lädt die Stringlänge in den Akkumulator und wenn sie null ist, macht sie einen Rücksprung zu BASIC.

Die Adresse in hl wird auf dem Stapel abgespeichert. Der Akkumulator wird mit dem Byte geladen, auf welches Registerpaar bc zeigt. Wenn es mit dem Byte, auf welches hl zeigt, übereinstimmt, wird zu "match" (deutsch: übereinstimmen) gesprungen. Wenn die beiden Bytes verschieden sind, wird hl mit der Adresse auf dem Stapel geladen. Dies wird dann inkrementiert, damit es auf die nächste Speicherzelle zeigt.

Die Register de und hl werden auf dem Stapel gespeichert. hl wird mit der Adresse des ersten RAM-Bytes, de mit der Länge des Strings geladen. de wird von hl subtrahiert; das ergibt die höchstmögliche Startadresse des Strings. Diese wird inkrementiert, damit es auf die erste Adresse zeigt, in welcher der String nicht abgelegt sein kann.

Die Adresse auf dem Stapel wird in de geladen, und diese wird von hl subtrahiert. Die Routine merkt sich das Ergebnis dieser Operation, während sie hl mit dem Inhalt von de und mit der Zahl auf dem Stapel lädt. Wenn das Ergebnis null ist, lädt die Routine bc mit 65535 und springt zu BASIC zurück, da der String im ROM nicht existiert. Wenn das Ergebnis nicht null ist, macht die Routine eine Schleife zurück zu "restart".

Bei "match" wird Register d inkrementiert, so daß es die Anzahl derjenigen Bytes enthält, welche die Routine schon als übereinstimmend erkannt hat. Wenn dies gleich der Länge des Strings ist, holt die Routine bc vom Stapel zurück und springt zu BASIC zurück. Wenn in Register d nicht die Länge des Strings steht, werden sowohl hl als auch bc inkrementiert, damit sie auf die nächsten Bytes zeigen. Die Routine springt dann zu "compare".

Instr$

Länge: 168
Anzahl der Variablen: Ø
Prüfsumme: 19875

Arbeitsweise

Diese Routine gibt die Position eines Unterstrings (B$) in einem Hauptstring (A$) aus. Wenn ein Fehler vorliegt, gibt sie null aus.

Aufruf

Let P = USR Adresse

Fehlerüberwachung

Wenn einer der beiden Strings nicht existiert, die Länge des Unterstrings größer als die des Hauptstrings ist oder die Länge des Unterstrings null ist, gibt die Routine den Wert Null aus.

Wenn kein Fehler vorliegt, aber der Unterstring nicht im Hauptstring gefunden werden kann, gibt die Routine ebenfalls den Wert Null aus.

Kommentare

Bei Rücksprung von der Maschinencode-Routine steht in Variable P (man könnte jede andere Variable verwenden) der Rücksprungs-(Ausgabe-)Wert. Die Strings, auf die man sich bezieht, können nicht mit einer DIM-Anweisung "dimensioniert" werden wie Zeichenarrays. Will man die verwendeten Strings ändern, so muß man die Zahlen, die mit einem Stern versehen sind, ändern. 66* ist der Unterstring, 65* der Hauptstring. Um diese zu ändern, ersetze man die Zahlen durch die Codes der verlangten Zeichen (A bis Z = 65 bis 9Ø).

Maschinencode Listing

Marke	Assemblersprache	einzugebende Zahlen
	sub a	151
	ld b,a	71
	ld c,a	79
	ld d, a	87
	ld e,a	95
	ld hl, (23627)	42 75 92
next variable	ld a, (hl)	126
	cp 128	254 128
	jr z, not found	4Ø 95
	bit 7,a	2Ø3 127

Marke	Assemblersprache	einzugebende Zahlen
	jr nz, for-next	32 41
	cp 96	254 96
	jr nc, number	48 29
	cp 65	254 65*
	jr nz, substring	32 2
	ld d,h	84
	ld e,l	93
substring	cp 66	254 66*
	jr nz, check	32 2
	ld b,h	68
	ld c,l	77
check	ld a,d	122
	or e	179
	jr z, string	40 4
	ld a, b	120
	or c	177
	jr nz, found	32 38
string	push de	213
	inc hl	35
	ld e, (hl)	94
	inc hl	35
	ld d, (hl)	86
add	add hl,de	25
	pop de	209
	jr increase	24 5
number	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
	inc hl	35
increase	inc hl	35
	jr next variable	24 206
for-next	cp 224	254 224
	jr c, next bit	56 6
	push de	213
	ld de, 18	17 18 0
	jr add	24 234
next bit	bit 5,a	203 111
	jr z, string	40 225
next byte	inc hl	35
	bit 7, (hl)	203 126
	jr z, next byte	40 251
	jr number	24 227
found	ex de,hl	235
	inc hl	35
	inc hl	35
	push hl	229
	push hl	229
	inc bc	3
	push bc	197

Marke	Assemblersprache	einzugebende Zahlen
	ld a, (bc)	1∅
	ld e,a	95
	inc bc	3
	ld a, (bc)	1∅
	ld d,a	87
	or e	179
	jr z, zero length	4∅ 11
	push de	213
	ld a, (hl)	126
	dec hl	43
	ld l, (hl)	11∅
	ld h,a	1∅3
	and a	167
	sbc hl,de	237 82
	jr nc, continue	48 8
	pop bc	193
zero length	pop bc	193
	pop bc	193
error	pop bc	193
not found	ld bc, ∅	1 ∅ ∅
	ret	2∅1
continue	pop ix	221 225
	pop bc	193
	ex de,hl	235
	pop hl	225
	inc bc	3
	inc bc	3
save	inc hl	35
	push hl	229
	push bc	197
	push ix	221 229
	push de	213
compare	ld a, (bc)	1∅
	cp (hl)	19∅
	jr z, match	4∅ 12
	pop de	2∅9
	pop ix	221 225
	pop bc	193
	pop hl	225
	ld a,d	122
	or e	179
	jr z, error	4∅ 225
	dec de	27
	jr save	24 234
match	inc hl	35
	inc bc	3
	push hl	229
	dec ix	221 43
	push ix	221 229
	pop hl	225

Marke	Assemblersprache	einzugebende Zahlen
	ld a,h	124
	or l	181
	pop hl	225
	jr nz, compare	32 227
	pop de	2Ø9
	pop de	2Ø9
	and a	167
	sbc hl,de	237 82
	pop de	2Ø9
	pop de	2Ø9
	pop de	2Ø9
	and a	167
	sbc hl,de	237 82
	ld b,h	68
	ld c,l	77
	ret	2Ø1

Wie es funktioniert

Der Akkumulator, das Registerpaar bc und das Registerpaar de werden alle mit null geladen. Später wird bc auf die Adresse von B$ und de auf die Adresse von A$ gesetzt. hl wird mit der Adresse des Variablenbereichs geladen.

Die Routine lädt den Akkumulator mit dem von hl adressierten Byte. Wenn im Akkumulator 128 steht, springt die Routine zu "not found" (deutsch: nicht gefunden), da das Ende des Variablenbereiches erreicht ist. Wenn Bit 7 des Akkumulators auf eins steht, wird zu "for-next" gesprungen, weil die gefundene Variable kein String oder keine Zahl ist, deren Name einbuchstabig ist. Wenn im Akkumulator eine Zahl über 95 steht, springt die Routine zu "number".

Wenn die Routine bis hierher gelangt, muß sie einen String gefunden haben. Wenn im Akkumulator 65 abgelegt ist, hat sie die Position von A$ schon ausgemacht, und sie kopiert den Inhalt von hl in de. Steht im Akkumulator 66, ist B$ gefunden worden, und hl wird in bc kopiert. Sofern weder in de noch in bc null steht, sind beide Strings gefunden worden. Deshalb springt die Routine dann zu "found".

Sobald die Routine "string" erreicht, wird de auf dem Stapel abgelegt und mit der Länge des vorgefundenen Strings geladen. Dies wird zu der Adresse des höherwertigen Bytes der Stringlänge addiert und in hl gespeichert. Die Routine holt de vom Stapel zurück und springt zu "increase".

Bei "number" wird hl fünfmal inkrementiert, damit es auf das letzte Byte einer vorgefundenen Zahl zeigt. Dann inkrementiert die Routine hl, weil es auf die nächste Variable zeigen soll. Danach springt sie zu "next variable".

Bei "for-next" wird zu "next bit" gesprungen, wenn im Akkumulator eine Zahl kleiner als 224 steht, denn die vorgefundene Variable ist keine FOR-NEXT Schleifen-Laufvariable. Wenn der Wert im Akkumulator größer als 223 ist, addiert die Routine achtzehn zu hl, damit es auf das letzte Byte der Laufvariablen zeigt. Die Routine macht dann eine Schleife zu "increase".

Wenn die Routine "next bit" erreicht und Bit 5 des Akkumulators auf null steht, springt sie zu "string", damit hl mit der Adresse der folgenden Variablen geladen wird, weil ein Array gefunden wurde.

Wenn die Routine "next byte" erreicht, ist eine Zahl mit einem mehrbuchstabigen Namen gefunden worden. Deshalb erhöht sie hl, bis es auf das letzte Zeichen des Variablennamens zeigt; dann springt sie zu "number".

Bei "found" wird hl mit der Adresse von A$ geladen, und dies wird zweimal inkrementiert, wobei sich dadurch die Adresse des höherwertigen Bytes der Stringlänge ergibt. Dieser Wert wird dann zweimal auf dem Stapel abgespeichert. bc wird inkrementiert und zeigt somit auf das niederwertige Byte der Stringlänge für B$. Die Adresse in bc wird dann auf dem Stapel abgelegt. Die Routine lädt de mit der Länge von B$, und wenn dies null ist, springt sie zu "zero length". Dann wird de auf dem Stapel abgelegt. Die Routine lädt hl mit der Länge von A$, und wenn dies nicht kleiner als de ist, springt sie zu "continue". Die Routine bringt den Stapel dann wieder auf seine ursprüngliche Größe, lädt bc mit null und springt zu BASIC zurück.

Bei "continue" wird ix auf die Länge von B$, bc auf die Adresse des niederwertigen Bytes der Länge für B$ gesetzt. Die Längendifferenz von A$ und B$ wird in de geladen; die Adresse des höherwertigen Bytes der Länge für A$ wird in hl geladen. Dann inkrementiert die Routine bc zweimal, was die Adresse des ersten Zeichens in B$ ergibt. hl wird inkrementiert und zeigt auf das nächste Zeichen von A$.

Dann werden hl, bc, ix und de auf dem Stapel gespeichert. Die Routine lädt den Akkumulator mit dem von bc adressierten Byte. Wenn dieses mit dem von hl adressierten Byte übereinstimmt, springt sie zu "match". Dann werden de, ix, bc und hl vom Stapel zurückgeholt. Wenn in de null steht, springt die Routine zu "error", da B$ nicht in A$ vorkommt. Der Zähler in de wird dann dekrementiert, und die Routine macht eine Schleife zurück zu "save".

Wenn die Routine "match" erreicht, werden sowohl hl als auch bc inkrementiert und zeigen auf die nächsten Zeichen von A$ beziehungsweise B$. hl wird dann auf dem Stapel abgelegt. Die Routine dekrementiert den Zähler in ix und, nachdem hl vom Stapel zurückgeholt wurde, springt sie zu "compare" zurück, sofern in ix keine Null steht.

Wenn die Routine bis hierher gelangt, hat sie mit Sicherheit herausgefunden, daß B$ in A$ vorkommt. Die Länge von B$ wird dann von hl abgezogen. Dann wird die Adresse des höherwertigen Bytes der Länge für A$ von hl abgezogen. Das Ergebnis ist die Position von B$ in A$. Die Routine kopiert dies in Registerpaar bc und springt dann zu BASIC zurück.

Anhang A

In diesem Anhang gibt es zwei wichtige Tabellen mit Befehlen. Tabelle A2 führt die 1-Byte-Befehle und diejenigen 2-Byte-Befehle auf, vor denen 2∅3 (in hexadezimaler Darstellung CB) oder 237 (in hexadezimaler Darstellung ED) steht. Tabelle A3 listet die Indexregister-Befehle auf.

Es gibt viele Muster im Befehlssatz. Beispielsweise sind die Register fast immer folgendermaßen angeordnet: b, c, d, e, h, l, (hl), a wie etwa bei den Zahlen 64 bis 127 in der Gruppe der Ladebefehle aus 8-Bit-Register in Register. In ähnlicher Weise verhalten sich die Indexregister-Codes wie die hl-Codes, wobei vor ihnen 221 (in hexadezimaler Darstellung DD) steht, wenn man sich auf Indexregister ix bezieht, und 253 (in hexadezimaler Darstellung FD), wenn auf Indexregister iy Bezug genommen wird.

Einige Befehle sind mit einem oder mehreren der folgenden Symbole versehen

n eine ganze Zahl, die in ein Byte hineinpaßt, zwischen ∅ und 255 einschließlich

d eine 1-Byte-Distanz zwischen –128 und 127 (bei Indexregisterbefehlen und Sprungbefehlen); man erhält negative Werte für d durch Subtraktion des positiven Wertes von 256

nn eine ganze Zahl, die in zwei Bytes hineinpaßt, zwischen ∅ und 65535 einschließlich; das höherwertige Byte liegt an zweiter Stelle, z. B. 16384 (= ∅ + 256*64) wird als ∅,64 gespeichert

Diese Symbole werden immer in das Byte oder die Bytes geschrieben, die auf den entsprechenden Befehl folgen. Eine Ausnahme bilden die 3-Byte-Indexregister-Befehle (in Tabelle A3 die Spalten 5 und 6). Hier stehen sie zwischen dem zweiten und dritten Byte. Beispiele findet man in Tabelle A1.

Tabelle A1

Einige Beispiele aus dem Z8∅A-Befehlssatz. Spalte eins bezieht sich auf die entsprechende Spalte in den Tabellen A2 und A3.

Tabellen- und Spaltenzahl	Allgemeine Form	spezielles Beispiel	Dezimaldarstellung			
A2,3	–	inc b	4			
A2,3	ld e,n	ld e,25	3∅	25		
A2,3	ld a, (nn)	ld a, (23296)	58	∅	91	
A2,4	–	res 2,d	2∅3	92		
A2,5	ld (nn),de	ld (2376∅),de	237	53	2∅8	92
A3,3	–	add ix,bc	221	9		
A3,3	ld (ix + d),n	ld (ix + 19),5	221	54	19	5
A3,4	–	add iy,bc	253	9		
A3,4	ld (nn), iy	ld (2376∅)iy	253	34	2∅8	92
A3,5	rrc (ix + d)	rrc (ix + 5)	221	2∅3	5	14
A3,6	rrc (iy + d)	rrc (iy + 5)	253	2∅3	5	14

Tabelle A2
Z8∅A-Befehle ohne die Indexregister-Codes (siehe Tabelle A3).

Dezimal	Hex	Opcode	Nach 2∅3 (hex CB)	Nach 237 (hex ED)
∅	∅∅	nop	rlc b	
1	∅1	ld bc,nn	rlc c	
2	∅2	ld (bc),a	rlc d	
3	∅3	inc bc	rlc e	
4	∅4	inc b	rlc h	
5	∅5	dec b	rlc l	
6	∅6	ld b,n	rlc (hl)	
7	∅7	rlca	rlc a	
8	∅8	ex,af,af′	rrc b	
9	∅9	add hl,bc	rrc c	
1∅	∅A	ld a, (bc)	rrc d	
11	∅B	dec bc	rrc e	
12	∅C	inc c	rrc h	
13	∅D	dec c	rrc l	
14	∅E	ld c,n	rrc (hl)	
15	∅F	rrca	rrc a	
16	1∅	djnz d	rl b	
17	11	ld de,nn	rl c	
18	12	ld (de),a	rl d	
19	13	inc de	rl e	
2∅	14	inc d	rl h	
21	15	dec d	rl l	
22	16	ld d,n	rl (hl)	
23	17	rla	rl a	
24	18	jr d	rr b	
25	19	add hl,de	rrc	
26	1A	ld a, (de)	rr d	
27	1B	dec de	rr e	
28	1C	inc e	rr h	
29	1D	dec e	rr l	
3∅	1E	ld e,d	rr (hl)	
31	1F	rra	rr a	
32	2∅	jr nz,d	sla b	
33	21	ld hl,nn	sla c	
34	22	ld (nn),hl	sla d	
35	23	inc hl	sla e	
36	24	inc h	sla h	
37	25	dec h	sla l	
38	26	ld h,n	sla (hl)	
39	27	daa	sla a	
4∅	28	jr z,d	sra b	
41	29	add hl,hl	sra c	
42	2A	ld hl, (nn)	sra d	
43	2B	dec hl	sra e	
44	2C	inc l	sra h	
45	2D	dec l	sra l	

Dezimal	Hex	Opcode	Nach 2∅3 (hex CB)	Nach 237 (hex ED)
46	2E	ld l,n	sra (hl)	
47	2F	cpl	sra a	
48	3∅	jr nc,d		
49	31	ld sp,nn		
5∅	32	ld (nn),a		
51	33	inc sp		
52	34	inc (hl)		
53	35	dec (hl)		
54	36	ld (hl),n		
55	37	scf		
56	38	jr c,d	srl b	
57	39	add hl,sp	srl c	
58	3A	la d, (nn)	srl d	
59	3B	dec sp	srl e	
6∅	3C	inc a	srl h	
61	3D	dec a	srl l	
62	3E	ld a,n	srl (hl)	
63	3F	ccf	srl a	
64	4∅	ld b,b	bit ∅,b	in b, (c)
65	41	ld b,c	bit ∅,c	out (c),b
66	42	ld b,d	bit ∅,d	sbc hl,bc
67	43	ld b,e	bit ∅,e	ld (nn),bc
68	44	ld b,h	bit ∅,h	neg
69	45	ld b,l	bit ∅,l	retn
7∅	46	ld b, (hl)	bit ∅, (hl)	im ∅
71	47	ld b,a	bit ∅,a	ld i,a
72	48	ld c,b	bit 1,b	in c, (c)
73	49	ld c,c	bit 1,c	out (c),c
74	4A	ld c,d	bit 1,d	adc hl,bc
75	4B	ld c,e	bit 1,e	ld bc,(nn)
76	4C	ld c,h	bit 1,h	
77	4D	ld c,l	bit 1,l	reti
78	4E	ld c, (hl)	bit 1, (hl)	
79	4F	ld c,a	bit 1,a	ld r,a
8∅	5∅	ld d,b	bit 2,b	in d, (c)
81	51	ld d,c	bit 2,c	out (c),d
82	52	ld d,d	bit 2,d	sbc hl,de
83	53	ld d,e	bit 2,e	ld (nn),de
84	54	ld d,h	bit 2,h	
85	55	ld d,l	bit 2,l	
86	56	ld d, (hl)	bit 2, (hl)	im 1
87	57	ld d,a	bit 2,a	ld a,i
88	58	ld e,b	bit 3,b	in e,(c)
89	59	ld e,c	bit 3,c	out (c),e
9∅	5A	ld e,d	bit 3,d	adc hl,de
91	5B	ld e,e	bit 3,e	ld de,(nn)
92	5C	ld e,h	bit 3,h	
93	5D	ld e,l	bit 3,l	

Dezimal	Hex	Opcode	Nach 2Ø3 (hex CB)	Nach 237 (hex ED)
94	5E	ld e, (hl)	bit 3, (hl)	im 2
95	5F	ld e,a	bit 3,a	ld a,r
96	6Ø	ld h,b	bit 4,b	in h, (c)
97	61	ld h,c	bit 4,c	out (c),h
98	62	ld h,d	bit 4,d	sbc hl,hl
99	63	ld h,e	bit 4,e	ld (nn),hl
1ØØ	64	ld h,h	bit 4,h	
1Ø1	65	ld h,l	bit 4,l	
1Ø2	66	ld h, (hl)	bit 4, (hl)	
1Ø3	67	ld h,a	bit 4,a	rrd
1Ø4	68	ld l,b	bit 5,b	in l,(c)
1Ø5	69	ld l,c	bit 5,c	out (c),l
1Ø6	6A	ld l,d	bit 5,d	adc hl,hl
1Ø7	6B	ld l,e	bit 5,e	ld hl,(nn)
1Ø8	6C	ld l,h	bit 5,h	
1Ø9	6D	ld l,l	bit 5,l	
11Ø	6E	ld l, (hl)	bit 5, (hl)	
111	6F	ld l,a	bit 5,a	rld
112	7Ø	ld (hl),b	bit 6,b	in f,(c)
113	71	ld (hl),c	bit 6,c	
114	72	ld (hl),d	bit 6,d	sbc hl,sp
115	73	ld (hl),e	bit 6,e	ld (nn),sp
116	74	ld (hl),h	bit 6,h	
117	75	ld (hl),l	bit 6,l	
118	76	halt	bit 6, (hl)	
119	77	ld (hl),a	bit 6,a	
12Ø	78	ld a,b	bit 7,b	in a,(c)
121	79	ld a,c	bit 7,c	out (c),a
122	7A	ld a,d	bit 7,d	adc hl,sp
123	7B	ld a,e	bit 7,e	ld sp, (nn)
124	7C	ld a,h	bit 7,h	
125	7D	ld a,l	bit 7,l	
126	7E	ld a, (hl)	bit 7, (hl)	
127	7F	ld a,a	bit 7,a	
128	8Ø	add a,b	res Ø,b	
129	81	add a,c	res Ø,c	
13Ø	82	add a,d	res Ø,d	
131	83	add a,e	res Ø,e	
132	84	add a,h	res Ø,h	
133	85	add a,l	res Ø,l	
134	86	add a, (hl)	res Ø, (hl)	
135	87	add a,a	res Ø,a	
136	88	adc a,b	res 1,b	
137	89	adc a,c	res 1,c	
138	8A	adc a,d	res 1,d	
139	8B	adc a,e	res 1,e	
14Ø	8C	adc a,h	res 1,h	
141	8D	adc a,l	res 1,l	

Dezimal	Hex	Opcode	Nach 2Ø3 (hex CB)	Nach 237 (hex ED)
142	8E	adc a, (hl)	res 1, (hl)	
143	8F	adc a,a	res 1,a	
144	9Ø	sub b	res 2,b	
145	91	sub c	res 2,c	
146	92	sub d	res 2,d	
147	93	sub e	res 2,e	
148	94	sub h	res 2,h	
149	95	sub l	res 2,l	
15Ø	96	sub (hl)	res 2, (hl)	
151	97	sub a	res 2,a	
152	98	sbc a,b	res 3,b	
153	99	sbc a,c	res 3,c	
154	9A	sbc a,d	res 3,d	
155	9B	sbc a,e	res 3,e	
156	9C	sbc a,h	res 3,h	
157	9D	sbc a,l	res 3,l	
158	9E	sbc a, (hl)	res 3, (hl)	
159	9F	sbc a,a	res 3,a	
16Ø	AØ	and b	res 4,b	ldi
161	A1	and c	res 4,c	cpi
162	A2	and d	res 4,d	ini
163	A3	and e	res 4,e	outi
164	A4	and h	res 4,h	
165	A5	and l	res 4,l	
166	A6	and (hl)	res 4, (hl)	
167	A7	and a	res 4,a	
168	A8	xor b	res 5,b	ldd
169	A9	xor c	res 5,c	cpd
17Ø	AA	xor d	res 5,d	ind
171	AB	xor e	res 5,e	outd
172	AC	xor h	res 5,h	
173	AD	xor l	res 5,l	
174	AE	xor (hl)	res 5, (hl)	
175	AF	xor a	res 5,a	
176	BØ	or b	res 6,b	ldir
177	B1	or c	res 6,c	cpir
178	B2	or d	res 6,d	inir
179	B3	or e	res 6,e	otir
18Ø	B4	or h	res 6,h	
181	B5	or l	res 6,l	
182	B6	or (hl)	res 6, (hl)	
183	B7	or a	res 6,a	
184	B8	cp b	res 7,b	lddr
185	B9	cp c	res 7,c	cpdr
186	BA	cp d	res 7,d	indr
187	BB	cp e	res 7,e	otdr
188	BC	cp h	res 7,h	
189	BD	cp l	res 7,l	

Dezimal	Hex	Opcode	Nach 203 (hex CB)	Nach 237 (hex ED)
190	BE	cp (hl)	res 7, (hl)	
191	BF	cp a	res 7,a	
192	C0	ret nz	set 0,b	
193	C1	pop bc	set 0,c	
194	C2	jp nz, nn	set 0,d	
195	C3	jp nn	set 0,e	
196	C4	call nz,nn	set 0,h	
197	C5	push bc	set 0,l	
198	C6	add a,n	set 0, (hl)	
199	C7	rst 0	set 0,a	
200	C8	ret z	set 1,b	
201	C9	ret	set 1,c	
202	CA	jp z,nn	set 1,d	
203	CB	–	set 1,e	
204	CC	call z,nn	set 1,h	
205	CD	call nn	set 1,l	
206	CE	adc a,n	set 1, (hl)	
207	CF	rst 8	set 1,a	
208	D0	ret nc	set 2,b	
209	D1	pop de	set 2,c	
210	D2	jp nc,nn	set 2,d	
211	D3	out (n),a	set 2,e	
212	D4	call nc,nn	set 2,h	
213	D5	push de	set 2,l	
214	D6	sub,n	set 2, (hl)	
215	D7	rst 16	set 2,a	
216	D8	ret c	set 3,b	
217	D9	exx	set 3,c	
218	DA	jp c,nn	set 3,d	
219	DB	in a,(n)	set 3,e	
220	DC	call c,nn	set 3,h	
221	DD	–	set 3,l	
222	DE	sbc a,n	set 3, (hl)	
223	DF	rst 24	set 3,a	
224	E0	ret po	set 4,b	
225	E1	pop hl	set 4,c	
226	E2	jp po,nn	set 4,d	
227	E3	ex (sp),hl	set 4,e	
228	E4	call po,nn	set 4,h	
229	E5	push hl	set 4,l	
230	E6	and n	set 4, (hl)	
231	E7	rst 32	set 4,a	
232	E8	ret pe	set 5,b	
233	E9	jp (hl)	set 5,c	
234	EA	jp pe,nn	set 5,d	
235	EB	ex de,hl	set 5,e	
236	EC	call pe,nn	set 5,h	
237	ED	–	set 5,l	

Dezimal	Hex	Opcode	Nach 2Ø3 (hex CB)	Nach 237 (hex ED)
238	EE	xor n	set 5, (hl)	
239	EF	rst 4Ø	set 5,a	
24Ø	FØ	ret p	set 6,b	
241	F1	pop af	set 6,c	
242	F2	jp p,nn	set 6,d	
243	F3	di	set 6,e	
244	F4	call p,nn	set 6,h	
245	F5	push af	set 6,l	
246	F6	or n	set 6, (hl)	
247	F7	rst 48	set 6,a	
248	F8	ret m	set 7,b	
249	F9	ld sp,hl	set 7,c	
25Ø	FA	jp m,nn	set 7,d	
251	FB	ei	set 7,e	
252	FC	call m,nn	set 7,h	
253	FD	–	set 7,l	
254	FE	cp n	set 7, (hl)	
255	FF	rst 56	set 7,a	

Tabelle A3
Indexregister-Codes. Die Spalten 3 und 5 beziehen sich auf das Register ix. Sie Spalten 4 und 6 beziehen sich auf das Register iy. Alle Befehle in dieser Tabelle verhalten sich wie die Befehle für das Registerpaar hl in Tabelle A2.

Dezi-mal	Hex	Nach 221 (hex DD)	Nach 253 (hex FD)	Nach 221, 2Ø3 (hex DD, CB)	Nach 253, 2Ø3 (hex FD, CB)
6	Ø6			rlc (ix+d)	rlc (iy+d)
9	Ø9	add ix, bc	add iy,bc		
14	OE			rrc (ix+d)	rrc (iy+d)
22	16			rl (ix+d)	rl (iy+d)
25	19	add ix,de	add iy,de		
3Ø	IE			rr (ix+d)	rr (iy+d)
33	21	ld ix,nn	ld iy,nn		
34	22	ld (nn),ix	ld (nn),iy		
35	23	inc ix	inc iy		
38	26			sla (ix+d)	sla (iy+d)
41	29	add ix,ix	add iy,iy		
42	2A	ld ix,(nn)	ld iy,(nn)		
43	2B	dec ix	dec iy		
46	2E			sra (ix+d)	sra (iy+d)
52	34	inc (ix+d)			
53	35	dec (ix+d)	dec (iy+d)		
54	36	ld (ix+d),n	ld (iy+d),n		

Dezi-mal	Hex	Nach 221 (hex DD)	Nach 253 (hex FD)	Nach 221, 2∅3 (hex DD, CB)	Nach 253, 2∅3 (hex FD, CB)
57	39	add ix,sp	add iy,sp		
62	3E			srl (ix+d)	srl (iy+d)
7∅	46	ld b, (ix+d)	ld b, (iy+d)	bit ∅,(iy+d)	bit ∅,(iy+d)
78	4E	ld c,(ix+d)	ld c,(Iy+d)	bit 1,(ix+d)	bit 1,(iy+d)
86	56	ld d,(ix+d)	ld d,(iy+d)	bit 2,(ix+d)	bit 2,(iy+d)
94	5E	ld e,(ix+d)	ld e,(iy+d)	bit 3,(ix+d)	bit 3,(iy+d)
1∅2	66	ld h,(ix+d)	ld h,(iy+d)	bit 4,(ix+d)	bit 4,(iy+d)
11∅	6E	ld l,(ix+d)	ld l,(iy+d)	bit 5,(ix+d)	bit 5,(iy+d)
112	7∅	ld (ix+d),b	ld (iy+d),b		
113	71	ld (ix+d),c	ld (iy+d),c		
114	72	ld (ix+d),d	ld (iy+d),d		
115	73	ld (ix+d),e	ld (iy+d),e		
116	74	ld (ix+d),h	ld (iy+d),h		
117	75	ld (ix+d),l	ld (iy+d),l		
118	76			bit 6, (ix+d)	bit 6, (iy+d)
119	77	ld (ix+d),a	ld (iy+d),a		
126	7E	ld a,(ix+d)	ld a,(iy+d)	bit 7, (ix+d)	bit 7,(iy+d)
134	86	add a,(ix+d)	add a,(iy+d)	res ∅,(ix+d)	res ∅,(iy+d)
142	8E	adc a,(ix+d)	adc a,(iy+d)	res 1,(ix+d)	res 2,(iy+d)
15∅	96	sub (ix+d)	sub (iy+d)	res 2,(ix+d)	res 2,(iy+d)
158	9E	sbc a,(ix+d)	sbc a,(iy+d)	res 3,(ix+d)	res 3,(iy+d)
166	A6	and (ix+d)	and (iy+d)	res 4,(ix+d)	res 4,(iy+d)
174	AE	xor (ix+d)	xor (iy+d)	res 5,(ix+d)	res 5,(iy+d)
182	B6	or (ix+d)	or (iy+d)	res 6,(ix+d)	res 6,(iy+d)
19∅	BE	cp (ix+d)	cp (iy+d)	res 7,(ix+d)	res 7,(iy+d)
198	C6			set ∅,(ix+d)	set ∅,(iy+d)
2∅6	CE			set 1,(ix+d)	set 1,(iy+d)
214	D6			set 2,(ix+d)	set 2,(iy+d)
222	DE			set 3,(ix+d)	set 3,(iy+d)
225	E1	pop ix	pop iy		
227	E3	ex (sp),ix	ex (sp),iy		
229	E5	push ix	push iy		
23∅	E6			set 4,(ix+d)	set 4,(iy+d)
233	E9	jp (ix)	jp (iy)		
238	EE			set 5,(ix+d)	set 5,(iy+d)
246	F6			set 6,(ix+d)	set 6,(iy+d)
249	F9	ld sp,ix	ld sp,iy		
254	FE			set 7,(ix+d)	set 7,(iy+d)

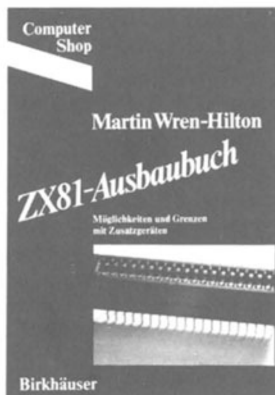
Birkhäuser Computer Shop

Ian Stewart
Robin Jones

Weitere Kniffe und Programme mit dem ZX Spectrum

1983, 160 Seiten, Broschur

Dieser Folgeband zu «Sinclair ZX Spectrum – Programmieren leicht gemacht» hilft Ihnen dabei, noch mehr aus Ihrem ZX Spectrum herauszuholen. Das Buch präsentiert eine ganz neue Auswahl von Programmen und Anwendungen, die nur einen 16-K-RAM-Speicher benötigen, also mit beiden Versionen des Spectrum gefahren werden können.



Martin Wren-Hilton

ZX 81 Ausbaubuch

Möglichkeiten und Grenzen mit Zusatzgeräten

1983, 100 Seiten, Broschur

O. K., Sie haben Ihren ZX 81 bekommen und gelernt, wie man ihn programmiert. Nun wollen Sie aber mal etwas wirklich Nützliches mit ihm anstellen! Dieses Buch bietet einen Überblick über die Hardware, die Sie zum ZX 81 kaufen können, und zusätzlich einige sehr brauchbare Programme, um diese Hardware dann auch zum Laufen zu bringen.

Weitere Bände in Vorbereitung:

Owen Bishop

Einfache Peripheriegeräte im Selbstbau

ca. 170 Seiten, Broschur

Ian Sinclair

Programmieren mit dem Commodore 64

ca. 140 Seiten, Broschur

Änderungen vorbehalten.