Assembler ist keine Alchimie (Teil 5)

In dieser Folge des Assembler-Kurses wird die relative Adressierung erklärt. Damit verbunden sind auch die wichtigen Vergleichsoperationen. Anhand einer sehr häufig verwendeten Betriebssystem-Routine können Sie Ihr neu erworbenes Wissen testen.

In der letzten Ausgabe haben wir die Branch-Befehle kennengelernt. Heute wollen wir uns mit der relativen Adressierung dieser Befehle und noch einer anderen Art der Adressierung befassen. Weiterhin werden Sie einige neue Assembler-Worte lernen, nämlich die Vergleichsbefehle. Wie ganze Zahlen im Computer gespeichert sind, wissen wir bereits. Heute untersuchen wir die Speicherung von Zeichen. Schließlich werden wir unsere Nase noch ein wenig in die eingebaute Software des C 64 stecken.

Die relative Adressierung

Als wir den BNE-Befehl das erstemal verwendet haben, stellten wir fest, daß zum Beispiel BNE 1200 nicht — wie eigentlich zu erwarten war — ein 3-Byte-Befehl, sondern ein 2-Byte-Befehl ist. Damals mußten wir uns mit der Bemerkung zufrieden geben, es läge an der besonderen Art der Adressierung, nämlich der relativen Adressierung. Relativ bedeutet ja »bezogen auf etwas«. Wenn wir also beispielsweise BNE 1200 schreiben, liegt es nur an der Benutzerfreundlichkeit des SMON und vieler anderer Assembler, daß dieser die so geschriebene absolute Adresse 1200 in die richtige Form, nämlich die relative umrechnet. In Wahrheit verlangt der 6502 (und natürlich ebenso der 6510) eine Angabe darüber, wieviele Bytes nach vorne oder hinten im Programm er zur weiteren Programmverarbeitung springen (verzweigen) soll. Es gilt nun also, zwei Fragen zu klären:

1. Relativ wozu wird gesprungen und
2. Wie berechnet sich die Angabe, um wieviele Bytes nach vorne oder hinten im Programm der Sprung vollzogen werden soll.

Zur Klärung verwenden wir ein hypothetisches Programmsegment mit einem Sprungbefehl und sehen uns das Disassembler-Listing an:

```
2000 AD 00 30    LDA 3000
2003 F0 05       BEQ 200A
2005 A9 00       LDA #00
2007 8D 00 30    STA 3000
200A 60          RTS
```

Dieses Programm-Teilchen lädt den Inhalt der Speicherstelle 3000 in den Akku, überprüft dann, ob dieser Inhalt null ist und verzweigt beim Vorliegen der Null zum Rücksprung (RTS). Ist der Inhalt von 3000 nicht Null, dann wird 3000 auf Null gesetzt. 3000 könnte zum Beispiel eine Flagge sein.

Der Pfad, dem der Computer bei der Abarbeitung des Programmes folgt, wird durch den Programmzähler vorbereitet. Dieser ist dann, wenn der BEQ-Befehl an der Reihe ist, schon einen Schritt weiter, nämlich im Programmzähler steht dann die Adresse 2005.

Relativ zu dieser Adresse hat dann der Sprung zu erfolgen. Zum Inhalt des Programmzählers muß also die Sprungweite (auch häufig Offset genannt) addiert werden. Soweit zur Frage 1.

Zur Klärung von Frage 2 listen wir uns mal Byte für Byte unser Programm auf:

Byte		Inhalt	Bedeutung
2000		AD	LDA
2001		00	LSB von 3000
2002		30	MSB von 3000
2003		F0	BEQ
2004		05	Offset
2005		A9	LDA #
2006	1	00	
2007	2	8D	STA
2008	3	00	LSB von 3000
2009	4	30	MSB von 3000
200A	5	60	RTS

Neben der Byte-Nummer ist noch die Entfernung zu 2005 geschrieben. Daraus ist deutlich zu erkennen, daß die Sprungweite, die zum Programmzähler addiert wird, 05 sein muß, wenn der Sprung zum RTS erfolgen soll. Für Vorwärts-Verzweigungen gilt also: Von der Adresse des Befehls an, der auf den Branch-Befehl folgt, zählt man die Byte-Anzahl bis zum Sprungziel. Das Ergebnis ist der Offset.

Nun gibt es genauso häufig Rückwärts-Sprünge. In den bisher gezeigten Programmen sind sie mehrmals aufgetreten. Wie berechnet man den Offset in diesen Fällen? Sehen wir uns wieder das Disassembler-Listing eines solchen Programmsegmentes an:

```
1000    A2 00    LDX #00
1002    E8       INX
1003    D0FD     BNE 1002
1005    00       BRK
...
```

Dieses Programmchen tut nichts anderes, als das vorher auf Null gesetzte X-Register hochzuzählen, bis es über 255 läuft (dann tritt ja wieder 0 auf!). Solange der Inhalt des X-Registers ungleich Null ist, erfolgt ein Sprung zurück bis zur INX-Anweisung in Zeile 1002. Erst wenn die Null durch den Überlauf aufgetreten ist, endet das Programm mit einem BRK in Zeile 1005.

Wir wissen schon, daß der Programmzähler beim Verarbeiten des BNE-Befehls auf 1005 steht. Sehen wir uns auch dieses Programm Byte für Byte an:

Byte		Inhalt	Bedeutung
1000		A2	LDX #
1001		00	
1002	3	E8	INX
1003	2	D0	BNE
1004	1	FD	Offset
1005		00	BRK

Wieder ist neben der Bytenummer die Entfernung vom aktuellen Programmzählerstand angegeben. Wir müssen also vom Inhalt des Programmzählers 3 abziehen, um zum INX-Befehl in Byte 1002 zu gelangen. Das kennen wir aber schon aus den vergangenen Ausgaben: Wenn der Computer eine Zahl abzieht, dann addiert er das Zweierkomplement dieser Zahl. Hier soll nun 3 subtrahiert werden. Wir berechnen das Zweierkomplement:

3 = 0000 0011 (binär)

Das Einerkomplement davon ist:

1111 1100

Dann wird eine 1 addiert

1111 1101

Dies ist das Zweierkomplement. In hexadezimal ausgedrückt heißt diese Zahl $FD und ist unser Offset. Für Rückwärts-Verzweigungen gilt also: Von der auf die Branch-Anweisung folgenden Speicherstelle an zählt man die Bytes zurück bis zum Sprungziel. Das Zweierkomplement der sich dadurch ergebenden Byte-Anzahl ist der Offset.

Das sieht reichlich kompliziert aus, aber zum einen haben Sie ja einen ganz freundlichen Assembler und nur in seltenen Notfällen müssen Sie den Offset berechnen. Zum anderen gibt es noch eine Faustregel, mit der man sich das ganze vereinfa-

chen kann. Die soll durch folgendes Schema erläutert werden:

Byte	Inhalt	Offset
...		
1995		F9
1996		FA
1997		FB
1998		FC
1999		FD
2000	BNE	FE
2001	Offset	FF
2002	Programmzählerstand	
2003		01
2004		02
2005		03

Bei Vorwärtssprüngen ist ohnehin alles klar: Bei einem Sprung nach Adresse 2005 müßte man in vorliegendem Fall einen Offset von 03 eingeben. Bei Rückwärts-Verzweigungen zählt man einfach von $FF an rückwärts bis zur Zieladresse. Eine Verzweigung nach 1996 würde im vorliegenden Fall also einen Offset von $FA erfordern.

Eine Einschränkung der relativen Adressierung können Sie nun auch sofort verstehen, wenn Sie an Zweierkomplementzahlen denken: Der Offset belegt ein Byte. Die größte positive Zahl in einem Byte ist
0111 1111 = +127 = $7F
und die kleinste negative Zahl ist
1000 0000 = —128 = ($80)

Es sind keine größeren Vorwärts-Verzweigungen als um 127 Bytes möglich, weil in diesem Fall ein Offset größer als $7F, also mit einem Bit 7 gleich 1 nötig wäre, was aber wieder als negative Zweierkomplementzahl verstanden und einen Rückwärtssprung verursachen würde. Ähnliches gilt anders herum: Es ist kein weiterer Rücksprung als um 128 Bytes möglich, weil das im Offset zum gelöschten Bit 7 führen würde, also zu einem Offset kleiner als $80, was wiederum anstelle des Rücksprunges eine Vorwärts-Verzweigung herbeiführen würde.

Darauf sollte man achten beim Erstellen eines Assembler-Programmes, daß man nie weitere Rückwärtssprünge als um 128, beziehungsweise Vorwärtssprünge um 127 Bytes verlangt. Auch wenn man im Assembler gar nicht auf relative Adressierung Rücksicht nehmen muß, weil der Assembler sich mit den Absolutadressen begnügt, sollte man wissen, daß zum Beispiel folgende Zeile aufgrund dieser Einschränkung nicht möglich ist:
3000 BNE 1000

Die meisten Assembler reagieren auf solch eine Zeile mit einer Fehlermeldung oder so wie der SMON, der klammheimlich die Programmstartadresse statt 1000 einsetzt. Aber es ist doch ärgerlich, wenn man auf dem Papier ein Programm fertig hat und erst beim Eintippen feststellt, daß der Computer das so nicht haben will.

Zeropage-Adressierung

Weil wir nun gerade mit der Adressierung so schön in Schwung sind, stelle ich Ihnen noch eine andere vor: Die Adressierung der Zeropage. Was ist die Zeropage? Auf deutsch heißt das Nullseite. Am besten versteht man das, wenn man sich in Erinnerung ruft, wie Adressen in unserem Computer verwaltet werden. Da haben wir doch ein LSB (Least Significant Byte) und ein MSB (Most Significant Byte), zum Beispiel $1F 04 (mit 1F als MSB und 04 als LSB). Nun hat unser C 64 65535 Adressen von $0000 bis $FFFF. Bei den ersten 256 Adressen von $0000 bis $00FF ist das MSB $00. Man nennt so einen 256-Byte-Block eine Seite (engl. page). Weil hier für alle Adressen dieser ersten Seite des MSB Null ist heißt sie Nullseite = Zeropage. Messerscharf werden Sie schließen, daß man die Seite mit den MSBs $01 als erste Seite bezeichnet, die mit den MSBs $02 als 2. Seite und so weiter.

Wenn wir nun zum Beispiel den Akku mit dem Inhalt der Zeropage-Adresse $00FA laden wollen, dann könnten wir schreiben:
3000 LDA 00FA

Unser Mikroprozessor versteht uns aber auch, wenn wir nur schreiben:
3000 LDA FA

Das ist sie, die Zeropage-Adressierung. Anstelle eines 3-Byte-Befehls ist das jetzt ein 2-Byte-Befehl, was Speicherplatz und vor allem Rechenzeit einspart. Auf diese Weise kann man von den bisher kennengelernten Befehlen folgende adressieren:
LDA, LDX, LDY, STA, STX, STY, INC, DEC, ADC und SBC

Sie können sich merken, daß man (bis auf zwei Ausnahmen, die wir noch kennenlernen werden) alle absolut adressierbaren Befehle auch Zeropage-absolut anwenden kann. Genauere Angaben über die Codes, die Ausführungszeiten und die Beeinflussung der Flaggen (letztere ist identisch mit der absoluten Adressierung) entnehmen Sie bitte der angefügten Tabelle 1.

Zum Thema Geschwindigkeit: Wenn Sie die benötigten Taktzyklen von absolut und von 0-absolut adressierten Befehlen in den Tabellen miteinander vergleichen, werden Sie jeweils einen Unterschied von einem Zyklus feststellen. Das mag Ihnen läppisch vorkommen. Bedenken Sie aber, daß Sie sehr häufig Schleifen programmieren müssen, die mehrere 100 Mal durchlaufen werden, die vielleicht als oft zu verwendende Unterprogramme dienen... Sie werden bald feststellen, daß da schnell beachtliche Zeitunterschiede auftreten können: Für zeitkritische Programme ist die Verwendung der Zeropage-Adressierung dringend geboten.

Dieser Tatsache waren sich leider auch die Schöpfer unseres Betriebssystems und des Basic-Interpreters voll bewußt. Die Zeropage ist nahezu randvoll mit Speicherstellen, in denen sich beide Programmkomplexe tummeln. Fast jede Kernal- und Interpreter-Routine notiert sich irgendwelche Werte auf der Seite Null. Das macht es uns als Assembler-Programmierer nicht gerade leicht, die Zeropage-Adressierung zu verwenden, wenn wir außerdem den Interpreter oder das Betriebssystem benutzen wollen. Es kann geradezu katastrophale Folgen

Befehlswort	Adressierung	Byteanzahl	Code Hex	Code Dez	Dauer in Taktzyklen	Beeinflussung von Flaggen
LDA	0-Page, abs.	2	A5	165	3	N,Z
LDX	0-Page, abs.	2	A6	166	3	N,Z
LDY	0-Page, abs.	2	A4	164	3	N,Z
STA	0-Page, abs.	2	85	133	3	—
STX	0-Page, abs.	2	86	134	3	—
STY	0-Page, abs.	2	84	132	3	—
INC	0-Page, abs.	2	E6	230	5	N,Z
DEC	0-Page, abs.	2	C6	198	5	N,Z
ADC	0-Page, abs.	2	65	101	3	N,V,Z,C
SBC	0-Page, abs.	2	E5	229	3	N,V,Z,C
CMP	unmittelbar	2	C9	201	2	N,Z,C
	absolut	3	CD	205	4	
	0-Page, abs.	2	C5	197	3	
CPX	unmittelbar	2	E0	224	2	
	absolut	3	EC	236	4	
	0-Page, abs.	2	E4	228	3	
CPY	unmittelbar	2	C0	192	2	
	absolut	3	CC	204	4	
	0-Page, abs.	2	C4	196	3	

Tabelle 1: Kenndaten der neuen Befehle und Adressierungen

lsn $	msn	→ $ bin. binär	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111
0	0	000	NUL	DLE	SP	0	@	P		p
			NULL	DLE	SP	0	@	P	CHR$(96)	CHR$(112)
1	0	001	SOH	DC1	!	1	A	Q	a	q
			SOH	DC1	!	1	A	Q	CHR$(97)	CHR$(113)
2	0	010	STX	DC2	"	2	B	R	b	r
			STX	DC2	"	2	B	R	CHR$(98)	CHR$(114)
3	0	011	ETX	DC3	#	3	C	S	c	s
			ETX	DC3	#	3	C	S	CHR$(99)	CHR$(115)
4	0	100	EOT	DC4	$	4	D	T	d	t
			EOT	DC4	$	4	D	T	CHR$(100)	CHR$(116)
5	0	101	ENQ	NAK	%	5	E	U	e	u
			ENQ	NAK	%	5	E	U	CHR$(101)	CHR$(117)
6	0	110	ACK	SYN	&	6	F	V	f	v
			ACK	SYN	&	6	F	V	CHR$(102)	CHR$(118)
7	0	111	BEL	ETB	'	7	G	W	g	w
			BEL	ETB	'	7	G	W	CHR$(103)	CHR$(119)
8	1	000	BS	CAN	(	8	H	X	h	x
			BS	CAN	(	8	H	X	CHR$(104)	CHR$(120)
9	1	001	HT	EM	)	9	I	Y	i	y
			HT	EM	)	9	I	Y	CHR$(105)	CHR$(121)
A	1	010	LF	SUB	*	:	J	Z	j	z
			LF	SUB	*	:	J	Z	CHR$(106)	CHR$(122)
B	1	011	VT	ESC	+	;	K	[	k	{
			VT	ESC	+	;	K	[	CHR$(107)	CHR$(123)
C	1	100	FF	FS	,	<	L	\	l	¦
			FF	FS	,	<	L	£	CHR$(108)	CHR$(124)
D	1	101	CR	GS	—	=	M	]	m	}
			CR	GS	—	=	M	]	CHR$(109)	CHR$(125)
E	1	110	SO	RS	.	>	N	↑	n	~
			SO	RS	.	>	N	↑	CHR$(110)	CHR$(126)
F	1	111	SI	US	/	?	O	—	o	DEL
			SI	US	/	?	O	←	CHR$(111)	CHR$(127)

Bild 2: ASCII-Code (jeweils oben) und Commodore-ASCII-Code (jeweils unten) (msn = most significant nibble; lsn = least significant nibble)

haben, einige Zeropage-Adressen zu überschreiben. Andere werden ständig neu beschrieben durch das Betriebssystem oder den Interpreter, was unseren eigenen — vielleicht gerade in so einer Speicherzelle gelagerten — Zwischenwerten den Garaus machen würde. Man sollte sich also die ersten 256 Speicherstellen ganz genau ansehen, bevor man sie adressiert oder aber auf das Betriebssystem und den Basic-Interpreter verzichten. Ersteres erleichtern uns Tabellen der Speicherbelegung (zum Beispiel Babel, Krause, Dripke »Das Interface Age Systemhandbuch zum Commodore 64«, Interface Age Verlag, oder »Das Commodore 64 Buch, Band 4, Ein Leitfaden für Systemprogrammierer«, Markt und Technik Verlag) und auch die Serie von Dr. Helmut Hauck »Memory Map mit Wandervorschlägen«, die seit Ausgabe 11/84 erscheint.

Ohne Hemmungen nutzen dürfen wir nur die Speicherstellen (jedenfalls beim C 64) $02 und $FB bis $FE. Weil das doch recht mikrig ist, hat jeder Assembler-Programmierer spezielle Tips, welche Zellen er noch mit welchen Vorsichtsmaßnahmen benutzt. Wenn man bestimmte Routinen aus dem Betriebssystem oder dem Interpreter nicht aufruft, bleiben dazugehörige Zeropage-Adressen unbeeinflußt und sind dann für eigene Zwecke nutzbar. Manchmal ist es notwendig, den alten Zustand einer Adresse nach Beendigung eigener Programme wieder herzustellen, manchmal nicht. Interessant und viel beschrieben in allen möglichen Zeitschriften, Büchern etc. ist die Möglichkeit, die Notizen, die sich das Betriebssystem oder der Interpreter auf der Zeropage macht, zu verändern. Im Prinzip schreibt man damit kleine Teile dieser Großprogramme um oder variiert Tabellenteile davon. Wie schon Dr. Hauck in seiner Serie sagt, geschieht das im Rahmen der »Tricks« mit irgendwelchen POKEs mehr oder weniger blind, weshalb auch bevorzugt Abstürze des Computers dabei festzustellen sind.

FLAGGE	Akku X Y > DATEN	Akku X Y = DATEN	Akku X Y < DATEN
N	0	0	1
Z	0	1	0
C	1	1	0

Bild 1. Flaggen bei den Vergleichsbefehlen

Warum Abstürze? Na, stellen Sie sich mal ein von Ihnen geschriebenes Programm vor — zum Beispiel das aus der letzten Ausgabe, zur Berechnung der Summe einer arithmetischen Reihe — und POKEn Sie dann anstelle irgendeines Befehlscodes, der dorthin gehört, jetzt eine 0 (also ein BRK) hinein. Die Wirkung dürfte ähnlich sein. Wenn man allerdings die Funktion der betreffenden Speicherstelle genau kennt, lassen sich recht nützliche Änderungen hervorrufen, wie zum Beispiel die SchutzPOKEs für den Basic-Speicher durch Verändern der Adressen $33, $34, $37 und $38.

Wir werden im folgenden immer dann, wenn wir mit Zeropage-Adressierung arbeiten oder Routinen des Betriebssystems oder Interpreters untersuchen, spezielle Stellen der Nullseite kennenlernen.

Vorhin hatte ich noch angedeutet, daß man dann die Zeropage fast vollständig nutzen könne, wenn man auf den Basic-Interpreter und das Betriebssystem verzichtet. Das ist tatsächlich möglich. Nur wird man dann erstaunt feststellen, wieviel Arbeit uns die computerinterne Software abnimmt oder anders herum: Viele bislang selbstverständliche Dinge werden wir dann plötzlich selbst programmieren müssen, und das kann ein hartes Brot sein!

Als Beispiel für ein Programm, das nicht nur die Zeropageadressierung verwendet, sondern sogar selbst komplett in der Zeropage steht, werden wir uns die CHRGET-Routine ansehen. Eine Klasse von Befehlen, die dort angewendet wird, die Vergleichsbefehle, soll zuvor noch gezeigt werden.

Die Vergleichsbefehle: CMP, CPX, CPY

Vergleichen heißt in englischer Sprache »to compare«, woraus Sie unschwer erkennen können, woher die Bezeichnung CMP und die CPs in CPX beziehungsweise CPY kommen. Verglichen wird jeweils der Akku-Inhalt (bei CMP), der Inhalt des X- (bei CPX) oder des Y-Registers (bei CPY) mit Daten, die der Compare-Befehl adressiert. Einige Beispiele werden Ihnen das klarer machen:

CMP #FF

vergleicht den Akku-Inhalt mit der Zahl $FF. Hier liegt die unmittelbare Adressierung vor, die ebenso für CPX und CPY verwendbar ist. Außerdem ist das dann ein 2-Byte-Befehl.

CPX 3000

vergleicht den Inhalt des X-Registers mit dem Inhalt der Speicherstelle $3000. Die absolute Adressierung ist also auch anwendbar (natürlich auch für CMP und CPY). Der Compare-Befehl besteht so aus 3 Bytes.

CPY A8

vergleicht den Inhalt des Y-Registers mit dem Inhalt der Zeropage-Stelle $A8. Diese soeben frisch gelernte Zeropage-Adressierung ist bei allen drei Vergleichsbefehlen möglich und macht aus ihnen 2-Byte-Befehle.

Für CPX und CPY sind das alle Möglichkeiten der Adressierung. CMP erlaubt weitere, die wir noch kennenlernen werden. Nun interessiert uns natürlich noch, wie das Vergleichsergebnis zu erhalten ist! Bei diesen Befehlen geschieht merkwürdiges: Die Vergleichsdaten werden vom Inhalt des Akkus (beziehungsweise X- oder Y-Registers) abgezogen, aber: Weder wird dieser Inhalt noch werden die adressierten Daten verändert! Der Trick ist, daß drei Flaggen das Ergebnis anzeigen: Die Negativ-Flagge N, die Null-Flage Z und das Carry-Bit C. Diese Anzeige geschieht so:

1) Der Registerinhalt (Akku, X-, Y-Register) ist größer als die Vergleichsdaten:
Dann ist das Carry-Bit = 1, die N- und die Z-Flagge = 0.

2) Der Registerinhalt ist gleich den Vergleichsdaten:
Dann sind Carry- und Z-Flagge = 1, die N-Flagge = 0.

3) Der Registerinhalt ist kleiner als die Vergleichsdaten:
Die N-Flagge ist dann = 1, Carry- und Zero-Flagge sind 0.

Damit Sie die Übersicht behalten können, ist in Bild 1 das ganze als Schema gezeigt.

Sie werden sich vermutlich schon denken können, wie der Hase weiterläuft: Mit den Verzweigungsbefehlen prüfen wir die Flaggen und springen die gewünschten weiteren Programm-Routinen an.

Die Kombination der Compare-Befehle mit den Verzweigungsoperationen wird Ihnen im weiteren Verlauf dieses Kurses noch ganz geläufig werden. Ein Beispiel sehen Sie nachher ebenfalls in der CHRGET-Routine. Leider muß ich Sie immer noch etwas vertrösten, denn mit Verstand begreifen läßt sich diese Routine nur dann, wenn man etwas mehr über die Codierung von Zeichen weiß. Deswegen werden wir uns nun noch mit dem ASCII-Code und dem Commodore-ASCII herumschlagen.

Zeichencodierung mit dem ASCII- und dem Commodore-ASCII-Code

ASCII ist die Abkürzung von »American Standard Code for Information Interchange« und das heißt auf deutsch »amerikanischer Standard-Code zum Informations-Austausch«. Diese Zeichenverschlüsselungsart ist international als ISO-7-Bit-Code genormt, und es wäre wirklich nett, wenn alle sich daran halten würden. Tatsächlich aber finden wir zum Beispiel bei unserem C 64 eine Abart des Normcodes, den Commodore-ASCII-Code. Über die damit erzwungenen Umrechnungen können alle diejenigen Dramen erzählen, die zum erstenmal einen (Nicht-Commodore-)Drucker an ihr Gerät anschließen oder aber blauäugig in den Online-Betrieb mit anderen Computern eintreten wollten.

Sehen wir uns zunächst einmal den ASCII-Code an. Es handelt sich um einen 7-Bit-Code, das heißt 128 Zeichen können in nur 7 Bits untergebracht werden (0000 0000 bis 01111111). Das achte Bit dient bei manchen Operationen mit Computer-Peripherie als Paritäts-Bit. Bei dieser Gelegenheit soll auch gleich erklärt werden, was Parität in diesem Zusammenhang bedeutet. Wer-

NUL	Null	
SOH	Start of heading	Beginn des Kopfes
STX	Start of text	Textbeginn
ETX	End of text	Textende
EOT	End of transmission	Übertragungsende
ENQ	Inquiry	Anfrage
ACK	Acknowledge	Bestätigung
BEL	Bell	Klingel
BS	Backspace	Zurücksetzen
HT	Horizontal tabul.	Horizontaltabulator
LF	Line feed	Zeilenvorschub
VT	Vertical tabulator	Vertikaltabulator
FF	Form feed	Formatvorschub
CR	Carriage return	Wagenrücklauf/Zeilenwechsel
SO	Shift out	Rückschaltung
SI	Shift in	Dauerumschaltung
DLE	Data link escape	Datenverbindungsumschaltung
DC1-4	Device control	Gerätesteuerung
NAK	Negative acknowl.	Negativ-Bestätigung
SYN	Synchronous idle	Synchronisations-Leerlauf
ETB	End of transmission block	Ende des Übertragungsblockes
CAN	Cancel	Annullieren
EM	End of medium	Datenträgerende
SUB	Substitute	Ersetzen
ESC	Escape	Umschaltung
FS	File separator	Dateitrennzeichen
GS	Group separator	Gruppentrennzeichen
RS	Record separator	Satztrennzeichen
US	Unit separator	Einheiten-Trennz.
SP	Space	Leerzeichen
DEL	Delete	Löschzeichen

Bild 4. Die Bedeutung der Abkürzungen im ASCII-Code

den Daten übertragen, muß immer mit Übermittlungsfehlern gerechnet werden. Das Paritätsbit dient dazu festzustellen, ob ein Byte korrekt angekommen ist. Bei der sogenannten geraden Parität zählt man die Einser im Byte zusammen und setzt Bit 7 auf 1 wenn sich eine ungerade Zahl ergibt. Mit dem Paritätsbit haben wir dann eine gerade Zahl. Ist die Quersumme des Byte schon gerade, bleibt Bit 7 eine Null. Ebensogut kann man die ungerade Parität verwenden, indem dann Bit 7 so gewählt wird, daß sich immer eine ungerade Zahl ergibt. Welche Art der Parität zur Anwendung kommt, ist Vereinbarungssache. Nehmen wir mal an, es sei gerade Parität gefordert und ein Byte mit der Information 00010110 soll übermittelt werden. Die Quersumme ist 3, also ungerade. Das Paritätsbit muß auf 1 gesetzt werden. Wir senden das Byte 10010110. Der Empfänger überprüft zunächst auf gerade Parität und verwendet dann nur die Bits 0 bis 6. Doppelfehler, die mittels des Paritätsverfahrens nicht festgestellt werden können, sind sehr selten. Leider kann auf diese Weise nur bemerkt werden, daß ein Übertragungsfehler aufgetreten sein muß, aber nicht welcher. Die Information muß dann neu angefordert werden.

Sehen wir uns nun den Commodore-ASCII-Code an. Durch die Einbindung der Grafikzeichen brauchen wir mehr als die 128 Kombinationen. Commodore benutzt deswegen einen 8-Bit-Code. Mit dem Basic-Befehl CHR$(x) können Sie sich alle 256 Möglichkeiten ansehen. Erschwerend kommt aber noch hinzu, daß wir nicht nur einen Zeichensatz, sondern deren vier zur Verfügung haben, die durch den jeweiligen Schreibmodus ansprechbar sind (Klein-/Großschriftmodus, Großschriftmodus, beide Modi mit Reverse-ON oder OFF). Im Zeichen-ROM liegen insgesamt 512 Muster abrufbereit. Zu diesen kommen beim CHR$-Befehl noch eine ganze Reihe von Steuerzeichen hinzu... die Verwirrung ist perfekt! Wir wollen an dieser Stelle keine Entwirrung vornehmen, sondern wir durchschlagen den Gordischen Knoten, indem wir nur die ersten 128 Zeichen mit den ASCII-Zeichen vergleichen. In Bild 2 und 3 finden Sie unsere Gegenüberstellung.

Einige Kombinationen dienen als Steuer-Codes. (Die Bedeutung der dabei verwendeten Abkürzungen sehen Sie unten.

Nur ein Teil dieser Codes wird tatsächlich genutzt. Andere haben — je nach Gerät an das sie gesandt werden — unterschiedliche Bedeutungen. Denken Sie dabei nur mal an die verschiedenen Betriebssysteme des Commodore-Druckers 1526, wo man bei dem einen mit CHR$(1), bei dem anderen mit CHR$(14) den Breitschrift-Modus anschaltet. Innerhalb unseres Computers werden offensichtlich bestimmte Codes anders genutzt. Das sind:

Anstelle von	geschieht folgendes:
ENQ	Zeichen weiß
BS	Blockieren der Umschaltung Klein-/Großschrift
HT	Zulassen der obigen Umschalt.
DC1	Cursor abwärts
DC2	Reverse-Modus an
DC3	Cursor in HOME-Position
DC4	INST/DEL
FS	Zeichen rot
GS	Cursor rechts
RS	Zeichen grün
US	Zeichen blau

Der auffälligste Unterschied ist der, daß beim Commodore-ASCII anstelle der Kleinbuchstaben Grafikzeichen liegen. Sollte anstelle des Normalmodus der Klein-/Großschriftmodus eingeschaltet sein, findet man anstelle der Großbuchstaben die kleinen.

Jetzt haben wir alle nötigen Kenntnisse, um die CHRGET-Routine in unserem Computer zu verstehen.

Die CHRGET-Routine

Das Kürzel CHRGET kommt von »Get a character«, was bei uns heißt: »Hole ein Zeichen«. Es handelt sich um eine sehr häufig benutzte Routine unseres Basic-Interpreters, die — wie schon vorhin erwähnt — komplett in der Zeropage steht. Wenn Sie mit dem SMON mal nachsehen wollen, dann geben Sie den Befehl

D 0073 008B

ein. Sie haben dann die komplette Routine vor sich:

```
0073  E6  7A    INC 7A
0075  D0  02    BNE 0079
0077  E6  7B    INC 7B
0079  AD  2502  LDA 0225
007C  C9  3A    CMP #3A
007E  B0  0A    BCS 008A
```

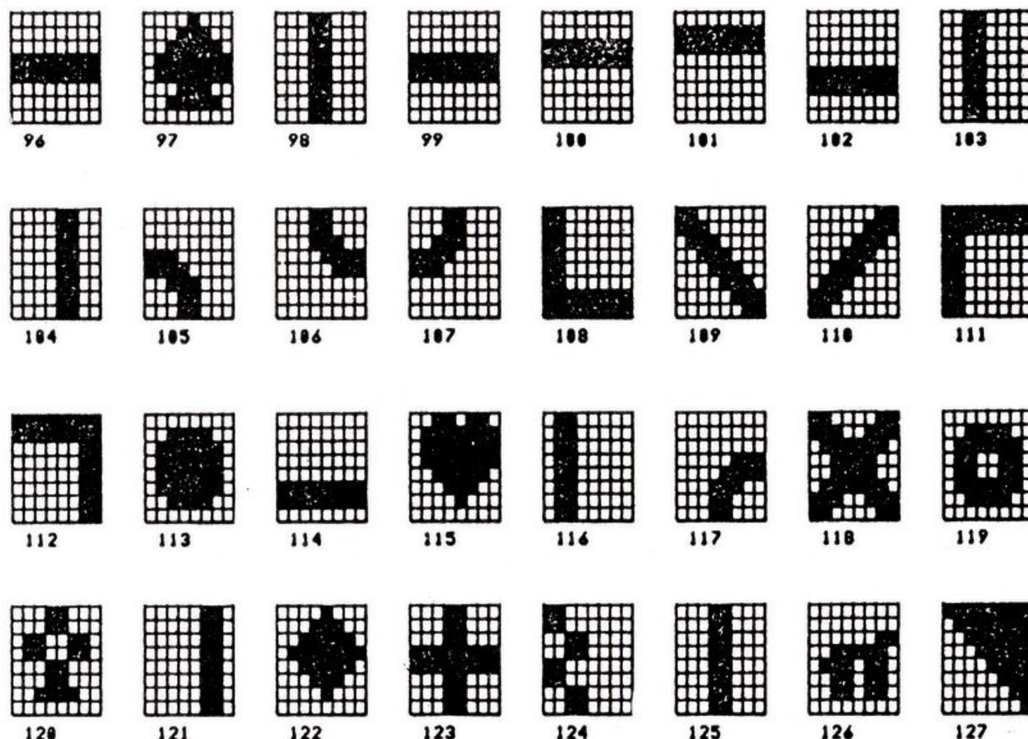

Bild 3. Die Grafikzeichen zu den entsprechenden CHR$-Codes

```
0080  C9  20      CMP #20
0082  F0  EF      BEQ 0073
0084  38  SEC
0085  E9  30      SBC #30
0087  38  SEC
0088  E9  D0      SBC #D0
008A  60  RTS
```

Eventuell sieht die Zeile 0079 bei Ihnen anders aus. Das liegt dann an den Speicherstellen 7A und 7B, welche einen Zeiger darstellen (LSB = 7A und MSB = 7B), der bei Ihnen gerade auf einen anderen Platz zeigt als auf $0225.

Diese CHRGET-Routine besteht aus drei Teilen:
Zeilen 0073 bis 0079
Weiterstellen des CHRGET-Zeigers und Einladen des dadurch angezeigten Speicherzelleninhaltes in den Akku.
Zeilen 007C bis 0082
Prüfroutinen
Zeilen 0084 bis 008A
Flaggen-Routinen

Im ersten Teil haben wir schon gleich etwas neues vor uns: Ein sich selbst veränderndes Programm. Die Speicherstelle (aus dem Basic-Eingabepuffer), aus der der Akku ein Zeichen holt, wird um 1 weitergezählt mit INC 7A.

Dabei handelt es sich um das LSB der Adresse und die nächste Zeile prüft, ob ein Überlauf (255 + 1) stattgefunden hat: BNE 0079.

Diese Technik kennen wir schon aus den letzten Folgen: Bei Überlauf wird die Z-Flagge auf 1 gesetzt und der BNE-Befehl führt keinen Sprung herbei, Den Offset von 02 können wir leicht nachrechnen: Der Programmzähler steht schon auf 0077. Die Zieladresse 0079 ist also noch 2 Bytes entfernt. Hat eine Überschreitung des Höchstwertes 255 stattgefunden, dann muß das dazugehörige MSB um 1 erhöht werden. Dies tut die nächste Zeile: INC 7B

In beiden Fällen ist nun der Zeiger 7A/7B um eine Stelle weitergerückt und der Inhalt der dadurch angezeigten Speicherstelle wird in den Akku geladen. Zwei Dinge können wir uns aus diesem kurzen Programmteil merken:
1) Wie man eine 16-Bit-Zahl hoch- (oder auch herunter-) zählt und
2) eine Möglichkeit, Zeiger einzusetzen. Wir werden noch eine Reihe anderer Zeigertypen kennenlernen und sehen, daß es nicht immer so direkt zugeht wie hier.

Im zweiten Teil finden wir die Prüfroutinen. Die Vergleichsbefehle beschränken sich auf den Akkuinhalt, also CMP. CMP #3A testet, in welcher Beziehung das im Akku befindliche Zeichen zum Wert $3A = dezimal 58 steht. Erinnern wir uns an das Schema in Bild 1:

1) Commodore-ASCII-Code im Akku größer als 58, also Zeichen hinter dem Doppelpunkt (Buchstaben, Grafikzeichen, einige Sonderzeichen). Dann ist die Carry-Flagge = 1, N- und Z-Flagge sind 0.
2) Im Akku steht genau der Code 58, also der Doppelpunkt. Dann sind Carry-Bit und Z-Flagge = 1, nur die N-Flagge = 0.
3) Der Code des Zeichens im Akku ist kleiner als 58 (das wären alle Zahlen, einige Sonderzeichen und Steuerzeichen). In diesem Fall ist die N-Flagge = 1. Die beiden anderen Flaggen zeigen Null.

Der nun folgende Befehl BCS 008A überprüft die Carry-Flagge. Wenn sie gesetzt ist, wenn also der Code im Akku größer oder gleich dem eines Doppelpunktes (58) ist, springt der Programmzähler zum RTS. Der Code (und auch die Flaggen) wird unverändert zum aufrufenden Hauptprogramm weitergegeben. Zur Übung können Sie ja nochmal den Offset nachrechnen. Der Rest des Programms wird nur noch durchlaufen, wenn Codes kleiner als 58 im Akku stehen.

Die nächste Zeile CMP #20 dient zum Vergleich des Space-Codes $20 = dezimal 32 (Leertaste). Die Flaggen treten dann, wie schon oben beim ersten Vergleich gezeigt, je nach Akku-Inhalt auf. Durch die Verzweigung BEQ 0073 erfolgt ein Rücksprung zum Beginn der CHRGET-Routine dann, wenn die Z-Flagge gesetzt ist, also ein Space-Code im Akku liegt. Somit werden die Leerzeichen einfach übersprungen und das nächste Zeichen geholt. Alle anderen Zeichen, die bis hierher durchgehalten haben, werden nun im letzten Teil der CHRGET-Routine einer Prozedur unterworfen, die ich Flaggen-Routine genannt habe.

Durch zwei aufeinanderfolgende Subtraktionen, die insgesamt den Wert im Akku unverändert lassen (es wird 256 abgezogen), wird die Carry-Flagge beeinflußt. Verfolgen wir, was da passiert:
SEC dient als Vorbereitung für die folgende Subtraktion.
SBC #30 zieht vom Akku-Inhalt $30 = dezimal 48 ab. Wir wissen inzwischen, daß das der Addition des Zweierkomplementes entspricht. Dieses ist (rechnen Sie mal nach!) 1101 0000.

Nehmen wir mal an, wir hätten den Code der Zahl 4 (also dezimal 52 oder $34) im Akku stehen. Die Rechnung sieht dann so aus:

52	0011	0100
	1101	0000
+		
(1)	0000	0100

Das Ergebnis ist also 4, der Übertrag wird vernachlässigt.

Als anderes Beispiel sei nun der Code für das Ausrufungszeichen im Akku (dezimal 33 = $21 = binär 0010 0001). Die Rechnung ist dann:

33	0010	0001
+	1101	0000
	1111	0001

Das Ergebnis ist —15.

Alle Codes, die nicht für Zahlen stehen, haben nach dieser Subtraktion ein negatives Ergebnis im Akku hinterlassen und durch das »Borgen« das Carry-Bit gelöscht.

Nun machen wir weiter ab Zeile 0087:
SEC
SBC #D0

Wir ziehen $D0 = dezimal 208 ab. Das Zweierkomplement ist: ...Doch da kommen wir ins Stocken! Denn dieses Zweierkomplement ist nicht mehr mit 8-Bit-Zahlen darzustellen. Schon die Zahl 208 im Binärformat (1101 0000) würde als negative Zahl angesehen werden, weil Bit 7 gleich 1 ist. Wir machen es uns einfach und sagen, daß sich das Zweierkomplement wie bisher bilden läßt, aber dabei das Carry-Bit mit einbezogen wird. Unser Zweierkomplement ist dann also: 0011 0000 und das Carry-Bit ist gelöscht. Nun nehmen wir unser erstes Beispiel. Dort war nach der Subtraktion im Akku eine 4 verblieben:

	0000	0100
+	0011	0000
	0011	0100

Das ist wieder unser ursprünglicher Wert dezimal 52 = $34 = Code für die Zahl 4. Das Carry-Bit bleibt gelöscht.

Im zweiten Beispiel mit dem Ausrufungszeichen stand noch im Akku eine —15:

	1111	0001
+	0011	0000
(1)	0010	0001

Da haben wir wieder den Code für das Ausrufungszeichen ($21 = dezimal 33) im Akku und ein gesetztes Carry-Bit. Was kommt also bei der CHRGET-Routine heraus?

1) Alle Zeichen außer dem Space werden unverändert an das aufrufende Programm über den Akku weitergegeben. Space wird unterdrückt.
2) Bei allen Zeichen außer bei den Zahlen ist das Carry-Bit gesetzt.
3) Manche der aufrufenden Routinen überprüfen außer dem Zustand der Carry-Flagge auch den der Z- oder N-Flagge, die ja beim ersten CMP-Befehl ebenfalls gesetzt werden. So liefert die CHRGET-Routine noch weitere Informationen.

In der einschlägigen Literatur stoßen Sie auch auf eine Routine, die CHRGOT genannt wird. Es handelt sich dabei ebenfalls um die hier beschriebene CHRGET-Routine, nur erfolgt der Einsprung nicht bei $0073, sondern bei $0079.

Der Zeiger $007A/7B wird in diesem Fall nicht weitergestellt. Das vorher schon einmal in den Akku geladene Zeichen wird damit noch einmal angesprochen (got ist die Vergangenheitsform von get).

Mit dem CHRGET-Programm haben wir eines der wichtigsten Unterprogramme unserer computerinternen Software kennengelernt. Will man sich Interpreter-Routinen zunutze machen, stolpert man ständig darüber. Außerdem aber liegt die CHRGET-Routine im RAM. Das bedeutet, daß wir sie ohne weiteres für unsere Zwecke verändern können.

Ein Beispiel für so eine Änderung hat Christoph Sauer in seiner Serie über den »gläsernen VC 20« in der Ausgabe 9 (Seite 158) gezeigt. Dort wird die CHRGET-Routine nach dem LDA angezapft und auf das Pi-Zeichen geprüft, das neuen Befehlen vorangestellt wurde. Sehen Sie sich das Programm dort (auf Seite 160f.) mal genau an, viel kann man durch Nachvollziehen fremder Programme für die eigene Programmiertechnik lernen. Wir werden im Verlauf dieser Serie noch andere Möglichkeiten behandeln, die CHRGET-Routine zu verändern.

Damit sei es für diesmal genug. Als Assembler-Alchimisten gehören Sie jetzt zu den fortgeschrittenen Eleven, denn Sie können immerhin schon so trickreiche Programme wie die CHRGET-Routine nachvollziehen.

(Heimo Ponnath/gk)