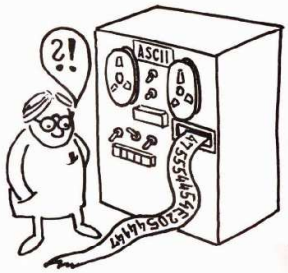
Trotz allen Komforts höherer Programmiersprachen lassen sich viele Software-Probleme nur in der prozessor-spezifischen Maschinensprache lösen, besonders, wenn es um geringen Speicherbedarf oder um hohe Verarbeitungsgeschwindigkeit geht. Am Beispiel des 6502 wird hier gezeigt, wie man zeit-, nerven- und speichersparend programmiert. Dabei wird auch auf Unterschiede zwischen den heute gängigen Mikroprozessoren eingegangen.

Herwig Feichtinger

Bits und Bytes

Eine kleine Einführung in die Maschinenprogrammierung



Bits, Bytes, Worte, Zahlen

Grundsätzlich besteht ein Mikrocomputer aus der CPU, dem Mikroprozessor also, einem Programmspeicher, der meist als ROM (read-only memory) ausgelegt ist, einem Datenspeicher (RAM, random access memory) und Bausteinen zur Ein- und Ausgabe von Informationen.

Daten und Programme in RAM und ROM werden vom Mikroprozessor „wortweise" über den Datenbus transferiert. Bei 8-bit-Prozessoren (SCMP, 8080, 8085, 6800, 6502, Z-80 usw.) ist ein solches Wort acht bit lang, bei 16-bit-Prozessoren (TMS 9900, 8086, 68000, Z-8000) ist es 16 bit lang. Ein Bit ist die kleinstmögliche Informationseinheit, nämlich eine Ja-Nein-Entscheidung. Ein Byte ist eine zusammenhängende Folge von 8 Bits, dies gilt auch im Sprachgebrauch der 16-bit-Prozessoren. Ordnet man den acht Bits die binäre Wertigkeit von 2^0 bis 2^7 zu, so kann ein Byte die dezimalen Werte 0...255 annehmen.

Bei 8-bit-Prozessoren ist der Datenbus 8 bit breit, d.h. er besteht aus acht Leitungen, die die CPU mit den Speichern verbinden. Bei 16-bit-Prozessoren ist er meist 16 bit breit; manchmal spaltet man das 16-bit-Datenwort aber auch in zwei Bytes auf, die nacheinander auf den Datenbus gegeben werden. Dadurch geht zwar der Geschwindigkeitsvorteil der 16-bit-Prozessoren verloren, man spart aber acht Pins am CPU-Gehäuse (Beispiel: TMS 9980). Den gleichen Effekt erzielt man durch Multiplexen des Datenbus mit dem Adreßbus (8086).

Hexadezimale Darstellung

Wenn Sie sich Mikrocomputer-Programme oder Speicherinhalte auf dem Display eines kleinen Systems oder auch in Zeitschriften und Büchern ansehen, so werden Sie feststellen, daß die einzelnen Bytes nicht als Folge von acht Nullen und Einsen, also als Bitfolge, wiedergegeben werden, sondern als Ziffern und Buchstaben, nämlich hexadezimal. Dabei steht eine „Ziffer" (0...F) für vier Bits und repräsentiert eine Dezimalzahl zwischen 0 und 15. Hier eine kleine Tabelle:

Hex	Dezimal	Binär
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Ein Byte oder acht Bits können also durch zwei Hexadezimal-Ziffern dargestellt werden. Da dies dezimal nicht geht, verwenden die meisten Mikrocomputer auch eine hexadezimale Anzeige. Vorsicht: Die Bitfolge 1011 0111 ist zwar hexadezimal B7, aber nicht dezimal 117! Wir wollen uns hier aber nicht weiter um Zahlenumrechnungen kümmern – diese Problematik wurde in der FUNKSCHAU ja schon mehrfach, z. T. auch mit Taschenrechner-Umwandlungsprogrammen, abgehandelt.

Einige wenige Mikrocomputer, z. B. der H-8 von Heathkit, arbeiten nicht mit hexadezimaler, sondern oktaler Ein- und Ausgabe. Dabei repräsentiert eine Ziffer von 0...7 drei Bits. Pro Byte werden dann drei Ziffern benötigt. Bei den moderneren Mikrocomputern konnte sich das Verfahren der oktalen Darstellung nicht durchsetzen.

ASCII

Will man in einem Mikrocomputer nicht nur Zahlen, sondern auch Buchstaben, z. B. Text oder Befehle einer höheren Programmiersprache speichern, so wird jeweils ein Byte verwendet, um ein Zeichen zu repräsentieren. Dies geschieht nach dem sog. ISO-7-bit-Code bzw. ASCII (American Standard Code for Information Interchange). Mit 8 bit lassen sich bis zu 256 unterschiedliche Zeichen darstellen, wovon „nur" 128 für Schriftsymbole und der Rest entweder gar nicht oder für Grafiken genutzt werden.

Die acht Bits des ISO-Codes lassen sich natürlich wieder hexadezimal darstellen. Und so lassen sich die Schriftsymbole der Ziffern 0...9 z. B. einfach dadurch in das hexadezimale ASCII-Äquivalent umrechnen, indem man (hex) 30 addiert. So entspricht der Ziffer 5 z. B. der Hex-Code 35. Die Buchstaben sind alphabetisch mit A beginnend ab hex 41 durchnumeriert. Eine vollständige Tabelle der ASCII-Zeichen

findet sich in FUNKSCHAU 1979, Heft 7, Seite 397. Allerdings braucht man sich darum nur zu kümmern, wenn der Mikrocomputer über alphanumerische Ein- und Ausgabemöglichkeiten verfügt.

Zur Darstellung von Schriftzeichen gibt es auch andere Codes, die in der Mikrocomputer-Technik aber nicht gebräuchlich sind, z. B. der 5-bit-Baudot-Code, der im Fernschreibnetz üblich ist, oder der bei manchen elektrischen Schreibmaschinen verwendete EBCDIC-Code. Eine eventuell notwendige Codeumsetzung ist mit einem Mikrocomputer per Software leicht möglich. So wurde etwa ein Programm zum Umsetzen von ASCII in Baudot in FUNKSCHAU 1979, Heft 1, für den 6502 beschrieben.

Bevor wir uns den Innereien eines Mikroprozessors zuwenden, noch ein Wort über den Ausdruck „Maschinensprache". Er stammt aus einer Zeit, als die Computer noch aus Tausenden einzelner Transistoren und einiger Mechanik wie Relais und Trommelspeicher bestanden. „Maschine" war für diese lärm- und hitzeerzeugenden Stromfresser wohl durchaus das richtige Wort. Ein Maschinenprogramm besteht lediglich aus einer Folge von Bytes, der man zunächst nicht viel mehr entnehmen kann als einer mit japanischen Schriftzeichen gedruckten Bedienungsanleitung.

Register: Speicher in der CPU

Auch im Mikroprozessor selbst gibt es Speicherzellen, die man als „Register" bezeichnet. An dieser Stelle müssen wir, um nicht in graue Theorie zu verfallen, konkret werden und uns die Register eines realen, handelsüblichen Prozessors ansehen. Wir wählen die CPU 6502, weil sie recht verbreitet und in einer Reihe preiswerter Einplatinen-Computer zu finden ist, wie KIM-1, SYM-1, AIM-65 und PC-100. Dem Leser mag auffallen, daß solche hochwertigen Computer wie der PET 2001 hier nicht genannt sind. Solche Geräte sind für das Arbeiten in BASIC konstruiert; das Programmieren in der Maschinensprache der CPU ist bei ihnen sehr umständlich. Trotzdem sind dafür oft sog. Monitor-Programme lieferbar, z. B. TIM für den PET.

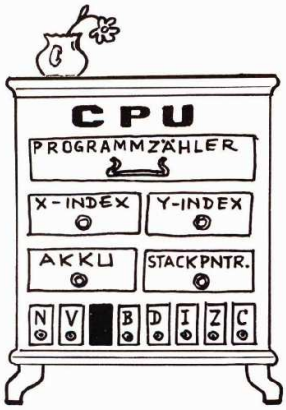
Unsere Wahl des 6502 bedeutet keinesfalls, daß die Besitzer anderer Prozessoren alles folgende überblättern müssen; ganz im Gegenteil werden sie oft „ihre" Prozessoren im Vergleich mit dem 6502 erwähnt finden, wobei dieser manchmal auch Federn lassen muß.

Der Programmzähler

Doch zurück zu den Registern! Jeder Prozessor besitzt zunächst einmal einen Programmzähler, der ihm sagt, wo der nächste Befehl im Speicher zu finden ist. Der Programmzähler ist ebenso breit wie der Adreßbus der CPU, bei 8-bit-Prozessoren meist 16 bit (damit lassen sich 2^{16} = 65536 Bytes adressieren) und bei 16-bit-Prozessoren im Idealfall 32 bit (z. B. 68000 oder 8086). Manche 16-bit-CPUs besitzen nur einen 14- oder 15-bit-Adreßbus bzw. -Programmzähler (TMS 9980/9900), was dazu führt, daß der adressierbare Speicherbereich sogar kleiner ist als der ihrer 8-bit-Kollegen.

Der Akku

Das wichtigste Register in der CPU ist der Akku, deshalb besitzt der 6800 gleich zwei davon. In der Regel steht aber nur ein Akku zur Verfügung, und nur in ihm kann man arithmetische Operationen wie Addieren und Subtrahieren ausführen. Der Akku ist bei 8-bit-CPUs 8 bit lang.



Der 16-bit-Mikroprozessor TMS 9900 besitzt wie der 68000 keinen speziellen Akku. Von den meisten anderen CPUs unterscheidet er sich darin, daß sich arithmetische Operationen gleichberechtigt in seinen 16-bit-Registern ausführen lassen, die übrigens nicht innerhalb der CPU liegen, sondern im RAM-Bereich.

Während die meisten Prozessoren mit dem 6502 noch das Vorhandensein eines Akkus gemeinsam haben, hört diese Gemeinsamkeit schon bei den Indexregistern X und Y auf. Der 6800 hat z. B. nur ein einziges Indexregister, jedoch mit 16 bit Länge; andere Prozessoren kennen diese Einrichtung überhaupt nicht.

X- und Y-Register

Der Mikroprozessor 6502 enthält zwei weitere Register, in die vom Programm Daten hineingeschrieben werden können, nämlich die beiden 8-bit-Indexregister X und Y. „Index" bedeutet hier „Adressenversatz" – eine nützliche Einrichtung, auf die wir bei der Besprechung der Adressierungsarten noch kommen werden. X und Y eignen sich zwar nicht für arithmetische Operationen, lassen sich aber z. B. zum Aufwärts- und Abwärtszählen einsetzen.

Das Statusregister

Ein weiteres CPU-Register wird von Akku, X- und Y-Register bei der Ausführung bestimmter Befehle beeinflußt: das Statusregister.

Es enthält ebenfalls acht Bits, von denen jedes einen bestimmten Zustand (Status) signalisiert – sozusagen mit einer Flagge, die einzelnen Bits heißen nämlich „Flags". Sie zeigen an, ob das Ergebnis der letzten Operation Null (Zero) ist, ob ein Übertrag (Carry) z. B. bei einer Addition auftrat, ob das höchstwertige Bit einer Zahl bei der letzten Operation 1 war (Negativ-Flag) und ob ein Software-Interrupt (Break-Befehl) auftrat. Dann gibt es noch etwas beim 6502, was die Besitzer anderer Prozessoren manchmal vor Neid erblassen läßt: das Dezimal-Flag. Ist es 1, also gesetzt, so addiert und subtrahiert der Prozessor dezimal; ist es rückgesetzt, erfolgen diese arithmetischen Operationen binär bzw. hexadezimal. Mit dem Dezimal-Flag kann man also die CPU zwischen zwei Zahlensystemen umschalten.

Dann existiert noch ein Status-Bit, mit dem man externe Interrupts (Programm-Unterbrechungen durch Hardware-Impulse, z. B. Datenanforderung eines Druckers) verhindern kann. Dieses Flag heißt deshalb „Interrupt Disable".

Das sog. Overflow-Statusbit (V) wird nur bei einigen wenigen Operationen beeinflußt und braucht uns hier noch nicht zu interessieren. Ein letztes Bit im Statusregister ist völlig unbenutzt und liegt in der CPU hardwaremäßig auf log. 1.

Ein weiteres CPU-Register wurde noch nicht erwähnt: der Stackpointer. Auch er ist ein 8-bit-Register (bei anderen Prozessoren z. T. 16 bit) und dient zur Zwischenspeicherung von Adressen und Daten. Doch hierzu später; um dieses Register brauchen wir uns erst bei der Behandlung von Unterprogrammen und Interrupts wieder zu kümmern.

Ein Befehl: 1...3 Bytes

Je nach Befehl holt sich der Prozessor 1...3 Bytes aus dem Programmspeicher und decodiert sie. (Es gibt auch 8-bit-Prozessoren, die nur 1...2-Byte-Befehle kennen, wie der SCMP, oder bei denen sogar vier Bytes pro Befehl denkbar sind, z. B. der Typ Z-80.)

Das erste Byte des Befehls ist der sog. Operationscode. Die Operationscodes des 6502 sind in FUNKSCHAU 1979, Heft 11, S. 657, abgedruckt. Sie stellen den eigentlichen Befehl dar.

Manche Befehle führen eine Operation aus, für die kein Argument benötigt wird; z. B. löst der Befehl SED (Set Decimal Flag, hex F8) eine CPU-interne Operation aus, für die keine Daten benötigt werden. Andere Befehle benötigen zur Ausführung zusätzliche Daten. So wäre etwa der Befehl LDA (Load Accu) sinnlos, wenn man nicht angeben würde, womit der Akku geladen werden soll. Will man den Akku mit dem Inhalt einer beliebigen Speicherzelle laden, so muß dem Befehl LDA eine 16-bit-Adresse folgen, die als vierstellige Hexadezimal-Zahl geschrieben werden kann, was zwei Bytes entspricht. Der Befehl setzt sich also hier aus drei Bytes zusammen – einem Byte für den Operationscode und zwei für das Argument.

Eine weitere Möglichkeit ist, daß der Akku direkt mit irgendwelchen Daten geladen werden soll, z. B. mit 56. Der Befehl lautet hier A9 56; der Operationscode ist A9, und 56 das Argument. Hier werden nur zwei Byte benötigt.

Die CPU erkennt am Operationscode bereits, wie viele Bytes noch zu dem Befehl gehören, und stellt den Programmzähler während der Befehlsausführung entsprechend weiter.

Sieht man sich im Speicher eines Mikrocomputers um, so erkennt man an den hexadezimalen Operationscodes natürlich nicht gleich, wie viele Datenbytes nach ihnen folgen bzw. wo der nächste Befehl beginnt. Hier leistet ein „Disassembler" gute Dienste: Er übersetzt nicht nur die Hex-Codes in die mnemonische Form (z. B. A9 in LDA), sondern erkennt auch die richtige Befehlslänge. Disassembler sind selbst Programme; für den 6502 wurde ein solches Programm in FUNKSCHAU 1978, Heft 21 veröffentlicht. Der Mikrocomputer AIM-65 hat bereits einen Disassembler im ROM „eingebaut". Voraussetzung für die Anwendung eines Disassemblers ist allerdings grundsätzlich eine alphanumerische Ausgabemöglichkeit.

Befehle und Daten

Eine naheliegende Frage ist nun: Da sowohl die Operationscodes als auch die zu verarbeitenden Daten, die ihnen folgen, zunächst einmal nur binäre Zahlen sind, woher weiß dann die CPU, was Befehle und was Daten sind?

Die Antwort ist sehr einfach: Sie weiß es nämlich nicht! Der Programmierer hat dafür zu sorgen, daß er das Programm an einer Adresse startet, an der auch tatsächlich ein Operationscode steht. Dann ist alles gerettet: Ab sofort weiß ja die CPU, wie viele Daten-Bytes dem Operationscode folgen, d.h. wo sie den nächsten Befehlscode findet.

Kritisch wird es erst bei Sprungbefehlen. Wenn der Programmierer nicht aufgepaßt hat, springt das Programm u.U. einmal nicht auf einen Operationscode, sondern auf die ihm folgenden Daten und interpretiert diese als Befehl. Dies kann katastrophale Folgen haben: Steht das Programm im RAM, so kann es sich u.U. selbst zerstören, indem undefiniert Daten an irgendwelche Adressen gespeichert werden. In solchen Fällen muß man oft das gesamte Programm neu eingeben.

Ein erstes Programm

Wir könnten nun schon versuchen, ein einfaches Programm selbst zu schreiben. Hier gehen wir aber einmal den umgekehrten Weg: Wir analysieren einen vorhandenen Speicherinhalt, der folgendermaßen aussieht:

```
0200 A9 05 18 69 07 8D 10 02
0208 4C 4F 1C (beim KIM-1)
0208 4C BF E0 (beim AIM-65)
```

Selbstverständlich stehen hier nicht alle Bytes an nur zwei Adressen, sondern A9 steht bei 0200, 05 bei 0201 usw. An der Adresse 0208 steht hier ein Rücksprungbefehl zum Monitor-Programm des jeweils verwendeten Mikrocomputers, der mit dem eigentlichen Programm nichts zu tun hat, sondern lediglich ermöglicht, daß nach dem Programmablauf wieder die Eingabe und Anzeige von Befehlen und Daten über Tastatur und Display möglich sind.

Die hier aufgelisteten Bytes können bei den verschiedenen Mikroprozessoren (6800, Z-80 usw.) eine ganz unterschiedliche Bedeutung haben. Nicht zuletzt deshalb eignen sich diese hexadezimalen Codes für das Dokumentieren von Programmen nicht besonders gut – der Besitzer eines anderen Prozessortyps kann mit ihnen nichts anfangen.

Zum Analysieren des Speicherinhaltes können wir uns entweder einer Operationscode-Tabelle bedienen, wir können aber auch einen Disassembler benützen, wie er bereits im AIM-65-Monitorprogramm vorhanden ist. Er liefert uns folgenden Ausdruck:

```
0200 A9    LDA  # 05
0202 18    CLC
0203 69    ADC  # 07
0205 8D    STA  0210
0208 4C    JMP  E0BF
```

Ganz links steht dabei jeweils die Adresse des Operationscodes, dann folgt der Operationscode und der Befehl in mnemonischer Form (drei Buchstaben) nebst dem Argument, sofern eines vorhanden ist.

Andere Disassembler lassen entweder den hexadezimalen Operationscode weg, oder aber sie drucken die Befehle etwa in folgender Form:

```
0200 A9 05   LDA # 05
```

Was tut nun dieser erste Befehl? LDA bedeutet „Load Accu", und das Doppelkreuz vor dem Argument besagt, daß der Akku nicht mit dem Inhalt einer Adresse außerhalb des Programms, sondern direkt mit dem Wert des dem Operationscode folgenden Byte geladen werden soll. Dieses direkte Laden wird in der Operationscode-Tabelle mit der „Adressierungsart Immediate" bezeichnet.

Der nun folgende Befehl CLC löscht das Übertrags-Flag im Status-Register. Dies ist notwendig, weil der nachfolgende Additionsbefehl, der zum Akkuinhalt 7 addieren soll, das Übertrags-Flag (Carry) mit einbezieht, um Additionen mehrstelliger Zahlen zu erlauben. Der Zustand des Carry-Flag ist vor der Ausführung des Programms nicht definiert; es kann zufällig irgendeinen Zustand angenommen haben.

Nach dem ADC-Befehl (Adressierungsart wieder „Immediate") steht im Akku die hexadezimale Summe von 5 und 7. (Das Monitorprogramm des Mikrocomputers hat vor der Programmausführung automatisch dafür gesorgt, daß der Prozessor im Hexadezimal-Modus arbeitet, d.h. das Dezimal-Flag gelöscht ist.)

Da wir nicht direkt in den Akku hineinsehen können, muß das Ergebnis an irgendeine Speicherzelle transferiert werden, an der wir es später prüfen können. Dafür sorgt der Befehl STA 0210; er speichert den Akkuinhalt an die Adresse 0210. Bitte beachten Sie: Das hexadezimale Programm ist so aufgebaut, daß nach dem Operationscode 8D zunächst das niederwertige, dann das höherwertige Byte der Zieladresse folgt; dies ist bei den meisten Mikroprozessoren üblich, nur der 6800 von Motorola macht hier eine Ausnahme.

Der letzte Befehl, JMP, ist systemabhängig und führt zu einem Sprung in das Monitorprogramm des Mikrocomputers, beim AIM-65 an die Adresse E0BF. Ebenso wie beim vorhergegangenen STA-Befehl ist die Adressierungsart „absolut": Es wird nach dem Operationscode eine 16-bit-Adresse als Argument genannt.

Wenn wir unseren Mikrocomputer mit dem Programm laden und es an der Adresse 0200 starten, muß danach an der Adresse 0210 das Additionsergebnis 0C (dezimal 12) stehen.

Zero-Page-Adressierung

Was ist eine „Page"?

Wie schon erwähnt, wird der adressierbare Speicherbereich eines Mikrocomputers von 16 Bits charakterisiert. Diese 16 Bits lassen sich mit vier hexadezimalen „Ziffern" (0...F) darstellen. Der gesamte Adressenbereich reicht also von 0000 bis FFFF.

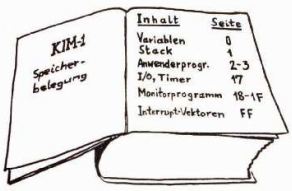
Eine Adresse kann als Folge zweier 8-bit-Bytes betrachtet werden; dies geschieht ja auch bei der gerade erwähnten „absoluten" Adressierung. Während sich mit zwei Byte der gesamte Adressenbereich von 65 536 Byte überstreichen läßt, kann man mit dem niederwertigen der beiden Adreßbytes nur $2^8 = 256$ Byte adressieren. Und einen solchen Bereich von 256 Byte nennt man auch „Page", zu deutsch „Seite". Der gesamte Adressenbereich eines Mikrocomputers kann also durch 256 Seiten mit je 256 Byte dargestellt werden.

Der untere Adressenbereich (Page 0, 1, 2...) ist in 6502-Systemen meist mit RAM, der obere mit dem das Monitorprogramm enthaltenden ROM belegt. Um Hardware zu sparen, werden in vielen Einplatinen-Computern nicht alle 16 Adreßleitungen decodiert, so daß sich manche Adressen „spiegeln"; so findet man etwa beim KIM-1 in Page 22 die gleichen Daten wie in Page 2, d.h. die Daten in den Zellen 0200...02FF sind scheinbar identisch mit denen im Bereich 2200...22FF.

Die nullte Seite

Diejenige „Page", bei der das höherwertige Adreßbyte 00 ist, hat beim 6502 eine besondere Bedeutung. Diese „Zero Page" – sie ist eine eigene Adressierungsart in der Operationscode-Tabelle – erlaubt Speicheroperationen mit nur zwei Bytes pro Befehl, was ideal für die Variablen eines Programms ist.

Ein Beispiel: Bei unserem Additionsprogramm wurde das Ergebnis im Akku mit einem Drei-Byte-Befehl (8D 10 02) an die Adresse 0210 gespeichert. Wenn wir es stattdessen bei 0010 abspeichern wollen, könnten wir natürlich den Befehl auf 8D 10 00 ändern; wir können aber auch die Adressierungsart „Zero Page" verwenden. Der Operationscode



ist dann laut Tabelle 85, und der gesamte Befehl hat nur noch zwei Bytes: 85 10. Disassembliert sähe das z. B. so aus:

```
0205 85 10  STA 10
```

Die nächste freie Adresse ist 0207, hier könnte jetzt – also ein Byte früher als vorher – der JMP-Befehl stehen. Wollen wir ihn nicht verschieben, so müssen wir den nun freien Platz auffüllen, nämlich mit dem Befehl „No Operation":

```
0207 EA NOP
```

Er besteht aus nur einem Byte, da er kein Argument benötigt, und verändert nichts in der CPU außer dem Programmzähler; d.h. das Statusregister bleibt unbeeinflußt.

Noch einmal: Das Statusregister

Sehen wir uns doch einmal an, was beim Ablauf des einfachen Additions-Programmes mit dem Statusregister in der CPU geschieht. Hier eine zusammenfassende Aufstellung:

0200
Das Null-Flag (Zero-Flag) wird auf Null rückgesetzt, weil der Akku mit einem Wert ungleich Null geladen wird.

0202
Das Carry-Flag wird gelöscht.

0203
Es ändert sich nichts: Da kein Übertrag bei der Addition auftritt, bleibt das Carry-Flag gelöscht. Auch das Zero-Flag bleibt Null, da das Ergebnis ungleich Null ist.

0205
Operationen, die den Inhalt eines CPU-Registers in den RAM-Speicher transferieren, ändern grundsätzlich kein Status-Bit; es ändert sich also wieder nichts.

Welches Bit im Statusregister bei welchen Befehlen wie beeinflußt wird, ist wiederum von Prozessortyp zu Prozessortyp unterschiedlich. Es ist daher am besten, in Zweifelsfällen im Programmierhandbuch des CPU-Herstellers nachzusehen.

Indizierte Adressierung

Die beiden Index-Register des 6502 blieben bisher unbenutzt. Sie lassen sich jedoch recht nützlich einsetzen, wie das folgende Programmbeispiel zeigt, dessen Aufgabe es ist, den Speicherbereich 0301...037F voller Nullen zu schreiben. (Der Mikroprozessor 6800 gestattet es, den Inhalt beliebiger Adressen ohne den Umweg über den Akku auf Null zu setzen (CLR). Beim 6502 ist dies leider nicht möglich.)

```
0200 A9 00     LDA  # 00
0202 A2 7F     LDX  # 7F
0204 9D 00 03  STA  0300,X
0207 CA        DEX
0208 D0 FA     BNE  0204
0204 4C ...    JMP  Monitor
```

Das X-Register dient hier als Adressenversatz für den STA-Befehl. Die tatsächliche Adresse, an die der Akkuinhalt gespeichert wird, ergibt sich aus der Summe der dem Operationscode 9D folgenden beiden Adreßbytes und dem Inhalt des X-Registers. Nach dem Programmstart wird zunächst der Akku mit 00 und das X-Register mit 7F geladen. Dann wird der Akku an die Adresse (0300 + X) gespeichert, was zunächst 037F ergibt. Schließlich wird das X-Register um eins erniedrigt (dekrementiert), und das Spiel beginnt von neuem.

Auch der DEX-Befehl beeinflußt das Status-Register. Und deshalb läuft die Programmschleife nur solange, bis X Null geworden ist. Dafür sorgt der Befehl BNE, sprich „Branch on Not Equal". Seltsamerweise hat der Disassembler hier eine Zwei-Byte-Adresse hinter den mnemonischen Befehl BNE gesetzt, obwohl es sich um einen Zwei-Byte-Befehl mit nur einem Byte als Argument handelt. Wieso dieses?

Relative Adressierung

Der 6502 kennt zwei Sorten von Sprungbefehlen: bedingte und unbedingte. Erstere heißen „Branch" (Verzweigung), letztere einfach JMP für Jump.

Während bei JMP stets eine Zwei-Byte-Adresse folgt, steht hinter den Branch-Befehlen nur ein Byte, das einen Adressenversatz angibt, der positiv (00...7F) oder negativ (80...FF) sein kann. 00 entspricht der dem Branch-Befehl unmittelbar folgenden Adresse, bei 02 würden die folgenden zwei Bytes übersprungen usw. Etwas komplizierter als bei Vorwärtssprüngen ist es, wenn der Branch-Befehl nach „rückwärts" verzweigt. Dabei gibt es eine einfache Methode, sich große Rechnereien zu ersparen: Man braucht nur (hexadezimal) abzuzählen, wo man hinspringen will. Will man z. B. wie bei unserem letzten Programmbeispiel sechs Byte zurückspringen, so braucht man nur, beginnend mit der dem Branch-Befehl folgenden Adresse 020A, bis 0204 zurückzuzählen: 00, FF, FE, FD, FC, FB, FA – und schon sind wir bei 0204, der Zieladresse des bedingten Sprunges. Schon nach kurzer Übung kann man das hexadezimale Vor- und Rückwärtszählen in- und auswendig.

Auch hier ist wieder größte Vorsicht geboten, um sicherzustellen, daß der Sprung auch tatsächlich zum gewünschten Operationscode führt; springt das Programm fälschlich auf irgendwelche Daten, so läuft es u.U. „Harakiri".

Branch-Befehle verzweigen nur dann, wenn bestimmte Flags im Statusregister gesetzt oder gelöscht sind. In unserem Beispiel wird der BNE-Sprung

nur dann ausgeführt, wenn das Ergebnis der letzten Operation (DEX) ungleich Null war, wenn also das Zero-Flag in der CPU gelöscht war. Der Befehl BEQ verzweigt dagegen bei gesetztem Zero-Flag. Andere Branch-Befehle verzweigen abhängig von den Carry-, Overflow- oder Negativ-Flags. Manche Prozessoren – der 6502 leider nicht – verfügen auch über die Möglichkeit, abhängig von einem „Parity-Bit" zu verzweigen, das eine Prüfsumme des Akkuinhaltes darstellt.

Der 6502 kennt auch keinen unbedingten Sprung mit relativer Adressierung. Die Abhilfe ist hier allerdings einfach. Sie besteht aus der Befehlsfolge CLC (Clear Carry) und BCC (Branch on Carry Clear) oder Gleichwertigem, z. B. SEC/BCS. Beim 6800 ist dieser Trick nicht erforderlich.

Das Monitorprogramm

An dieser Stelle muß auf etwas eingegangen werden, das bisher als selbstverständlich angesehen wurde, aber doch einige Bemerkungen wert ist: das Monitorprogramm.

Wie schon einmal erwähnt, steht in einem Mikrocomputer dieses Programm im ROM, damit es auch nach dem vorübergehenden Abschalten der Versorgungsspannung wieder zur Verfügung steht. Nun gibt es aber doch Systeme, die gar kein ROM haben – wie kommt dann das Programm hinein? Ein solches System ist z. B. das von Horst Pelka 1977 in der FUNKSCHAU beschriebene 8080-System: Hier wurden einfach die Daten über den Datenbus bei gleichzeitigem Einstellen der Adresse mit Schaltern ohne Hilfe der CPU direkt vom Programmierer in das System-RAM eingegeben.

Die Methode, ein Programm binär mit Schaltern für Adressen und Daten einzugeben, ist zwar vielleicht lehrreich, aber zeitraubend, fehlerträchtig und oft entmutigend. Käufliche Einplatinen-Computer (KIM-1, AIM-65 usw.) besitzen daher ein ROM, das ein fest gespeichertes Programm enthält. Dieses sog. „Monitorprogramm" fragt z. B. ein Hexadezimal-Tastenfeld ab (beim AIM-65 sogar ein alphanumerisches), decodiert die Tasten und speichert die eingegebenen Daten an die gewünschten Adressen ab. Außerdem ermöglicht das Monitorprogramm (von Billigcomputern wie dem SCMP/MK-14 einmal abgesehen) auch das Abspeichern von fertig entwickelten Programmen auf eine normale Tonband-Kassette, damit man längere Programme nach dem Einschalten des Systems nicht jedesmal wieder neu eingeben muß. Meist wird auch die Anzeige von Adressen und Daten auf einem Siebensegment-Display oder einer alphanumerischen Anzeige mit Hilfe des Monitorprogramms realisiert; es kann z. B. per Software hexadezimale Zahlen in den Siebensegment-Code umwandeln.

Monitorprogramme können – je nachdem, wie komfortabel sie sind – etwa zwischen 1 KByte und 8 KByte im System-ROM einnehmen. Gewisse Teile des Monitorprogrammes, nämlich die Unterprogramme, können von einem Programm mitbenutzt werden, die der Anwender in das System-RAM geschrieben hat. Zu beachten ist schließlich noch, daß das Monitorprogramm einige wenige Adressen im RAM als Zwischenspeicher mitbenutzt, in denen keine Anwenderprogramme stehen dürfen, da diese sonst überschrieben werden.

Der „Kellerspeicher"

Unterprogramme

Wie in den höheren Programmiersprachen, so gibt es auch in der Maschinensprache die Möglichkeit, Unterprogramme zu verwenden. Was in BASIC als Befehl GOSUB heißt, wird beim 6502 mit JSR (Hex-Code 20) bezeichnet, und der BASIC-Befehl RETURN hat sein 6502-Äquivalent in RTS (Return from Subroutine, hex 60).

Unterprogramme sind immer dann sinnvoll, wenn eine gleichartige Befehlsfolge mehrmals im Programm gebraucht wird, z. B. um die drei Speicherzellen 0000, 0001 und 0002 mit Nullen zu füllen. Das Unterprogramm sähe dabei so aus:

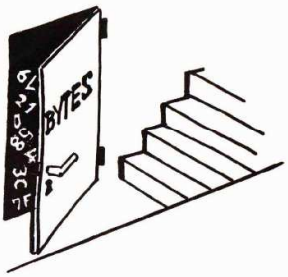
```
0250 A9 00    LDA  # 00
0252 85 00    STA  00
0254 85 01    STA  01
0256 85 02    STA  02
0258 60       RTS
```

Dem geneigten Leser wird auffallen, daß zum Abspeichern des Akkuinhaltes (00) die Zero-Page-Adressierung für die Adressen 0000...0002 verwendet wird. Am Ende des Programms steht der Rücksprung-Befehl RTS. Vorsicht: Starten Sie das Programm nicht an der Adresse 0250 ohne JSR-Aufruf! Es weiß sonst bei RTS nicht mehr, wohin es springen soll und läuft wiederum mit größter Wahrscheinlichkeit „Harakiri".

Ruft man dieses Unterprogramm auf, so geschieht das einfach mit JSR 0250, hexadezimal 20 50 02. Dies kann innerhalb des Hauptprogrammes beliebig oft geschehen. Auch kann ein Unterprogramm von einem „übergeordneten" Unterprogramm aufgerufen werden. Woher weiß es nun aber, wohin es zurückspringen muß, wenn die Subroutine-Befehlsfolge durchlaufen ist? Schließlich soll ja dann der dem JSR-Befehl folgende Befehl ausgeführt werden, d.h. das Unterprogramm soll zum nächsten Befehl des Hauptprogrammes zurückspringen. Irgendwie muß sich das Unterprogramm also die Rücksprungadresse „merken".

Der Stack als Adressenspeicher

Die CPU speichert die Rücksprungadresse bei Unterprogrammaufrufen schon während des JSR-Befehls in einen bestimmten Speicherbereich, den man „Stack" nennt; in der deutschen Literatur findet man ab und zu auch das abschreckende Wort „Kellerspeicher".



Beim 6502 liegt der Stack durch die CPU-interne Hardware fest im Bereich 0100...01FF, also in Page 1. Damit die CPU weiß, wohin sie mit der Adresse soll, die beim JSR-Befehl gespeichert werden muß, besitzt sie ein Register namens „Stackpointer", zu deutsch Kellerspeicherzeiger, das acht Bits umfaßt und vom Monitorprogramm des Mikrocomputers – solange kein Unterprogramm aufgerufen wird – auf FF gesetzt wird. Der Stackpointer zeigt also zunächst auf die Adresse 01FF.

Nehmen wir an, der Unterprogramm-Aufruf stünde an der Adresse 0220:

```
0220  20  50  02   JSR 0250
```

An der Adresse 0223 stünde dann der nächste Befehl im Programm, wo nach der Abarbeitung des Unterprogramms fortgefahren werden soll. Beim JSR-Befehl geschieht nun folgendes:

Die Adresse des letzten Bytes beim JSR-Befehl (hier 0222) wird in zwei Hälften gespalten. Der höherwertige Adressenteil (also 1 Byte, hier 02) wird an die Adresse gespeichert, auf die der Stackpointer zeigt (hier 01FF). Dann wird der Stackpointer um 1 erniedrigt, und der niederwertige Adressenteil (hier 22) wird an die Adresse 01FE gespeichert. Schließlich wird der Stackpointer nochmals dekrementiert, so daß er jetzt auf 01FD zeigt, und es erfolgt ein Sprung zum Unterprogramm.

Wenn dieses abgearbeitet ist, erfolgt bei RTS ein Rücksprung, indem der Stackpointer wieder inkrementiert wird; zuerst wird nun das niederwertige Byte und dann – nach nochmaliger Inkrementierung des Stackpointers – das höherwertige Byte der Rücksprungadresse in den Programmzähler der CPU transferiert. Der Stackpointer zeigt nun wieder auf 01FF. Bei verschachtelten Unterprogrammen, wenn also ein Unterprogramm ein oder mehrere weitere aufruft, müssen nacheinander mehrere Rücksprungadressen gespeichert werden, so daß der Stackpointer immer „tiefer" dekrementiert wird. Darauf ist zu achten, wenn man ein Programm in Page 1 stehen hat, das dann u.U. von den zu speichernden Rücksprungadressen überschrieben werden kann.

Da die Rücksprungadresse nicht die Adresse des nächsten Befehls im Hauptprogramm, sondern die des letzten Byte im JSR-Befehl ist, sorgt die CPU automatisch noch dafür, daß der Programmzähler bei RTS zusätzlich noch um 1 erhöht wird, so daß er auf den folgenden Befehl zeigt.

So rettet man den Akku auf den Stack

In unserem Unterprogramm-Beispiel, das drei Speicherzellen mit Null beaufschlagte, war natürlich nach dem Unterprogramm-Aufruf der Akkuinhalt gelöscht. Zuweilen stehen aber im Akku wichtige Dinge, die man auch nach einer längeren Befehlsfolge (die kein Unterprogramm zu sein braucht) wieder weiterverarbeiten möchte. Dann kann man sich der Befehle PHA und PLA bedienen. Das Beispiel macht sofort deutlich, wie das funktioniert.

```
PHA  Akku auf den Stack retten
...  Unterprogramm (zerstört den
     Akkuinhalt)
PLA  Akku vom Stack zurückholen
RTS
```

Der Befehl PHA speichert den Akkuinhalt an die durch den Stackpointer spezifizierte Adresse und erniedrigt dann diesen um 1. PLA dagegen inkrementiert den Stackpointer und lädt den Akku mit seinem ursprünglichen Inhalt.

Leider ist es nicht möglich, X- und Y-Register direkt auf den Stack zu „retten" – dies geht nur über den Umweg des Akkus. In dem obigen Beispiel kann es – wenn das Unterprogramm auch X und Y ändert – sinnvoll sein, die Befehlsfolge so zu erweitern:

```
PHA  Akku retten
TXA  X in den Akku transferieren...
PHA  ... und retten
TYA  ebenso Y
PHA
...  Unterprogramm
PLA  Y vom Stack holen
TAY
PLA  X vom Stack holen
TAX
PLA  Akku rückspeichern
RTS
```

Im Gegensatz zum 6502 erlauben es manche andere Mikroprozessoren, auch die übrigen Register ohne Umweg über den Akku auf den Stack zu retten, u.U. sogar automatisch beim JSR-Befehl.

Wieder andere Prozessoren besitzen gar keinen Stack; entweder, weil es bei ihnen keinen Unterprogramm-Befehl gibt (SCMP), oder, weil die Rücksprungadressen bzw. Registerdaten auf andere Weise gespeichert werden (TMS-9900-Familie).

Der Stackpointer

In den vorangegangenen Beispielen wurde deutlich, daß bei Unterprogrammaufrufen und PHA-Befehlen Daten und Adressen immer an die durch den Stackpointer spezifizierte Adresse abgespeichert werden. Da nach dem Einschalten der Versorgungsspannung der Stackpointer irgendwohin zeigen kann, wird er meist vom Monitorprogramm zunächst auf seinen Normalwert gesetzt, nämlich auf FF:

```
LDX  # FF
TXS
```

Die gleiche Befehlsfolge wird notwendig, wenn man aus einem Unterprogramm nicht über RTS, sondern – z. B. wegen einer Programmverzweigung – über einen JMP-Befehl in das Hauptprogramm oder auch in das Monitorprogramm springen will. Der Stackpointer steht nun ja nicht mehr in seiner „Ruhestellung" FF und muß neu gesetzt werden. Dies ist besonders dann wichtig, wenn sich im Bereich ab 0100 (Stack-Page) Programme und Daten befinden, die sonst von dem immer tiefer rückenden Stackpointer überschrieben werden könnten.

Allgemein muß der Stackpointer immer dann korrigiert werden, wenn ein JSR-Befehl nicht durch RTS wieder aufgehoben wird. (Gleiches gilt für die noch zu besprechenden „Interrupts".) Vergißt man das, so wird u.U. nach und nach der ganze durch den Stackpointer adressierbare Speicherbereich überschrieben – und manche Prozessoren haben einen 16-bit-Stackpointer...

Ein- und Ausgabe über Tastatur und Display

Es gibt bei den heute üblichen Mikrocomputer-Systemen zwei Arten der Ein- und Ausgabe: Einmal über die auf der Platine vorhandene Tastatur und Anzeige bzw. über ein externes Terminal, oder aber über Eingabe-Ausgabe-Leitungen (I/O-Ports). Erstere erlaubt die Kommunikation mit dem Benutzer des Systems, zweitere dient zur Steuerung und Abfrage externer Geräte, die Meßergebnisse liefern oder die der Mikrocomputer automatisch steuern soll.

Display-Ansteuerung

Für die Abfrage des Tastenfeldes und die Anzeige auf dem Display oder Terminal besitzt das Monitorprogramm des Mikrocomputers praktisch immer geeignete Unterprogramme, die auch vom Anwenderprogramm her aufgerufen werden können. Sie sind aus dem Monitorprogramm-Listing im Systemhandbuch ersichtlich. Für die Computer KIM-1 und SYM-1 wurden die wichtigsten in FUNKSCHAU 1979, Heft 11, Seite 653, veröffentlicht (in diesem Heft findet sich auch die 6502-Operationscode-Tabelle). Beim AIM-65 sind die Monitor-Unterprogramme ausführlich im Handbuch kommentiert.

Als typisches Beispiel eines Einplatinen-Computers mit Siebensegment-Display und Hexadezimal-Tastatur soll hier der KIM-1 dienen. Er besitzt eine Anzeigeroutine im Monitorprogramm an der Adresse 1F1F. Sie stellt den Inhalt der Zero-Page-Zellen 00FB, 00FA, 00F9 auf dem sechsstelligen Display drei Millisekunden lang dar. Vor und nach ihrem Aufruf ist das Display dunkel. Will man eine ständige Anzeige erreichen, so muß das Anzeige-Unterprogramm in einer Schleife dauernd durchlaufen werden. Will man z. B. auf dem Display lauter Nullen anzeigen, könnte das Programm etwa so aussehen:

```
0200 A9 00     LDA  # 00
0202 85 FB     STA  FB
0204 85 FA     STA  FA
0206 85 F9     STA  F9
0208 20 1F 1F  JSR  1F1F
020B 4C 08 02  JMP  0208
```

Hier werden zunächst die drei Anzeigebuffer-Zellen gelöscht. Dann folgt eine Programmschleife, bestehend aus dem Unterprogramm-Aufruf zur Anzeige und einem Sprungbefehl. Hat man das Programm an der Adresse 0200 einmal gestartet, kommt man z. B. durch Drücken der Reset-Taste wieder heraus. Eine andere Möglichkeit, das Programm zu unterbrechen, ergibt sich aus der Fähigkeit der Anzeige-Routine, zu erkennen, ob irgendeine Taste auf dem KIM-1 gedrückt ist. Ist das nämlich nicht der Fall, so ist der Inhalt des Akkus bei der Rückkehr aus dem Unterprogramm Null. War eine Taste gedrückt, ist der Akkuinhalt ungleich Null, und das Zero-Flag im Statusregister wird rückgesetzt. Wollen wir dafür sorgen, daß durch Drücken einer Taste aus der Anzeige-Programmschleife zum KIM-Monitorprogramm gesprungen werden kann, so brauchen wir nur folgendes zu ändern:

```
020B F0 FB      BEQ 0208
020D 4C 4F 1C   JMP 1C4F
```

Leider ist der „Erfolg" beim Start dieses Programms deprimierend: Es erfolgt nämlich sofort ein Rücksprung zum Monitorprogramm an die Adresse 1C4F, so daß lediglich der Inhalt der Adresse 0000 auf dem Display und nicht etwa sechs Nullen erscheinen.

Warum? Nun, wir starten das Programm ja mit der Taste GO. Und diese Taste ist höchstwahrscheinlich auch noch nach der 3 ms dauernden Anzeigeroutine gedrückt. Da aber ja ein Rücksprung erfolgen soll, wenn eine Taste gedrückt ist, kommt die Programmschleife zur Anzeige von sechs Nullen nicht zustande.

Was nun? Wir müssen noch eine zweite Schleife einbauen, die dafür sorgt, daß die eigentliche Anzeigeschleife erst dann erreicht wird, wenn die Taste GO wieder losgelassen wird.

```
020B D0 FB      BNE 0208
020D 20 1F 1F   JSR 1F1F
0210 F0 FB      BEQ 020D
0212 4C 4F 1C   JMP 1C4F
```

Tatsächlich funktioniert es jetzt! Der Unterprogramm-Aufruf bei 0208 dient hier eigentlich nicht der Anzeige, sondern nur der Tastenabfrage. Ist noch eine Taste gedrückt, kann das Programm nicht zur Adresse 020D weiterrücken.

Andere Mikrocomputer haben eine solche Überprüfung, ob die letzte Taste noch gedrückt ist, schon innerhalb des Monitorprogramms „eingebaut". Außerdem muß z. B. beim AIM-65 das Display-Unterprogramm nicht in einer Schleife laufen, weil die Ansteuerung der einzelnen Display-Stellen nicht per Software geschieht. Vielmehr braucht dem AIM-Display nur per Unterprogramm ein neues Zeichen übergeben zu werden, das dann von rechts in das Display geschoben wird. In fast allen Fällen wird, wenn ein Zeichen auf ein Terminal oder einen ASCII-Fernschreiber ausgegeben werden soll, dieses Zeichen im Akku übergeben. Beim AIM-65 gilt dies auch für das 20stellige alphanumerische Display.

Tastatur-Abfrage

Der Computer KIM-1 besitzt an der Adresse 1F6A ein Monitor-Unterprogramm namens „GETKEY". Es fragt das hexadezimale Tastenfeld ab und kehrt mit dem hexadezimalen Wert der Taste im Akku zurück. Aber: Auch hier wird nicht abgefragt, ob die zuletzt gedrückte Taste immer noch niedergedrückt ist. An folgendem kleinen Programm wird das sofort deutlich:

```
0200 20 6A 1F   JSR 1F6A
0203 85 F9      STA F9
0205 20 1F 1F   JSR 1F1F
0208 4C 00 02   JMP 0200
```

Dieses Programm stellt den Wert einer gerade gedrückten Taste in den niederwertigsten beiden Display-Stellen in Form eines Byte dar. Ist keine Taste gedrückt, so wird 15 angezeigt. Es wird sofort deutlich, daß sich der Mikrocomputer den Wert einer Taste nicht „merkt", sondern sofort wieder vergißt, wenn sie losgelassen wird. Will man z. B. ein Programm schreiben, um von der Tastatur her mehrere Ziffern wie bei einem Taschenrechner in das Display zu schreiben, so wäre auch hier eine zusätzliche Abfrage nötig, ob die letzte Taste noch gedrückt ist.

Beim AIM-65 ist dies alles wiederum nicht nötig. Das Unterprogramm zur Tastatur-Abfrage wartet in einer Schleife automatisch so lange, bis die vorher gedrückte Taste losgelassen und eine neue Taste gedrückt wird. Um hier von der Tastatur aus auf dem Display zu schreiben, genügt beim AIM folgendes Programm:

```
0200 20 73 E9   JSR E973
0203 4C 00 02   JMP 0200
```

Die Routine an der Adresse E973 sorgt nicht nur für die Tastaturabfrage, sondern stellt die gedrückten Tasten auch gleichzeitig auf dem Display dar.

Ein- und Ausgabe per „Memory Map"

Ebenso wie der 6800 besitzt auch der Mikroprozessor 6502 keine besonderen Ein- und Ausgabe-Befehle. I/O-Ports werden vielmehr als gewöhnliche Speicheradressen betrachtet, und alle CPU-Befehle, die eine Speicheroperation beinhalten, können auf I/O-Ports angewandt werden. Der englische Ausdruck für Ports, die normale Speicheradressen sind, ist „memory-mapped I/O", etwa „Speicherlandkarten-Ein- und Ausgabe" (schrecklich!).

Eine Besonderheit ist auch, daß die in den Bausteinen 6520, 6522, 6530 oder 6532 integrierten I/O-Ports entweder als Eingang oder Ausgang deklariert werden können. Zu diesem Zweck sind jedem 8-bit-Port zwei 8-bit-Register, also zwei Speicheradressen zugeordnet, beim KIM u. a.:

1700 PA (Port A)
1701 PAD (Port-A-Richtung, „Direction")

Nehmen wir an, wir wollten eine Leuchtdiode mit Hilfe eines Schalters ein- und ausschalten. Damit der Mikrocomputer etwas zu tun hat, soll das nicht direkt, sondern über zwei I/O-Anschlüsse geschehen. Den Schalter schließen wir am Pin PA 7 und die Leuchtdiode am Pin PA 0 an (das ist ganz willkürlich gewählt).

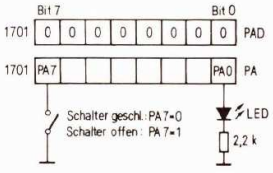
PA 7 muß also als Eingang, PA 0 als Ausgang geschaltet werden. Das geschieht durch Setzen entsprechender Bits im Datenrichtungs-Register PAD; eine Null bedeutet „Eingang", eine Eins „Ausgang". (Beim Drücken der Reset-Taste werden alle Ports als Eingänge geschaltet.) Das Bitmuster für PAD müßte dann so aussehen: 0000 0001. Nur PA 0 soll ja Ausgang sein; um das zu erreichen, beginnen wir unser Programm so:

```
0200 A9 01      LDA # 01
0202 8D 01 17   STA 1701
```

Der Akku wird mit hex 01 geladen, was binär dem gewünschten Bitmuster 0000 0001 entspricht, und an PAD abgespeichert. Das restliche Programm, um die LED mit dem Schalter ein- oder ausschalten zu können, sähe z. B. so aus:

```
0205 AD 00 17   LDA 1700
0208 10 04      BPL 020E
020A A9 01      LDA # 01
020C D0 02      BNE 0210
020E A9 00      LDA # 00
0210 8D 00 17   STA 1700
0213 4C 05 02   JMP 0205
```

Das Programm weist einige Besonderheiten auf. An der Adresse 0205 wird PA in den Akku geladen. Das höchstwertige Bit von PA wird dabei (wie bei allen Ladebefehlen des 6502) in das Negativ-Flag des Status-Registers übernommen. Abhängig von ihm (BPL = Branch on Plus, Sprung bei N-Flag = 0) wird entweder 01 oder 00 in den Akku geladen und an PA gespeichert, so daß PA 0 auf 1 oder 0 gesetzt wird. Der Befehl BNE bei 020C ist hier ausnahmsweise ein unbedingter Sprung: Vor ihm wurde der Akku mit 01 geladen, so daß das Zero-Flag des Statusregisters rückgesetzt ist und die Sprungbedingung (Branch on Not Equal) stets erfüllt ist. (Bei anderen Mikrocomputern müssen nur die Adressen von PA und PAD geändert werden; beim AIM-65 z. B. auf A001 und A003.)



Der BIT-Befehl

An der Adresse 0205 haben wir die Daten von PA (1700) in den Akku geladen, obwohl wir sie gar nicht weiterverarbeiten wollen. Vielmehr wollten wir ja nur das N-Flag im Statusregister abhängig von der Schalterstellung an PB 7 setzen oder rücksetzen; und dafür läßt sich auch der Befehl BIT einsetzen. Er hat auf das N-Flag die gleiche Wirkung wie der LDA-Befehl, aber ohne den Akkuinhalt zu ändern, was manchmal recht nützlich ist.

Außerdem beeinflußt er das Zero-Flag abhängig von einer AND-Operation zwischen Akku und der angesprochenen Adresse. Legen wir einmal den Schalter nicht an PA 7, sondern z. B. PA 3, so wird gleich deutlich, wie man das verwenden kann:

```
0200 A9 01      LDA  # 01
0202 8D 01 17   STA  1701
0205 A9 08      LDA  # 08
0207 2C 00 17   BIT  1700
020A D0 04      BNE  0210
020C A9 01      LDA  # 01
020E D0 02      BNE  0212
0210 A9 00      LDA  # 00
0212 8D 00 17   STA  1700
0215 4C 05 02   JMP  0205
```

Der BNE-Befehl bei 020A verzweigt hier in Abhängigkeit der AND-Operation zwischen Akku und PA, aber ohne den Akku zu verändern. Der Akku enthält während des BIT-Befehls hex 08, also das Bitmuster 0000 1000. An der Stelle von PA 3 ist also eine 1. Je nach dem Ergebnis (08 oder 00) der AND-Operation kann das Z-Flag also gesetzt oder rückgesetzt werden.

Das gleiche Programmierproblem könnte man auch mit dem AND-Befehl lösen; der Operationscode 2C müßte dann durch 2D ersetzt werden. Der einzige Unterschied, der hier allerdings keine Rolle spielt, ist, daß nach dem AND-Befehl im Akku tatsächlich das Ergebnis der Operation steht, während der BIT-Befehl nur die Status-Flags beeinflußt.

Viele Mikrocomputer kennen keinen BIT-Befehl. Bei ihnen wird das „Maskieren" des gerade abzufragenden Ports deshalb mit einem AND-Befehl vorgenommen, so daß im Akku nur die gerade interessierenden Bits des Ports übrigbleiben.

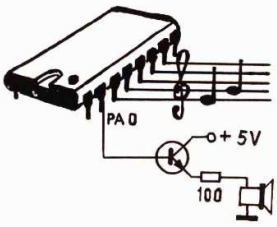
Bei den Prozessoren 8080 und Z-80 sind nicht Memory-Map-I/O-Bausteine üblich, sondern die Ein- und Ausgabe geschieht mit speziellen I/O-Befehlen der CPU, die dann an die Ein-Ausgabe-Bausteine ein Chip-Select-Signal schickt. Das hat allerdings den Nachteil, daß die I/O-Ports sich nicht in beliebige arithmetische Operationen (AND, OR, ADC usw.) einbeziehen lassen, da die I/O-Befehle nur einen Datentransfer erlauben. Andererseits hat diese Methode den Vorteil, für I/O-Operationen nur 2-Byte-Befehle zu benötigen, wenn man sich auf max. 255 Ports beschränkt; bei Memory-Map-I/O-Anordnungen sind dagegen normalerweise 3-Byte-Befehle nötig, da eine 2-Byte-Adresse angegeben werden muß. Die Erfahrung zeigt jedoch, daß die zu erwartende Geschwindigkeitseinbuße des Memory-Map-Verfahrens in der Praxis nicht existiert.

Hat man Töne?!

Eine recht nette Anwendung eines Mikrocomputers ist es, an einem I/O-Port einen Ton bestimmter Frequenz zu erzeugen. Sehen wir uns einmal ein einfaches Programm an, das dieses tut.

```
0200 A9 01      LDA  # 01
0202 8D 01 17   STA  1701
0205 A2 60      LDX  # 60
0207 CA         DEX
0208 D0 FD      BNE  0207
020A EE 00 17   INC  1700
020D 4C 05 02   JMP  0205
```

Das Programm, das für die I/O-Belegung des KIM-1 ausgelegt ist, erzeugt an PA 0 einen Ton, dessen Höhe abhängig von dem Inhalt der Speicherzelle 0206 ist. Damit wird nämlich das X-Indexregister geladen und in einer Verzögerungsschleife heruntergezählt, bis es Null ist.



Der INC-Befehl dient dazu, den Zustand von PA 0 umzudrehen. Bekanntlich wird beim Inkrementieren eines Byte jedesmal das niederwertigste Bit komplementiert; die höherwertigen interessieren uns hier nicht. (Hier zeigt sich der Vorteil des Memory-Map-Verfahrens: Die Methode mit dem INC-Befehl ist bei Prozessoren, die besondere Ein- und Ausgabe-Befehle haben, nicht möglich.)

Mit Hilfe des Programmier-Handbuches läßt sich errechnen, wie hoch die erzeugte Frequenz je nach dem Inhalt des X-Registers ist. Das KIM-1-Handbuch gibt ein Beispiel, wie man die erzeugte Frequenz mit Hilfe von Schaltern verändern kann – dabei wird ein I/O-Port abgefragt.

Dieses Tonerzeugungs-Programm ist ein typisches Beispiel dafür, daß man nicht alle Software-Probleme in einer höheren Programmiersprache wie BASIC lösen kann. Während für einen BASIC-Befehl u.U. mehrere Millisekunden gebraucht werden, begnügen sich die Maschinenbefehle mit wenigen Mikrosekunden. Die genauen Befehls-Ausführungszeiten hängen vom Prozessor, von der Adressierungsart und von der CPU-Taktfrequenz ab und können in den Software-Handbüchern und Befehlstabellen der CPU-Hersteller nachgeschlagen werden. Die Zeiten werden meist als Zyklus-Zahlen angegeben, was beim 6502 und 1 MHz Taktfrequenz (AIM-65, KIM-1, SYM-1, PC-100) gleichzeitig die Befehlszeit in Mikrosekunden ergibt.

Läßt man die Verzögerungsschleife (0205...0208) weg, so kann man mit Hilfe der nun maximal erzeugbaren Frequenz an einem I/O-Port einfache Vergleiche der Geschwindigkeit unterschiedlicher Prozessoren anstellen. Bei 1 MHz Taktfrequenz ergeben sich mit dem 6502 etwa 55 kHz. (Derzeit existiert die CPU 6502 in unterschiedlichen Versionen für max. 2 MHz Taktfrequenz.)

Ist das alles?

Wir haben inzwischen eine Reihe von Adressierungsarten und Befehlen kennengelernt. Natürlich soll das Papier, das Sie gerade vor sich haben, nicht das Programmier-Handbuch Ihres Mikroprozessors ersetzen. Viele Befehle, die dort aufgeführt sind, bedürfen auch gar keiner weiteren Erklärung, weil ihre Bedeutung schon aus ihrer Bezeichnung hervorgeht. Sinn der Sache ist es hier vielmehr, jene Dinge etwas näher zu beleuchten, die in manchen Programmierbüchern als selbstverständlich vorausgesetzt werden und dann zu Verständnisschwierigkeiten, ja zur Entmutigung führen.

Dabei ist das Programmieren in Maschinensprache recht einfach, wenn man einmal die notwendigsten Grundbegriffe und die wichtigsten Befehle „seines" Mikroprozessors kennt. Voraussetzung ist lediglich die Bereitschaft zum streng logischen, konsequenten Denken in kleinsten Schritten – nämlich in CPU-Befehlen.

Wenn Sie bis hierher mitgemacht und die Programme auch verstanden haben, so dürfte es Ihnen keine Schwierigkeiten bereiten, einfache Programme selbst zu schreiben, zum Beispiel eine 16-bit-Addition oder ein Programm, das die Anzeige auf dem KIM-1 blinken läßt.

Die ersten Programme, die man schreibt, sind immer ziemlich umständlich. Später stellt man fest, daß alles viel einfacher gegangen wäre, wenn man geeignetere Befehle angewandt

hätte, mit denen man erst nach einiger Zeit vertraut wurde. Nicht selten kann man seine „Erstlingswerke" um ein Drittel verkürzen.

Etwas, was man tatsächlich erst nach einigem Überlegen und einiger Erfahrung versteht, ist die indirekt indizierte Adressierungsart – eine nützliche Eigenart des 6502.

Indirekt indizierte Adressierung

Wir haben bereits ein Programm gesehen, das einen bestimmten Speicherbereich mit Nullen auffüllt; es war der Bereich 0301...937F. Die damals gewählte Methode der indizierten Adressierung mit dem nur 8 bit langen X-Index-Register beschränkt den überstreichbaren Bereich auf maximal hex FF (dezimal 255)Byte, also genau eine „Page". Was ist aber zu tun, um z. B. den Bereich 0200...03FF zu löschen, also das obere halbe KByte des KIM-1? Natürlich könnte man zwei solche Programme aneinanderketten, die jeweils eine Page löschen – bei größeren Speicherbereichen wird das schnell uneffektiv. (Andere Mikroprozessoren, z. B. der 6800, besitzen ein 16-bit-Indexregister, das dieses Problem löst – allerdings eben nur eines...)

Beim 6502 bedient man sich zum Überstreichen größerer Speicherbereiche der „indirekt indizierten Adressierung". Dabei werden zwei aufeinanderfolgende Zero-Page-Zellen als Indexregister verwendet, was praktisch einem 16-bit-Register entspricht. Dieser Trick ermöglicht es dem 6502, über 128 Register mit 16 bit Länge zu verfügen, denn die Zero Page umfaßt ja 256 Byte. Aus diesem Grunde ist es ungerecht, die wenigen internen Register der CPU 6502 mit dem umfangreicheren Registersatz von Prozessoren wie dem Z-80 zu vergleichen. Die scheinbare Schwäche wird durch eine Vielzahl nützlicher Adressierungsarten wettgemacht. Das Programm zum Löschen des Bereiches 0200...03FF sieht so aus:

```
0000 A9 00     LDA  # 00
0002 85 E0     STA  E0
0004 A9 02     LDA  # 02
0006 85 E1     STA  E1
0008 A0 00     LDY  # 00
000A A9 00     LDA  # 00
000C 91 E0     STA  (E0),Y
000E E6 E0     INC  E0
0010 D0 02     BNE  0014
0012 E6 E1     INC  E1
0014 A5 E1     LDA  E1
0016 C9 04     CMP  # 04
0018 D0 F0     BNE  000A
0019 4C ...    JMP  Monitor
```

Was geschieht hier? Zunächst werden die Zero-Page-Zellen 00E0 und 00E1 mit der Anfangsadresse des zu löschenden Bereichs geladen, wobei das höherwertige Adressenbyte in 00E1 steht. Akku und Y-Register werden dann mit 00 geladen – und jetzt wird der Akkuinhalt an diejenige Adresse gespeichert, die sich aus dem Inhalt der Adressen 00E0/00E1 plus dem Y-Index ergibt. Zunächst ist dies 0200. Die Befehlsfolge an den Adressen 000E...0012 erhöht die 16-bit-Adresse im Zellenpaar 00E0/00E1. Schließlich wird noch durch Abfragen des höherwertigen Adressenteils in 00E1 abgefragt, ob bereits Page 4 erreicht ist; wenn nein, wiederholt sich das Spiel für die nächste Adresse (0201), wenn ja, erfolgt ein Rücksprung zum Monitorprogramm, z. B. an die Adresse 1C4F beim KIM-1.

Dieses Programm ist eines der wenigen Beispiele, die die Programmierung des 6502 ein wenig umständlich erscheinen und die Herzen der Besitzer anderer Prozessoren höher schlagen lassen.

Von der Möglichkeit, zusätzlich einen Adressenversatz durch das Y-Register zu erreichen, wurde hier kein Gebrauch gemacht – Y bleibt während des ganzen Programmes Null. Übrigens können geübte Programmierer dieses Programm ohne weiteres um zwei oder mehr Byte verkürzen – versuchen Sie's mal! (Ein Hinweis: Man kann zu Beginn auch das Y-Register verwenden, um die Null in 00E0 zu laden.)

Die indirekt indizierte Adressierung gibt es beim 6502 nur in Verbindung mit dem Y-Register. Dem X-Register ist eine Adressierungsart vorbehalten, die nicht minder kompliziert ist.

Indiziert-indirekte Adressierung

Die indiziert-indirekte Adressierung wird nicht so häufig gebraucht und soll daher nur kurz vorgestellt werden. Ein Beispiel hilft, diese komplizierte Adressierungsart zu verstehen. Der Befehl STA (E0,X), der dem vorhin gebrachten Beispiel etwas ähnelt, tut folgendes:

Ist der Inhalt des X-Registers z. B. 07, so wird zunächst einmal die Summe von E0 und 07 errechnet, sie ergibt E7. Dann wird der Akku an jene Adresse gespeichert, die im Zellenpaar 00E7/00E8 steht.

Dieser Befehl (hex 91 E0) kann alternativ in unserem letzten Programm verwendet werden, wenn man das X-Register statt dem Y-Register vorher auf Null setzt. In diesem Fall (und nur dann) ist die Wirkung von indiziert-indirekter und indirekt indizierter Adressierung identisch.

Während man anfangs mit vielen Adressierungsarten noch nichts so recht anzufangen weiß, ärgert man sich nach einiger Zeit darüber, daß nicht für jeden Befehl jede Adressierungsart zur Verfügung steht. Leider haben die Entwickler des 6502 hier ein wenig gespart. Nur der 16-bit-Mikroprozessor 68 000 von Motorola kann hier als Musterknabe betrachtet werden: Für alle Befehle ist jede (sinnvolle) Adressierungsart vorhanden.

Interrupt-Verarbeitung

Bisher waren unsere Programme auf Neuhochdeutsch „straightforward" – sie bearbeiteten irgendwelche Daten, führten ab und zu Sprungbefehle oder Unterprogramme aus, ließen sich sonst aber durch nichts aus der Ruhe bringen.

Halt – nur durch eines natürlich: Wenn wir auf die Reset-Taste unseres Mikrocomputers drückten, hörte unser Anwenderprogramm mit der Ausführung seiner Befehle auf, und es war wieder möglich, mit der Tastatur und dem Display Adressen und Daten zu ändern. Die Reset-Taste bewirkt also einen gewaltsamen, durch Hardware hervorgerufenen Sprung zum Monitorprogramm, der durch nichts, aber auch gar nichts verhindert werden kann. Der Sprung wird dadurch ausgelöst, daß ein bestimmter Pin der CPU – die Reset-Leitung – durch Tastendruck kurz auf log. 0 (Low) gelegt wird.

Was passiert hier? Sofort, wenn die Reset-Leitung auf Low geht, lädt die CPU ihren internen Programmzähler mit einer Adresse, die – hardwaremäßig festgelegt – beim 6502 in den Adressen FFFC und FFFD steht. (Verwechseln Sie hier nicht Adresse mit Adresse – die Zellen FFFC/FFFD enthalten zwei Byte, die die CPU als neue Adresse betrachtet.) Dann fährt die CPU mit der Ausführung der Befehle am neuen Programmzählerstand fort, nämlich mit den Befehlen des Monitorprogramms. Das ist alles, was ein „Reset" in der CPU bewirkt. Eine solche hardwaremäßige, gewaltsame Programm-Unterbrechung bezeichnet man allgemein als Interrupt. Der 6502 kennt drei Interruptarten: Reset (siehe oben), NMI und IRQ.

NMI

NMI heißt „non-maskable interrupt" – das hat er eigentlich mit dem Reset gemeinsam: Er läßt sich softwaremäßig nicht verhindern. In einem unterscheidet er sich aber vom Reset: Wie bei einem Unterprogramm wird die Rücksprungadresse und hier zusätzlich noch der Inhalt des Statusregisters auf den Stack „gerettet". Der Sprung beim Auftreten des NMI erfolgt hier über den Inhalt des Zellenpaares FFFA und FFFB. Stehen dort z. B. die Daten 05 A3, so springt der Prozessor an die Adresse A305. Dort steht dann irgendeine Routine, die irgend etwas bearbeitet. Der Trick: Weil die Rücksprungadresse auf dem Stack abgespeichert wurde, kann man mit dem Befehl RTI wieder in das unterbrochene Programm zurückkehren, als wäre nichts geschehen – sogar das Statusregister hat dann wieder seinen alten Inhalt, weil RTI im Gegensatz zu RTS auch den Prozessorstatus vom Stack zurückholt.

IRQ

Eine dritte Interrupt-Leitung der CPU heißt „IRQ". Dieser Interrupt unterscheidet sich vom NMI dadurch, daß er per Software verhindert werden kann. Dafür gibt es den Befehl SEI = Set Interrupt Disable Flag. Vom Interrupt-Flag im Status-Register hängt es ab, ob ein Low-Signal am IRQ-Eingang der CPU tatsächlich zu einem Interrupt führt oder nicht. Ebenso wie beim NMI werden beim Auftreten des „Interrupt Request" Rücksprungadresse und Statusregister auf den Stack gerettet. Der IRQ-„Vektor" steht an den Adressen FFFE und FFFF.

Der Break-Befehl

Der IRQ-Interrupt kann softwaremäßig simuliert werden, nämlich mit dem BRK-Befehl (Operationscode 00). Auch hier werden Statusregister und Programmzähler auf den Stack gerettet. Der Break-Befehl ist beim Durchtesten von Programmen recht nützlich, da es mit ihm möglich ist, das Programm an beliebigen Stellen anzuhalten, indem man BRK einfügt.

Stackpointer-Korrektur

Erfolgt nach einem Interrupt oder nach dem Break-Befehl kein Rücksprung mit RTI, so muß man den Stackpointer wieder korrigieren (s.a. „Unterprogramme"), da er ja nicht mehr in seiner Normallage steht (FF). Dies kann wieder mit der Befehlsfolge
LDX # FF
TXS
erfolgen. Vergißt man dies, so können wiederum wichtige Daten oder Programmteile im Stackbereich (beim 6502 in Page 1) überschrieben werden. Bei Prozessoren, deren Stackpointer nicht 8, sondern 16 bit lang ist, kann das zu chaotischem Verhalten führen.

Interrupt-Quellen

Was kann überhaupt einen Interrupt auslösen? Nun, prinzipiell alles, was in der Lage ist, an einen Interrupt-Pin der CPU einen Low-Impuls zu liefern. Beim KIM-1 kann das z. B. die STOP-Taste sein, oder aber, wenn man den Timer-Ausgang PB7 mit NMI oder IRQ verbindet, auch ein auf dem Mikrocomputer befindlicher Timer, der mit einer bestimmten Zeit geladen werden kann und nach Ablauf dieser einen Interrupt auslöst. Eine hübsche Anwendung dieses Timer-Interrupts ist u.a. eine Software-Uhr, die für den KIM-1 in FUNKSCHAU 1979, Heft 11, Seite 657 beschrieben ist.

Die Register müssen auf den Stack

Wie bei manchen Unterprogrammen kann es notwendig sein, die CPU-Register beim Auftreten von Interrupts auf den Stack zu „retten". Da im Gegensatz zu Unterprogrammen aber nicht vorhergesagt werden kann, wo im Hauptprogramm der Interrupt auftritt – das kann bei jedem beliebigen Befehl geschehen – muß man in der Interrupt-Routine *alle* Register „retten", die in ihr verändert werden. Normalerweise ist dies der Akku, in manchen Fällen auch die Indexregister X und Y. Ein Beispiel: Beim Auftreten eines Interrupts soll der Inhalt des Ports A beim KIM-1 komplementiert werden (Port A muß dabei natürlich als Ausgang geschaltet sein; nur dann können alle seine Bits komplementiert werden).

Die Interrupt-Routine muß dann z. B. so aussehen:

0280	48	PHA
0281	AD 00 17	LDA 1700
0284	49 FF	EOR # FF
0286	8D 00 17	STA 1700
0289	68	PLA
028A	40	RTI

Das Komplementieren wird hier mit dem EOR-Befehl (Exklusiv-Oder) im Akku erreicht. Damit der Akku-Inhalt bei der Rückkehr in das Hauptprogramm (hier nicht aufgelistet) nicht verändert ist, wird der Akku zu Beginn der Interrupt-Routine auf den Stack gerettet und vor ihrem Ende zurückgeholt. Die obige Routine kann beim KIM-1 leicht durch Drücken der STOP-Taste prüfen, wenn der NMI-Vektor in den Zellen 17FA und 17FB auf die Adresse 0280 zeigt, d.h. in 17FA muß 80 und in 17FB muß 02 stehen.

Indirekte Sprünge

In den meisten Mikrocomputern wird der NMI- oder IRQ-Vektor nicht wirklich aus FFFA/FFFB bzw. FFFE/FFFF geholt, sondern aus Zellen im RAM-Bereich, zu denen ein indirekter Sprung aus dem Monitorprogramm führt. Die Vektoren in FFF... zeigen dabei auf diesen indirekten Sprungbefehl im Monitor-ROM. Beim KIM-1 sieht das so aus: Der Vektor in FFFA/FFFB zeigt auf die Adresse 1C1C. Dort wiederum, also im Monitor-ROM, steht der Befehl JMP (17FA), hex 6C FA 17. Das Programm springt daraufhin an die Adresse, die der Anwender in die RAM-Zellen 17FA und 17FB geschrieben hat.

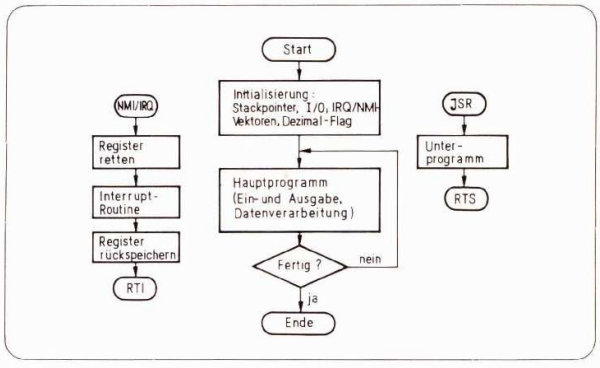
Über das Schreiben von Programmen

Mittlerweile haben wir schon das notwendige Rüstzeug, um fremde Programme ohne große Schwierigkeiten verstehen zu können; das ist oft nur eine Frage der Konzentration und Geduld.

Längere Programme selbst zu schreiben, erfordert schon etwas Erfahrung. Jedem Anfänger ist zu raten, erst einmal fertige Programme zu betrachten, eventuell gezielt Änderungen vorzunehmen und deren Wirkungen zu beobachten. Dadurch lernt man auch am schnellsten die unterschiedlichen Befehle und Adressierungsarten kennen. Einen guten Anhaltspunkt bieten oft die gut kommentierten Monitor-Programm-Listings in den Systemhandbüchern, wie etwa der Mikrocomputer KIM-1, SYM-1 oder AIM-65.

Das Schreiben von Programmen besteht grundsätzlich aus vier wichtigen Abschnitten:

1. *Exakte Definition der Problemstellung*
2. *Umsetzen des Problems in eine logische Folge (z. B. Flußdiagramm)*
3. *Erstellen des Maschinenprogrammes*
4. *Testen und Korrigieren des Programms*



Entgegen der Annahme vieler Neulinge beansprucht nicht etwa Punkt 2 oder 3 am meisten Zeit, sondern Punkt 4. Kein Programm läuft auf Anhieb!

Falsche Zeitvorstellungen existieren auch oft über Punkt 1. Eine exakte Problemdefinition vor dem Schreiben des Programms erspart umständliche, später kaum noch durchführbare Änderungen im Programmablauf. Dazu gehört speziell die Berücksichtigung von Randbedingungen, die z. B. bei ungewöhnlichen und unzulässigen Eingaben durch den späteren Programm-Benutzer oder durch u.U. unwahrscheinliche Konstellationen von Daten auftreten. Was passiert, wenn der Computer eine Zahl erwartet, der Benutzer aber den Buchstaben Z drückt? Wie soll der Computer reagieren, wenn zu viele Ziffern eingegeben werden? Was geschieht, wenn ein Kontakt an einem Eingangs-Port prellt? Alle diese Dinge gehören zu einer exakten Definition der Problemstellung.

Mit der Problemdefinition haben wir natürlich das Problem noch nicht gelöst. Oft muß man auch in das Erstellen des Flußdiagramms noch eine Menge Hirnschmalz investieren, da das Problem ja meist nicht als logische, zeitlich gestaffelte Instruktionsfolge vorliegt, sondern eher in einer chaotischen, unüberschaubaren Form.

Dann ist es soweit: Aus dem Flußdiagramm muß irgendwie der hexadezimale Maschinencode gewonnen werden. Bei einfachen Systemen wie dem KIM-1 oder SYM-1 bedeutet das, daß man – wenn man die Befehle noch nicht auswendig kennt – jeden Operationscode in einer Tabelle nachsehen muß. Der AIM-65 hat eine solche Tabelle eingebaut, nämlich im ROM, und übernimmt die Übersetzung der mnemonischen Befehle in den Hex-Code selbst; z. B. speichert er A9 ins RAM, wenn man die Tasten LDA # drückt. Einen solchen Übersetzer nennt man Assembler – die Rückübersetzung heißt dementsprechend Disassemblierung.

Für das Schreiben von Maschinenprogrammen gibt es einige Regeln, die man sich merken sollte:

1. Verwenden Sie bei Sprungbefehlen möglichst immer die relative Adressierung. Das spätere Verschieben von Programmteilen wird damit problemlos, auch wenn man keinen Assembler mit symbolischer „Label"-Adressierung hat.

2. Vermeiden Sie unbedingt Programme, die sich selbst ändern, d.h. die selbst Operationscodes innerhalb des Programmes ändern. Erstens läuft das Programm dann nur im RAM und nie im EPROM, zweitens weiß man später, wenn man das Programm ausdruckt, nie genau, in welchem „Zustand" es gerade war.

3. Wählen Sie keine „krumme" Startadresse, die sich niemand merken kann, sondern eine gerade, wie z. B. 0000 oder 0200.

4. Meiden Sie den Stackbereich (0100...01FF)! Wenn ein Programm wegen eines kleinen Fehlers einmal „Harakiri" läuft, ist der Stackbereich der erste, der ihm zum Opfer fällt. Dann ist man erst einmal damit beschäftigt, das Programm neu einzutippen.

5. Machen Sie sich eine genaue Aufstellung, welche Zellen außerhalb des Programms noch belegt werden, z. B. Zero-Page-Adressen. Und überhaupt: Versuchen Sie schon während des Programmierens, Ihre Software so zu dokumentieren, daß Sie das Programm auch noch nach einem Jahr verstehen können!

6. Wenn das Programm Interrupt-Routinen enthält, so sollte man es nicht dem Erinnerungsvermögen des Benutzers überlassen, vor dem Start die IRQ- oder NMI-Vektoren zu setzen. Das sollte besser gleich nach dem Start des Hauptprogrammes automatisch geschehen.

7. In der endgültigen Form sollten die einzelnen Programmteile nicht gleichmäßig über den gesamten RAM-Bereich des Mikrocomputers verteilt sein, sondern aneinandergehängt werden, damit man das Komplettprogramm auf einmal z. B. auf eine Kassette aufzeichnen oder auch in ein EPROM laden kann.

8. Unterprogramme des Monitor-ROM sollten nur dort verwendet werden, wo es wirklich notwendig ist. Andernfalls ist eine Adaption des Programmes an andere Mikrocomputersysteme oft unmöglich.

9. Vermeiden Sie möglichst Operationscodes, die zwar offensichtlich funktionieren, aber nicht im Programmierhandbuch des CPU-Herstellers verzeichnet sind. Diese Codes funktionieren u.U. bei CPUs anderer Hersteller nicht oder führen bei bestimmten Datenkonstellationen zu Problemen.

10. Legen Sie sich ein Telefon mit abschaltbarer Klingel zu. Zum Programmieren braucht man Konzentration!

Mit diesen „zehn Geboten für den Programmierer" wollen wir unseren Streifzug durch die Maschinenprogrammierung beenden. Sicher sind Sie jetzt in der Lage, Programme, die in Zeitschriften und Büchern veröffentlicht sind, zu verstehen – und manchmal sogar zu verbessern.

Stichworte zum Inhalt

Maschinenprogrammierung, Einführung, Grundlagen, hexadezimal, ASCII, Register, Adressierungsarten, Monitorprogramm, Stack, Memory Map, Interrupt.

Die „unterstrichenen" BASIC-Listings

An mehreren Stellen in diesem Heft finden Sie Programme in BASIC, die mit einem Centronics-779-Drucker aufgelistet wurden. Dieser Drucker ist zwar in der Lage, alle PET-Grafik-Zeichen wiederzugeben, bei der inversen Zeichendarstellung macht er aber nicht mehr mit und druckt dann einfach in der normalen Zeichenform. Wenn Sie in den Listings einzelne Zeichen unterstrichen finden, so bedeutet das, daß diese Zeichen auf dem PET-Bildschirm invers erscheinen; meist handelt es sich dabei um Steuerzeichen, z. B. Cursorbewegungen innerhalb eines PRINT-Befehls. Die Inversdarstellung wurde von den Drucker-Entwicklern nicht realisiert, weil sie den Druckknopf zu sehr abnützen würde; Grafikzeichen sind möglich.