Von Basic zu Assembler

Teil 8

Wie kann man als Assembler-Programmierer mathematische Routinen des Interpreters nutzen, um beispielsweise den Sinus, die Quadratwurzel oder einfach den Absolutwert einer Zahl zu berechnen? Wir zeigen Ihnen, wie es gemacht wird.

Das Basic 2.0 des C 64 stellt nicht allzu viele mathematische Funktionen bereit. Mit der in der letzten Ausgabe vorgestellten Basic-Erweiterung, die wir nun weiter besprechen werden, erweitern wir den Basic-Interpreter um neun weitere Befehle.

In Teil 7 dieses Kurses erhielten wir ein Programm, mit dem das Basic 2.0 des C 64 durch zusätzliche mathematische Routinen erweitert wurde. Wir begannen, das ebenfalls abgedruckte Quell-Listing zu besprechen und wollen nun mit dieser Erklärung fortfahren. Beginnen wir gleich mit den im Listing 3 aus Ausgabe 10/86 definierten Label, die Interpreter-Einsprungadressen definieren. (All diese Interpreter-Routinen existieren einsatzbereit in unserem Computer. Eine Übersicht finden Sie beispielsweise im Assembler-Sonderheft 8/85 ab Seite 178). Sehen wir sie uns der Reihe nach an:

ERROR
Ausgabe von Fehlermeldungen und READY-Status
$A437 dezimal 42039
Fehlernummer in X-Register

Alle weiteren Angaben, die Sie bei den meisten anderen Routinen finden (welche Speicherstellen benutzt werden, welche Register und wieviele Plätze des Stapelregisters) sind hier ohne Bedeutung, weil ohnehin das Programm abgebrochen und der READY-Status aktiv wird. Die Fehlernummern und ihre Zuordnungen zeigt Ihnen die folgende Übersicht:

Fehler	Meldung
1	too many files
2	file open
3	file not open
4	file not found
5	device not present
6	not input file
7	not output file
8	missing filename
9	illegal device number
10	next without for
11	syntax
12	return without gosub
13	out of data
14	illegal quantity
15	overflow
16	out of memory
17	undef'd statement
18	bad subscript
19	redim'd array
20	division by zero
21	illegal direct
22	type mismatch
23	string too long
24	file data
25	formula too complex
26	can't continue
27	undef'd function

Es gibt noch zwei weitere Meldungen: »28 VERIFY« und »29 LOAD«, die auf diese Weise aufgerufen werden können.

Kernel-Routinen . . .

Die drei nächsten Adressen NEWSTT ($A7AE, 42926), GONE1 ($A7E4, 42980) und INTEND ($A7E7, 42983) sollen uns in diesem Zusammenhang noch nicht interessieren. Es handelt sich um verschiedene Stellen der Interpreterschleife, in die nach Bearbeitung einer eigenen Routine gesprungen wird oder die den normalen Inhalt eines Vektors bilden. Wir werden auf diese Interpreterschleife in einer späteren Folge noch ausführlich zurückkommen.

FRMNUM ist eine der wichtigsten Interpreter-Routinen, die zum Einlesen eines numerischen Wertes in den FAC dient:

FRMNUM
Holt beliebigen numerischen Ausdruck aus dem Basic-Text in den FAC und überprüft den Ausdruck.
$AD8A dezimal 44426
viele verschiedene Speicherstellen, darunter auch FAC
benötigt alle Register
Stapelbedarf verschieden, je nach Ausdruck

FRMNUM erledigt eine Vielzahl von nützlichen Aufgaben quasi nebenher: Die Speicherstelle $0D erhält den Inhalt 0, wenn ein numerischer Ausdruck, aber $FF, wenn ein Stringausdruck angesprochen wird. Im letzteren Fall erfolgt auch eine Fehlermeldung. In der Speicherstelle $0E wird angezeigt, ob man eine Fließkomma- oder eine Integerzahl geholt hat: Bei Fließkommazahlen findet man hier den Inhalt 0, bei Integer-Zahlen den Inhalt $80. Liegt eine einfache Variable vor, dann ist anschließend ein Zeiger in $45/$46 auf das erste Byte des Variablennamens gerichtet etc. Ich empfehle Ihnen, sich einmal ein ROM-Listing zu dieser Routine anzusehen. Sie werden noch einiges Brauchbare mehr entdecken.

CHKCOM
Prüft, ob das gerade gelesene Byte ein Komma ist und überliest dieses, oder Ausgabe einer Fehlermeldung, wenn kein Komma vorliegt.
$AEFD dezimal 44797
Speicherstellen TXTPTR (also $7A und $7B)
Register A und Y
kein Stapelbedarf

Man verwendet diese Routine CHKCOM, um eine gewisse Struktur in die selbstgeschaffenen Basic-Befehle zu bringen. Es kann sonst unter Umständen leicht geschehen, daß der Interpreter den Basic-Text falsch liest.

FACINX
Wandelt eine Fließkommazahl im FAC in eine 2-Byte-Integer-Zahl um, die dann im Akku und Y-Register steht.
$B1AA dezimal 45482
Vorbereitungen: Zahl in FAC
benötigt alle Register

Das entspricht der INT-Funktion des Basic, das LSB befindet sich im Akku, das MSB im Y-Register.

GETBYTC
Liest aus dem Basic-Text eine Zahl zwischen 0 und 255 ins X-Register ein.
$B79B dezimal 47003
Speicherstellen mehrere, unter anderem auch FAC
benötigt alle Register

Diese Routine GETBYTC bringt gegenüber FRMNUM nur den Vorteil einer gewissen Bequemlichkeit. Im Grunde genommen ruft sie nämlich zuerst einfach FRMNUM auf, dann FACINX und stellt noch eine Reihe von Überprüfungen an.

FSUB
Subtrahiert den FAC von einer durch Akku und Y-Register angezeigten Zahl im Speicher und legt das Ergebnis im FAC ab.
$B850 dezimal 47184
Vorbereitungen: A/Y als Vektor auf Zahl im Speicher richten, FAC laden
Speicherstellen: mehrere, unter anderem auch FAC und ARG
benötigt alle Register

FSUB lädt erst die durch A/Y angezeigte MFLPT-Zahl in den ARG, dreht dann das Vorzeichen des FAC-Inhaltes um und addiert beide.

Die nächste Adresse EINS ($B9BC, 47548) und auch die später auftretende PHIALB ($E2E0, 58080) ist jeweils die Startadresse einer Zahl, die schon fest im ROM im MFLPT-Format verankert vorliegt. Wie schon die Namen sagen, liegt bei EINS der MFLPT-Wert von 1 und bei PIHALB der von Pi/2. Solche Zahlenwerte gibt es einige im ROM. Eine Übersicht finden Sie im Assemblerkurs (Kapitel 46, Sonderheft 8/85 des 64'er-Magazins).

. . . und Interpreter-Routinen

LOG
Bildet den natürlichen Logarithmus des FAC-Inhaltes und legt diesen dann im FAC ab.
$B9EA dezimal 47594
Vorbereitungen: Zahl in FAC
Speicherstellen mehrere, unter anderem auch FAC und ARG
benötigt alle Register

Diese Routine LOG prüft auch die Korrektheit des Argumentes im FAC. Die nächste Routine dient der Multiplikation zweier Fließkommazahlen:

FMULT
Der FAC-Inhalt wird mit einer MFLPT-Zahl multipliziert, auf die der Zeiger A/Y weist und das Ergebnis im FAC abgelegt.
$BA28 dezimal 47656

Vorbereitungen: Faktor 1 in FAC laden, Zeiger A/Y auf Faktor 2 richten
Speicherstellen mehrere, unter anderen auch FAC und ARG
benötigt alle Register

FDIV

Eine MFLPT-Zahl, auf die der Zeiger A/Y weist, wird durch den Inhalt des FAC geteilt und das Ergebnis im FAC abgelegt.
$BB0F dezimal 47887
Vorbereitungen: Divisor in FAC, Zeiger A/Y auf Dividenden richten
Speicherstellen mehrere, unter anderem auch FAC und ARG
benötigt alle Register

Die Routine FDIV meldet auch einen Fehler, wenn der Dividend gleich Null ist. Zur Gruppe der Transportroutinen gehören die beiden folgenden:

MOVFM

Transportiert eine MFLPT-Zahl, auf die A/Y weist, aus dem Speicher in den FAC und wandelt sie dabei um in das FLPT-Format.
$BBA2 dezimal 48034
Vorbereitungen: Zeiger A/Y auf erstes Byte der Zahl im Speicher richten
Speicherstellen $22, $23, FAC
Register: A und Y
Den umgekehrten Weg öffnet diese Routine:

MOVMF

Transportiert den Inhalt des FAC in den Speicher an die Stelle, auf die X/Y weist und wandelt das FLPT- ins MFLPT-Format um.
$BBD4 dezimal 48084
Vorbereitungen: Zeiger X/Y auf erstes Byte im Speicher richten
Speicherstellen $22, $23, FAC
benötigt alle Register

Beide Routinen transportieren eigentlich nicht, sondern sie kopieren die entsprechenden Inhalte nur. Daraus folgt, daß die jeweilige Quelle unverändert erhalten bleibt.

Mathematische Routinen

ABS

Ermittelt den Absolutwert einer Zahl im FAC
$BC58 dezimal 48216
Vorbereitungen: Zahl in FAC
Speicherstellen FAC
keine Register

Ein sehr kurzes Programm, sehen Sie mal in ein ROM-Listing. Entspricht etwa der ABS-Funktion in Basic.

FCOMP

Vergleicht den FAC-Inhalt mit einer Zahl im Speicher auf die A/Y weist.
$BC5B dezimal 48219
Vorbereitungen: Zahl 1 in FAC, Zeiger A/Y auf Zahl 2 richten
Speicherstellen $24, $25, FAC
benötigt alle Register

Das Ergebnis des Vergleiches FCOMP wird im Akku angezeigt. Dabei ergeben sich folgende Aussagen:

Akku	Aussage
$01	FAC-Inhalt größer als Speicherzahl
$00	FAC-Inhalt gleich Speicherzahl
$FF	FAC-Inhalt kleiner als Speicherzahl

Nun kommen wieder einige Funktionen. Zunächst einmal das Bilden der Quadratwurzel des FAC-Inhaltes:

SQR

Die Quadratwurzel des FAC-Inhalts wird gebildet und im FAC abgelegt.
$BF71 dezimal 49009
Vorbereitungen: Zahl in FAC
Speicherstellen mehrere, darunter auch FAC und ARG
benötigt alle Register

POLYX

Berechnen eines Polynoms, Ergebnis im FAC.
$E059 dezimal 57433
Vorbereitungen: Argument in FAC, Zeiger A/Y auf Anfang einer Konstantentabelle richten
mehrere Speicherstellen
benötigt alle Register

POLYX wird von uns später beim POLY-Befehl verwendet. Die Konstantentabelle muß folgende Angaben enthalten:
1. Byte: Polynomgrad, weitere Bytes enthalten die Koeffizienten in absteigender Reihenfolge (also a_n, a_{n-1} ,...) als MFLPT-Zahlen.

COS

Bildet den Cosinus des FAC-Inhaltes und legt diesen im FAC ab.
$E264 dezimal 57956
Vorbereitungen: Zahl in FAC
benötigt alle Register

Diese Funktion COS entspricht der Cosinus-Funktion im Basic ebenso wie die folgende Funktion SIN der Sinus-Funktion des Basic entspricht:

SIN

Bildet den Sinus des FAC-Inhalts und legt diesen im FAC ab.
$E26B dezimal 57963
Vorbereitungen: Zahl in FAC
benötigt alle Register

Zu guter Letzt verwenden wir auch noch die Arcustangens-Funktion des Interpreters:

ATN

Bildet den Arcustangens des FAC-Inhaltes und legt diesen wieder im FAC ab.
$E30E dezimal 58126
Vorbereitungen: Zahl in FAC
benötigt alle Register

Wir werden später bei den einzelnen Modulen direkt mit all diesen Interpreter-Routinen arbeiten.

Das Programm im Modul 1

Ab Zeile 480 (im Listing 3 aus Ausgabe 10/86, Seite 156) fangen zwei kurze Programmteile an, die dem Ein- und Ausschalten der Befehlserweiterung dienen. Wie Sie sehen, speichern wir einfach in den Vektor IGONE die Startadresse unserer eigenen Interpreterschleife. Von da an wird jeder Befehl aus dem Basic-Text zuerst durch unsere Schleife und erst danach durch die normale Interpreterschleife überprüft. Das verlängert — allerdings unmerklich — die Abarbeitung eines Programms. Deshalb kann mit dem Programmstück ab Zeile 560 wieder der normale Inhalt des IGONE-Vektors restauriert werden. Wie Sie gleich sehen werden, passiert das immer dann, wenn AUS als neuer Basic-Befehl auftritt.

Ab Zeile 640 tritt nun unsere eigene Interpreterschleife auf den Plan. Unsere neuen Befehle sind als normale ASCII-Buchstaben im Basic-Text abgelegt. Deshalb wird ein durch die CHRGET-Routine in den Akku geholtes Zeichen zunächst überprüft, ob es sich um einen Buchstaben handelt. Falls das nicht der Fall ist, geben wir die Kontrolle wieder an den normalen Basic-Interpreter zurück. In Modul 10 haben wir eine Reihe von Tabellen und Hilfszellen geschaffen. Eine davon — nämlich AKKU — dient zur Zwischenspeicherung des Akku-Inhaltes, in eine andere — BEFNR — schreiben wir über das X-Register eine Befehlsnummer 0. Außerdem packen wir für die spätere Verwendung ins Y-Register den Wert 0 und erhöhen die Befehlsnummer auf 1. Des weiteren enthält das Modul 10 die Tabelle der Befehlstexte, aus der wir nun ein Zeichen in den Akku holen, überprüfen, ob wir ein Trennzeichen vor uns haben (nämlich eine Null). Nun wird verglichen (nämlich ab Zeile 850) und zwar Byte für Byte, ob der Text in der Befehlstabelle und der aus dem Basic-Text übereinstimmt. Sobald eine Ungleichheit festgestellt wird, überlesen wir schnell den Rest des Befehlswortes und überprüfen den nächsten Eintrag in der Tabelle. Mit jedem neuen Befehlswort wird auch der Inhalt in BEFNR erhöht.

Wenn der Basic-Text und der Text in der Befehlstabelle übereinstimmt, dann sorgen wir zuerst für das Erhöhen des TXTPTR, damit dieser Zeiger hinter unseren eigenen Befehl weist. Die Befehlsnummer in BEFNR wird verdoppelt und als Index in eine weitere Tabelle, die Spungtabelle SPRTAB in Modul 10 verwendet. In dieser Tabelle liegen nacheinander die Startadressen aller Programmteile, die zu den einzelnen Befehlen gehören. Die Verdoppelung von BEFNR wird einfach dadurch nötig, daß jede Adresse aus zwei Byte besteht. In Zeile 1070 unseres Moduls steht nun ein Sprungbefehl, dessen Adresse noch aus einem Dummy-Wert besteht. Das LSB der Adresse ist sprung+1, das MSB sprung+2. Die aus der Sprungtabelle SPRTAB gelesene Adresse schreiben wir nun anstelle des Dummy-Wertes: Das Programm modifiziert sich an dieser entscheidenden Stelle selbst. Sobald das geschehen ist, sind wir schon bei dem Sprung in das Unterprogramm (also in ein anderes Modul) angekommen. Danach — also nach der Abarbeitung des Befehls — begeben wir uns zurück in den normalen Basic-Interpreter. In Bild 1 aus Teil 7 (Ausgabe 10/86, Seite 153) finden Sie noch ein Flußdiagramm, das Ihnen zum besseren Überblick über das Modul 1 dienen soll.

Die neuen Befehle

Interessant wird es nun bei den einzelnen neuen Befehlen: Wir lernen dabei die Anwendung der mathematischen Interpreter-Routinen kennen. Sie werden sehen, daß alles einfacher ist, als man gemeinhin denkt. Um welche Befehle geht es? Zur Erinnerung: Zunächst machen wir es uns etwas einfacher, indem wir Bogen- in Gradmaß durch BOG und GRD auszurechnen. Auch stört es viele, daß man immer mit der LOG-Funktion den reichlich ungebräuchlichen, natürlichen Logarithmus erhält: DLGR liefert uns den dekadischen Logarithmus. Der Vollständigkeit halber ist den Winkelfunktionen SIN, COS und TAN nun auch noch der COT (Cotangens) hinzugefügt. Die eingangs erwähnte Umkehrfunktion des Sinus (ARCS) ist ebenso vorhanden wie die des Cosinus (ARCC) und des Cotangens (ACOT). Mit der ohnehin schon vorhandenen ATN-Funktion für den Arcustangens ist somit auch der Satz der Umkehrfunktionen komplett. Zu guter Letzt gibt es da noch die POLY-Funktion, mit deren Hilfe man mit einem einzigen Befehl den Wert eines Polynoms berechnen kann.

Modulaufbau

Jedes Programm-Modul besteht aus einem Rahmen und einem Kern (siehe dazu Bild 1).

Der Rahmen ist überall nahezu identisch und soll daher nur einmal erklärt werden. Damit jeder neue Befehl sowohl im Pro-

gramm-, als auch im Direktmodus betrieben werden kann und die Ergebnisausgabe in Variablen erfolgt, hatten wir eine etwas ungewöhnliche, aber einfach zu durchschauende Technik gewählt: Das Ergebnis landet immer in der zuletzt aufgerufenen Variablen. So erfolgt der BOG-Aufruf beispielsweise durch

```
A = A:BOG,45
```

Das Ergebnis steht dann in A, was durch »PRINT A« leicht zu kontrollieren ist. Auf die Startadresse eines Variablenwertes weist der Zeiger »FORPNT«. Um sicher zu gehen, daß er bei den manchmal reichlich unübersichtlichen Interpreter-Routinen nicht doch mal überschrieben wird, schieben wir seinen Inhalt auf den Stapel:

```
lda forpnt
pha
lda forpnt+1
pha
```

Es folgt der Aufruf einer Syntaxkontrolle:

```
jsr chkcom
```

CHKCOM überprüft, ob im Basic-Text ein Komma vorliegt. Ist das der Fall, wird es einfach überlesen. Andernfalls meldet sich der Computer mit SYNTAX ERROR und das Programm endet im READY-Zustand. Weshalb diese unnötige Kontrolle, werden Sie fragen! Zum einen wird ein Programm besser lesbar, wenn klar erkennbare Trennzeichen eingeplant sind. Das können durchaus auch andere als das Komma sein (mit der Komma-Abfrage gehts aber besonders leicht). Zum anderen werden Sie staunen, was ein Basic-Interpreter alles aus so einem selbstgemachten Befehl ohne Trennzeichen herauslesen kann! Probieren Sie es doch einfach mal aus.

Der letzte Befehl im oberen Teil unseres Rahmens ist

```
jsr frmnum
```

Damit lesen wir den Ausdruck hinter dem Komma in den Fließkomma-Akku 1 (den FAC) ein. Diese Routine nimmt nur numerische Ausdrücke an, bei Strings meldet sie einen Fehler und der Computer geht in den READY-Zustand. Die damit eröffnete Variationsbreite der Möglichkeiten von numerischen Ausdrükken ist ungeheuer vielfältig: Wir können Integer-Zahlen, Fließkommaausdrücke und Festkommazahlen einlesen, einfache Variable oder Array-Elemente, kompliziert zusammengesetzte Formeln oder Funktionen. Das Ergebnis liegt danach immer in leicht verarbeitbarer Form im FAC vor, und unserer Kreativität sind nur wenige Grenzen gesetzt.

Nach dieser Zeile folgt in allen Modulen der jeweilige Kern. Den Abschluß bildet danach der zweite Teil des Rahmens, der lediglich den zuvor auf dem Stapel gespeicherten Variablenzeiger zurückholt und den FAC-Inhalt in die dadurch bezeichnete Variable schreibt:

```
pla
tay
pla
tax
jsr movmf
rts
```

Wie Sie sicher wissen, funktioniert der Stapelspeicher nach dem LIFO-Prinzip: Das heißt »Last In - First Out« und bedeutet, daß der zuletzt daraufgelegte Wert als erster wieder herunter genommen wird (wie bei einem Bücherstapel). Als letzter Wert wurde im ersten Teil unseres Modulrahmens das MSB des Vektors FORPNT+1 abgelegt. Der kommt also nun wieder zurück in den Akku und von dort aus ins Y-Register. Das LSB lag darunter, wird als zweiter Wert vom Stapel geholt und auf dem Umweg über den Akku ins X-Register transportiert. Damit haben wir die Vorbereitungen schon erledigt, die die Interpreter-Routine MOVMF braucht: LSB der Zieladresse ins X-Register, MSB ins Y-Register. Mittels des MOVMF-Aufrufes transportieren (genaugenommen kopieren) wir nun den Inhalt des FAC in den dafür bereitgehaltenen Speicherbereich der durch FORPNT markierten Variable. Nun steht uns das Ergebnis vom Basic aus leicht zur Verfügung. Die Module werden durch RTS beendet, was den Rücksprung in das Modul 1, die Hauptschleife unserer Basic-Erweiterung, bewirkt.

Modul 2: BOG

Wir werden nun immer nur noch die Modulkerne besprechen. Da hatten wir es zuerst also mit dem Kern von BOG zu tun. BOG soll aus einem Winkel im Gradmaß das Bogenmaß berechnen. Nach dem Aktivieren der Erweiterung kann das Bogenmaß von 60 Grad im Direktmodus beispielsweise durch:

```
A=A:BOG,60:PRINT A
```

ermittelt werden. Unser Modul folgt der Umrechnungsformel:

```
Bogenmaß = Gradmaßx(π/180)
```

Der Faktor π/180 wurde BOGFAK genannt und sein Wert (0,0174532925) als Fließkommazahl in einer Konstantentabelle ab der BOGFAK genannten Speicherstelle abgelegt. Weil sich im FAC schon der Winkel im Gradmaß befindet, brauchen wir nur noch die Interpreter-Routine FMULT aufrufen, nachdem wir zuvor noch die Startadresse von BOGFAK in den Akku (LSB) und das Y-Register (MSB) geschrieben haben. Das Ergebnis der Multiplikation befindet sich im FAC. In Bild 1 finden Sie diesen Kern als Flußdiagramm.

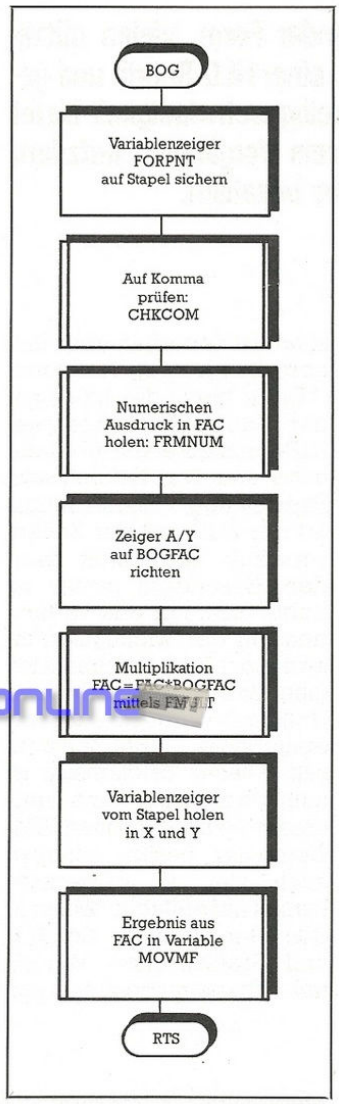

Bild 1. Wir erweitern den Basic-Wortschatz: Allgemeine Struktur der Programm-Module am Beispiel der neuen Befehle BOG, GRD und DLGR

Modul 3: GRD

Für die Umrechnung vom Bogen- in das Gradmaß gelten die gleichen Erläuterungen wie eben bei BOG: GRD leistet uns diesen Dienst nach der Gleichung:

```
Gradmaß = Bogenmaßx(180/π)
```

Auch diesen Faktor (er heißt GRDFAK und hat den Wert 57.2957795) finden Sie im Fließkommaformat in der Konstantentabelle ab GRDFAK. Außer dem anderen Faktor unterscheidet sich das Modul 3 nicht vom Modul 2.

Modul 4: DLGR

Spätestens an dieser Stelle werden Sie sich fragen, weshalb nicht der wesentlich einprägsamere Befehlsname DLOG gewählt wurde. Da spielt uns der Interpreter wieder einen Streich: Beim Eintippen von DLOG und <RETURN> am Ende der Zeile würde er nämlich das LOG als normales Basic-Befehlswort deuten und statt der ASCII-Zeichenkette LOG ein sogenanntes Token in den Programmtext einbauen, also eine Kennzahl, die der Computer beim Programmlauf später der Logarithmusfunktion zuordnet. Natürlich kann man als gewiefter Assembler-Programmierer auch diesem unerwünschten Interpreterverhalten einen Riegel vorschieben, aber dann wird es für diesen Kurs ein wenig unübersichtlich. Deshalb heben wir uns solche Feinheiten für eine spätere Folge auf und begnügen uns damit, Befehlsworte zu verwenden, die keine normalen Basic-Worte in sich enthalten.

Den dekadischen Logarithmus (log, also den Logarithmus zur Basis 10) berechnet man aus dem natürlichen (ln, also dem zur Basis e — dabei ist e die Eulersche Zahl 2,718281828459...) durch die Gleichung:

```
log (x) = ln (x) * (1/ln10)
```

Wieder haben wir einen Faktor vorliegen (nämlich 1/ln10, LOGFAK mit dem Wert 0,434294482), der als Fließkommazahl in der Konstantentabelle ab LOGFAK abgelegt ist. Diesmal wird aber nicht direkt der Wert im FAC mit der Konstanten multipliziert, sondern zuvor muß vom FAC-Inhalt noch der natürliche Logarithmus gebildet werden. Dazu verwenden wir — einfach durch den Routineaufruf mittels JSR — die Interpreterroutine LOG. Automatisch legt diese Routine den ln des eingegebenen Wertes im FAC ab, wo wir damit dann genauso weiterverfahren wie bisher: Akku und Y-Register auf LOGFAK richten und FMULT aufrufen.
Wie bentzt man DLGR? Hier ein Beispiel:

```
A=A:DLGR,Ausdruck:PRINT A
```

In A steht dann der dekadische Logarithmus von »Ausdruck«, der hier auf dem Bildschirm angezeigt wird.

In der nächsten Folge werden wir die noch fehlenden Module besprechen. Damit haben Sie dann eine leistungsfähige Befehlserweiterung vorliegen, die mathematische Funktionen unterstützt.

(Heimo Ponnath/dm)