

**Ulrich Krinzner**

## **OPL-Programmierung für Psion-Handhelds**

Serie 5/mx/mxPro/, Serie 7, netBook und Revo

### **- Teil 1: Grundwissen -**

Privatveröffentlichung, 1. Auflage, 04/2004

**Copyright © Ulrich Krinzner 2004**

**E-Mail: [krinzner@snaflu.de](mailto:krinzner@snaflu.de)**

Privatveröffentlichung, 1. Auflage, 04/2004

Dieses Buch wurde im Interesse einer noch immer großen Fan-Gemeinde produziert. Obwohl schon längst Rost und Motten die Produktionsstätten unserer Lieblinge zerfressen haben dürften, ist doch das Interesse an der leichten Programmierbarkeit dieser Geräte ungebrochen. In deutscher Sprache gedruckte Werke zum Thema sind selten. Das vorliegende Buch füllt diese Lücke für Geräte mit den Betriebssystemen EPOC32 r3 und r5, zuletzt auch als Symbian-OS 5 bezeichnet. Es wendet sich insbesondere an Einsteiger, die mit dem nötigen Basiswissen versorgt werden. Der zweite Teil enthält die umfangreiche Beschreibung des gesamten Befehlsatzes.

Im Sinne möglichst geringer Kosten für Interessenten, wurde mehr Wert auf den Inhalt als auf raffinierte Buchdruckerkunst gelegt.

Das vorliegende Werk ist in allen seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten. Teilweise oder vollständige Reproduktionen – in welcher Form auch immer-, jedwede Veröffentlichung, Übersetzung sowie Speicherung in elektronischen Medien ist nur mit ausdrücklicher schriftlicher Genehmigung des Autors zulässig.

Die Warenzeichen, Gebrauchsnamen, Handelsnamen usw. aller erwähnten Erzeugnisse und Firmen werden ausdrücklich als solche anerkannt. Ihre Verwendung dient ausschließlich der Information des Lesers und keiner kommerziellen Absicht.

Hier vorgestellte Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind lediglich für Lehrzwecke bestimmt.

Das vorliegende Material wurde unter großer Sorgfalt zusammengestellt, trotzdem ist Fehlerfreiheit nicht zu gewährleisten. Der Autor wird für solche möglichen Fehler und deren Folgen weder juristische Verantwortung noch sonstige Haftungen übernehmen

# Inhaltsverzeichnis

---

<b>INHALTSVERZEICHNIS .....</b>	<b>2</b>	TEXTAUSGABE .....	37
<b>1 EINFÜHRUNG .....</b>	<b>4</b>	ZAHLENEINGABE .....	38
WAS IST OPL? .....	4	DATUMS- UND ZEITEINGABEN .....	39
WAS MAN BRAUCHT .....	5	STRINGEINGABEN .....	40
DAS ERSTE MAL .....	6	AUS EINER LISTE WÄHLEN .....	40
DER EDITOR IST MEHR ALS EINE TEXTEINGABE .....	7	AUSWAHL PER CHECKBOX .....	41
<b>2 VARIABLEN .....</b>	<b>8</b>	NUTZERFREUNDLICHE DIALOGE .....	41
WORUM GEHT'S? .....	8	ALARM! .....	42
NAMENS GEBUNG .....	8	<b>9 EINFACHE GRAFIK .....</b>	<b>43</b>
NUMERISCHE VARIABLEN .....	9	CURSOR POSITIONIEREN .....	43
ARRAYS .....	10	GEOMETRISCHE ELEMENTE .....	44
STRING-VARIABLEN .....	10	ELEMENTE FÜLLEN .....	45
VARIABLEN DEKLARIEREN .....	11	VERSCHIEBEN, KOPIEREN, ENTFERNEN .....	46
MIT NUMERISCHEN VARIABLEN ARBEITEN .....	13	SCHNELLERE GRAFIK .....	47
MIT STRINGVARIABLEN UMGEHEN .....	13	FARBEN .....	48
<b>3 EINFACHE DATENVERARBEITUNG .....</b>	<b>14</b>	<b>10 GRAFISCHER TEXT .....</b>	<b>50</b>
DATEN FÜR DAS PROGRAMM .....	14	TEXT – PIXELGENAU GESETZT .....	50
DATEN SICHTBAR MACHEN .....	15	<b>11 FENSTER UND RAHMEN .....</b>	<b>54</b>
<b>4 VERZWEIGUNGEN .....</b>	<b>18</b>	FENSTER ERZEUGEN UND MANIPULIEREN .....	54
ENTSCHEIDUNGEN TREFFEN .....	18	FENSTER-RAHMEN UND SCHATTEN .....	56
<b>5 SCHLEIFEN .....</b>	<b>22</b>	<b>12 BITMAPS .....</b>	<b>59</b>
WIEDERHOLUNGEN LEICHT GEMACHT .....	22	UNSICHTBARE FENSTER .....	59
<b>6 PROZEDUREN .....</b>	<b>26</b>	<b>13 DATENBANKEN .....</b>	<b>62</b>
EINFACHE PROZEDURAUFRUFE .....	26	SPARBÜCHSE FÜR DATEN .....	62
PARAMETERÜBERGABE .....	26	DATENBANKEN UND TABELLEN ANLEGEN .....	63
FUNKTIONEN .....	27	DATEN SPEICHERN .....	64
PRO & CONTRA GLOBAL .....	27	BESCHLEUNIGUNG DURCH TRANSAKTIONEN .....	65
Z.B.: LOTTO .....	28	SUCHEN ... UND FINDEN .....	66
AUFRUF PER @-OPERATOR .....	30	ÄNDERUNGEN .....	67
<b>7 MENÜS .....</b>	<b>32</b>	DATEIVERWALTUNG .....	68
DAS PRINZIP .....	32	<b>14 FEHLER .....</b>	<b>69</b>
KOMPLETTBEISPIEL .....	33	FEHLERARTEN .....	69
KASKADEN .....	34	<i>Syntax-Fehler</i> .....	69
SPIELREGELN .....	35	<i>Fehler zur Programmlaufzeit</i> .....	69
OPTIONEN .....	35	TRAP – FEHLER IGNORIEREN .....	69
POPUP-MENÜS .....	36	FEHLERANALYSE .....	70
<b>8 DIALOGE .....</b>	<b>37</b>	FEHLER BEHANDELN .....	70
DAS PRINZIP .....	37	RAISE – MEIN FEHLER! .....	71
		KORREKTER EINSATZ DER FEHLERBEHANDLUNG ..	72

<b>15 APPLIKATIONEN .....</b>	<b>73</b>	PACKAGE-DATEI ANLEGEN .....	78
DER WEG NACH OBEN .....	73	SIS-DATEI ERZEUGEN .....	79
FORMALITÄTEN .....	73	POLITUR .....	79
EINZELHEITEN .....	74	<b>17 AUSSICHTEN .....</b>	<b>80</b>
CAPTION .....	74	INPUT-/OUTPUT- (I/O-) BEFEHLE .....	80
ICONS .....	75	ASYNCHRONES ARBEITEN .....	80
FLAGS .....	77	GEHOBENE PROGRAMMENTWICKLUNG .....	80
<b>16 SIS-DATEIEN ERSTELLEN .....</b>	<b>78</b>	OPX-MODULE .....	81
WIE MAN SIS-DATEIEN ERSTELLT .....	78	<b>18 INDEX .....</b>	<b>82</b>

# 1 Einführung

---

## Was ist OPL?

Die Konstruktionsprinzipien von Handheld-Computern ähneln denen von Desktop-Computern. So benötigen sie in der Grundausstattung einen Prozessor, Speicher für das Betriebssystem (zumeist ROM, Read Only Memory) und die Arbeit (RAM, Random Access Memory), ein Anzeigemedium (den Bildschirm) und eine Eingabemöglichkeit (Tastatur). Die Tastatur benutzen wir, um den Computer von außen mit Daten zu füttern oder ihn zur Arbeit aufzufordern.

Ohne die anderen Komponenten geringschätzen zu wollen: die Hauptarbeit leistet der Prozessor. Was er machen soll, wird ihm in elementarer Weise in Form von Nullen und Einsen mitgeteilt. Wollen wir, die Programmierer, ihm diktieren, was er zu tun hat, müssen wir seine Sprache lernen.

Allerdings kommt der Mensch dabei recht schnell ins Schwitzen – Nullen und Einsen sind einfach nicht seine Sache. Folglich bediente er sich zunächst der prozessornahen Assemblersprache. Die ist ihm zwar etwas verständlicher, aber immer noch sehr kryptisch. Zudem ist der Lernaufwand immens und immer nur auf einen Prozessor-Typ bezogen. In Sonderfällen greift man jedoch selbst heute noch darauf zurück, denn in der Hand des erfahrenen Programmierers entsteht ein schneller und kurzer Code.

Am Dartmouth-College (USA) entschlossen sich Kurtz und Kemeney Mitte der 60er Jahre, Abhilfe zu schaffen. Sie entwickelten eine Programmiersprache, die dem Anwender erlaubt, sich unabhängig vom Prozessor allein auf den kreativen Teil des Programmierens zu konzentrieren. Das Übersetzen in die Sprache der Maschine überließen sie einem Hilfsprogramm – dem Interpreter. So entstand die Programmier-Hochsprache BASIC (Beginners All Purpose Instruction Code). BASIC-Programme können mit einem ganz normalen Texteditor geschrieben werden. Das entstehende Produkt nennt sich "Programm" oder auch "Quelltext". Der Interpreter nimmt sich den Text dann Zeile für Zeile vor, übersetzt ihn passend und füttert den Prozessor damit – das Programm läuft.

Im Moment des Programmlaufs müssen sich also Programmtext und Interpreter zugleich im Speicher befinden. Das bedeutet Speicherplatzverbrauch. Andere Hochsprachen gehen daher einen anderen Weg. Sie übersetzen den Quelltext einmalig gleich in die prozessorgerechte Sprache. Als Werkzeug benutzen Sie dazu einen "Compiler". Das entstehende Programm ist sofort und eigenständig lauffähig.

Beide Verfahren haben Vor- und Nachteile. Es ist aber völlig müßig, darüber zu referieren, da wir ja definitiv mit OPL arbeiten wollen.

OPL entpuppt sich als eine BASIC-Variante, die natürlich auch dessen Grundzüge trägt, aber darüber hinaus dem Psion angepasste spezifische Befehle und Funktionen enthält. Außerdem benutzt OPL einen "Trick", um dem ursprünglich etwas lahmen BASIC auf die Sprünge zu helfen. So wird der Quelltext zunächst in einen Zwischencode übersetzt. Solche übersetzten Dateien erhalten die Dateiendung ".OPO".

OPO-Dateien sind kompakter als der Quelltext und werden durch den Interpreter, der in jeden Psion bereits eingebaut ist, nur noch gescannt und zur Ausführung gebracht. Der Geschwindigkeitszuwachs sorgt dafür, dass Programme realisierbar werden, die mit dem Ursprungs-BASIC nicht mehr machbar wären – ich denke beispielsweise an den Bereich Grafik. Die Erweiterung um "OPX-Module" tut ein übriges, um den Programmierer in Komfort schwelgen zu lassen, ohne dass er eine komplizierte Sprache lernen muss. OPX-Module sind vorgefertigte, in schnellem C programmierte, leistungsfähige Programm-Stückchen, die von OPL aus aufgerufen werden können.

OPL ist wegen seines Urvaters BASIC gerade für Anfänger ein mächtiges Werkzeug, mit dem sich – je nach Kenntnisstand – auch professionelle Programme erstellen lassen. Wer allerdings für seine Projekte noch mehr Geschwindigkeit braucht und kompliziertere Programmroutinen schreiben will (z.B. im Bereich der Datenübertragung), wird nach wie vor zu C- und C++-Programmierungsumgebungen greifen müssen.

## Was man braucht

Um mit OPL programmieren zu können, benötigt man neben dem eigentlichen Psion-Computer vor allem einen geeigneten Editor, mit dem das Programm eingegeben wird. Wer über irgendeinen Serie 5, einen Serie 7 oder das netBook verfügt, findet bereits von Haus aus eine passende Software auf seinem Gerät vor. Sie versteckt sich auf der Extras-Programmleiste unter dem Titel "Programm" auf deutschen oder "Program" auf englischen Psions.

Lediglich der Revo hat den Editor nicht im Gepäck. Man findet ihn aber seit Anfang 2001 kostenlos im Internet unter wechselnden Adressen. Die Datei ist um die 50 KB groß und wird wie jedes andere EPOC-Programm über die Systemsteuerung oder über den PC installiert. Es handelt sich dabei um eine englische Ausführung – eine andere gibt es derzeit nicht.

An Geräten mit großen Bildschirmen lässt es sich so schon recht ordentlich arbeiten. Noch komfortabler geht es, wenn man sich das OPL-SDK (Software Development Kit) besorgt. Das SDK wird nach Anweisung auf dem Windows-PC installiert und bietet u.a. einen EPOC32r5-Emulator (EPOC heißt das Betriebssystem des Psions, die 32 steht für 32bit-System und r5 kennzeichnet die Version/Release 5). Der Emulator bildet entweder einen englischen 5mx oder einen netBook-Verschnitt nach. Wer eine Revo-Oberfläche sehen will, braucht ein Zusatzmodul.

Herauszufinden, unter welcher Internetadresse sich das SDK gerade erhalten lässt, wäre eine Aufgabe für den Staatsanwalt, der einst den legendären Richard Kimble jagte. Auch Kimble hat seinen aktuellen Wohnsitz nie jemandem verraten wollen. Auf der Internetseite der Symbian-Entwickler, der einstigen Heimat des SDK, sucht man das gute Stück inzwischen auch vergebens. Die Politik, die Symbian in dieser Angelegenheit betreibt, ist dem "kleinen Entwickler" völlig unverständlich.

Immerhin, es gibt das SDK und auch die oben beschriebene Editor-Datei nach kostenloser Registrierung noch immer. Allerdings muss man oft etwas suchen, wie z.B. bei dieser Internetadresse:

<http://www.psionteklogix.com/de> (nach OPL SDK für "netBook" bzw. "netPad" fahnden)

Man kann das SDK also letztlich frei herunterladen, muss aber den dicken Brocken in Form einer rund 26 MB großen ZIP-Datei schlucken. Vielleicht kennen Sie ja jemanden mit Flatrate oder DSL-Zugang ... Bei Psion Teklogix besteht auch eine Kaufmöglichkeit. Man bekommt das SDK und eine Reihe anderer Dateien auf CD. Benutzen Sie die Kontaktmöglichkeiten unter der oben angegebene Adresse.

Der andere Haken: Möglicherweise ist das erforderliche Zusatzmodul "Revo" ist hier nicht zu haben, konnte ich aktuell nicht ermitteln ...

Sollte sich mit der Drucklegung die Adresse wieder geändert haben, fragen Sie am besten einmal beim Buchautor nach ... meist hat der eine Lösung für das Problem. Leider können aus rechtlichen Gründen die Dateien nicht zusammen mit dem Buch zur Verfügung gestellt werden.

Mit dem Emulator ist ein entspanntes Arbeiten an einer großen übersichtlichen Bildschirmfläche möglich. Bedient wird er wie eines der genannten Geräte – nur der Stift wird durch die Maus ersetzt. Und er verfügt natürlich über den OPL-Editor, was ihn am Ende erst für uns so richtig interessant macht. Aber Achtung: Ganz verlassen darf man sich nicht auf den Emulator – testen Sie auf alle Fälle die fertigen Programme immer auf dem Zielgerät!

Über die beschriebenen Gelegenheiten zum OPL-Programmieren hinaus gibt es weitere Möglichkeiten, die sich teils in dem OPL-SDK verstecken oder noch andere, die kostenpflichtig sind. In meiner Praxis hat sich die Arbeit mit dem Emulator am PC bestens bewährt, schon wegen der gewohnten großen Tastatur.

Für die Suche nach anderen Programmierungsumgebungen ist wie immer in Psion-Angelegenheiten die Website

<http://www.psionwelt.de>

ein guter Startpunkt.

## Das erste Mal

Wir wollen zusammen unser erstes Programm schreiben. Es ist üblich, im ersten Kontakt mit einer Programmiersprache ein Programm zu schreiben, das den Text "Hallo Welt" auf den Bildschirm bringt. Diese Tradition werden wir wahren ...

Starten Sie den OPL-Editor ("Programm") von der Extras-Leiste aus. Wenn Sie ihn noch nie verwendet haben, öffnet er gleichzeitig eine Datei mit Namen "Programm", ansonsten die zuletzt verwendete. Sollten Sie den Emulator oder den Revo mit englischem Editor benutzen, müssen Sie im folgenden immer die sinngemäßen englischen Menüs und Tastaturkürzel verwenden!

Speichern Sie die geöffnete Datei über das Menü: *Datei>Speichern unter*. Geben Sie ihr den Namen "Hallo Welt". Löschen Sie danach den gesamten Text der Datei (Menü: *Bearbeiten>Alles markieren* und <Entf>-Taste drücken) und geben sie den unten stehenden Programmtext ein. Der Editor funktioniert dabei so ähnlich wie die Applikation WORD.

```
PROC Hallo:
  PRINT "Hallo Welt"
  GET
ENDP
```

Das ist schon das Programm! Damit es laufen kann, müssen Sie es jetzt übersetzen lassen (Menü: *Extras>Übersetzen*). Die im Anschluss auftauchende Frage: "Programm ausführen?" beantworten Sie mit "Ja" (entsprechenden Button mit dem Stift drücken oder <J>-Taste drücken) und "Hallo Welt" erscheint auf dem Bildschirm. Durch den Druck auf eine beliebige Buchstabentaste endet das Programm und der Editor wird wieder aufgeblendet.

Wenn das Programm noch einmal laufen soll, wiederholen Sie Übersetzen und Starten oder Sie starten es direkt über das Menü: *Extras>Programm ausführen* oder das Tastaturkürzel <Strg><R>. Damit wird die übersetzte Form unseres Programmes ("Hallo Welt.opo") aus einer Liste abgerufen und gestartet.

Das kurze Programm zeigt, wie OPL-Programme prinzipiell aufgebaut sind, ohne nun gleich alle Feinheiten preiszugeben. Grundsätzlich beginnen sie mit *PROC Name:* und enden mit *ENDP*. *PROC* steht für den Begriff "Prozedur". Jede Prozedur bekommt einen einmaligen Namen, der mit einem Doppelpunkt abgeschlossen wird. Der Name darf maximal 32 Zeichen haben.

Zwischen *PROC Name:* und *ENDP* befinden sich die Befehle, die ausgeführt werden sollen. So bringt *PRINT* den dahinter stehenden Text auf den Bildschirm. *GET* wartet auf den nächsten Tastendruck, erst danach schreitet das Programm fort. Da es keine weiteren Anweisungen gibt, endet das Programm einfach.

Der Aufbau lässt vermuten, dass es in einem Programm mehr als eine Prozedur geben darf. Die Vermutung ist richtig! Beispiel:

```
PROC Hallo:
  Ausgabe:
  GET
ENDP

PROC Ausgabe:
  PRINT "Hallo Welt"
ENDP
```

Wenn Programme groß werden, trennt man sie in kleinere Einheiten auf, so bleibt die Übersicht erhalten. Oftmals können auch Programmstücke mehrfach verwendet werden – das ist ebenfalls ein guter und platzsparender Grund, getrennte Prozeduren ("Unterprogramme") zu schreiben.

Die Prozeduren werden einfach mit ihrem Namen aufgerufen, erledigen das, was in ihnen steht, und geben danach den Staffelstab wieder an die aufrufende Prozedur zurück.

In unserem Beispiel überlässt man die Textausgabe dem Unterprogramm *Ausgabe*:. Zu seinem Aufruf wird einfach sein Name einschließlich Doppelpunkt als Kommando in eine Zeile des Hauptprogramms *Hallo*: geschrieben. Ist das Unterprogramm mit seiner Arbeit fertig, setzt das Hauptprogramm seine Tätigkeit in der folgenden Zeile wie gehabt fort.

Wichtig zu wissen: Ein OPL-Programm beginnt die Abarbeitung der Befehle immer mit der ganz zu Anfang stehenden Prozedur, egal wie diese heißt. Es haben sich dafür Namen wie *Start*:, *Beginn*: oder das englische *Main*: eingebürgert.

## Der Editor ist mehr als eine Texteingabe

Wie gerade beschrieben, dient der Editor in erster Linie zur Texteingabe. Theoretisch wäre dazu auch die Applikation WORD geeignet. Unser Editor leistet aber weit mehr, denn er übernimmt zusätzliche Funktionen. So stößt er auch die Übersetzung des von uns geschriebenen Programmcodes in ein lauffähiges Programm an.

Der Name des lauffähigen Programmes bildet sich dabei automatisch aus dem von uns vergebenen Namen für die OPL-Datei und dem Anhängsel ".opo". Die OPO-Datei wird in demselben Ordner abgelegt, in dem sich auch die OPL-Datei befindet.

Obwohl OPO-Dateien aus dem Systembildschirm heraus durch Doppeltipp mit dem Stift oder durch die <Enter>-Taste gestartet werden können, ist die Ausführung auch aus dem Editor heraus möglich (Menü: *Extras>Programm ausführen*). Im übrigen läuft eine OPO-Datei auch dann noch, wenn sie in einen anderen Ordner verfrachtet wird.

Sofern man beim Schreiben des Programms keinen Fehler gemacht hat, bemerkt man eine weitere wichtige Funktion überhaupt nicht: Während des Übersetzens wird nämlich zugleich auch immer die Syntax des Programmtextes überprüft, also, ob die Regeln von OPL der Form nach eingehalten wurden.

Stolpert der Übersetzer über einen Fehler, meldet er ihn und stellt den Cursor an die Fehlerstelle im Code. Nach erfolgreicher Fehlerbeseitigung startet man die Übersetzung erneut. Erst wenn alle Fehler beseitigt sind, wird die Übersetzung korrekt beendet und die lauffähige OPO-Datei erstellt.

Die Syntax-Prüfung bedeutet leider nicht, dass das Programm auch fehlerfrei läuft – für die logischen Fehler sind Editor & Co. leider nicht zuständig, das liegt ganz in unserem Geschick.

## 2 Variablen

---

### Worum geht's?

Vielleicht zu Ihrem Trost sei vorangestellt: Dies ist eines der Kapitel, die ich früher selbst nur zu gern überblättert habe. Später, beim Experimentieren mit anderen Programmiersprachen, war es eines der ersten, die ich aufschlug ...

Worum geht's? Man muss sicherlich kein Universitätsstudium abgeschlossen haben, um sich an die "x" und "y" oder "a", "b" und "c" genannten Variablen mathematischer Funktionen zu erinnern – guter alter Pythagoras ... All diese Buchstaben sind gewissermaßen Behälter – jeder Behälter hat Platz für eine einzige Zahl. Zu jedem Zeitpunkt nehmen die Zahlen in diesen Behältern einen konkreten Wert an, doch schon der nächste Rechengang kann diesen Wert verändern. Eben genau das verleiht dem Zahlenbehälter den Namen "Variable". Um genau zu sein: In diesem Fall hier reden wir von Zahlen- oder numerischen Variablen.

OPL benutzt genau solche Variablen und kennt darüber hinaus auch noch eine Text-Variable, Synonyme dafür sind auch: Zeichenketten- oder String-Variable. Diese Sorte nimmt einzelne oder mehrere Zeichen auf. Zeichen sind: Buchstaben, Ziffern und Sonderzeichen. Im Unterschied zu numerischen Variablen kann man mit ihnen nicht rechnen, aber manipulieren lassen auch sie sich.

### Namensgebung

Alle Variablen besitzen einen eigenen Namen, mit dem man sie anspricht oder mit dem man sie in Formeln und Funktionen einsetzt.

OPL erlaubt Variablennamen, die aus bis zu 32 Zeichen bestehen dürfen. So kann man bequem prägnante und sinnige Namen vergeben, z.B. "lohn" oder "abstand" usw. Die Einschränkung: Es dürfen keine vorbelegten OPL-Kommandos oder Funktionsbegriffe benutzt werden.

Der Name darf gemischt Buchstaben, Ziffern und den Unterstrich enthalten, aber nur mit Buchstaben oder Unterstrich beginnen. Innerhalb eines Programmes verwende man keine gleichlautenden Namen. Über die Ausnahmen reden wir noch.

Den verschiedenen Variablentypen wird, mit einer Ausnahme, zur Kennzeichnung ein spezifisches Suffix angehängt. Hier die Übersicht – Einzelheiten im nachfolgenden Text.

Bezeichnung	Suffix	Deklaration	Wertebereich
<b>Zahlen:</b>			
Integer	%	name%	-32.768 bis +32.768
Long Integer	&	name&	-2.147.483.648 bis +2.147.483.648
Fließkomma	ohne	name	2,2250738585072015E-308 bis 1,7976931348623157E+308 und 0 EPOC16: +/- 9,999999999999E99
<b>Zeichen:</b>			
Text	\$	na-me\$(zahl%)	Zeichen beliebiger Folge



Mit `zahl%` wird die Anzahl der Zeichen angegeben, welche in der Variablen maximal gespeichert wird. Der Bereich der Fließkommazahlen ist in der "wissenschaftlichen Darstellung" angegeben, wie sie OPL verwendet. In der Praxis entspricht z.B. die Angabe `2,0E+3` dann dem "richtigen" Wert  $2,0 \cdot 10^3$ . Wenn man Zahlen in diesem Format angeben will, muss immer auch das Vorzeichen des Exponenten mit angegeben werden.

## Numerische Variablen

Die numerischen Variablen teilt man noch einmal auf. Die sogenannten Gleitkomma- oder auch Fließkommazahlen werden überall dort verwendet, wo mit Kommastellen genau gerechnet werden muss, während sich die Integerzahlen (Ganzzahlen ohne Komma) gut für einfache Rechnungen und zum Zählen eignen. Die Variablentypen verbrauchen unterschiedlich viel Platz im Speicher und je mehr Speicherplatz sie verbrauchen, desto länger dauern auch Berechnungen mit ihnen. Man wählt den Typ also immer passend für die zu lösende Aufgabe aus. Die Integerzahlen sind deshalb auch noch einmal eingeteilt in normale (kurze) und lange (Long-) Integerzahlen. Die normalen Integers mit dem Suffix `"%"` im Namen haben ein Wertespektrum von -32768 bis +32767 und belegen 2 Bytes im Speicher, Long-Integers mit dem Suffix `"&"` gehen von -2147483648 bis +2147483647 und nehmen 4 Bytes im Speicher ein. Gleitkommazahlen fressen ganze 64 Bit und erfassen damit den Zahlenbereich von 2,2250738585072015E-308 bis 1,7976931348623157E+308. Unter EPOC 16, dem alten 16-Bit-Betriebssystem, stehen nur 8 Bits zur Verfügung, der Zahlenbereich ist damit auf +/- 9,999999999999E99 eingeschränkt.

Als Einsteiger habe ich die Mühe gehasst, jedesmal an die Integerzahlen diese Zusatzzeichen anhängen zu müssen. Heute bin ich geradezu vernarrt in diese Unterscheidungsmöglichkeit. Einige andere Programmiersprachen bieten diese Möglichkeit erst gar nicht und man kommt ziemlich schnell mit den Typen ins Schwitzen, erfindet man nicht selbst irgendeine passende Markierung. Wenn man also die Suffixe nicht als lästige Pflicht empfindet, kann man sich schon bald an der guten Überschaubarkeit erfreuen!

Die Zuweisung von Werten an die Variablen geschieht wie in der Mathematik:

```
anzahl%= 27
menge& = 79000
kosten = 20028.67
```

Man beachte, dass in den Programm-Quelltexten das Komma wegen der englischen Schreibweise durch einen Punkt zu ersetzen ist!

Gegenseitige Zuweisungen aus den verschiedenen Variablentypen sind möglich, sofern der Gültigkeitsbereich der "empfangenden" Variable das gestattet. Dabei finden Typ-Umwandlungen des Inhaltes statt:

```
menge&= 79000
kosten= menge& (Ergebnis: kosten= 79000.0)
kosten= 20028.67
menge&= kosten (Ergebnis: menge&= 20028)
```

Im letzten Fall wird aus der Gleitkommazahl eine Integerzahl durch Runden in Richtung der Null. Mit der negativen Ganzzahl -20028.67 wäre das Ergebnis also -20028.

Eine Besonderheit sind Zuweisungen bei Rechnungen, die dieselbe Variable im Rechenprozess als auch auf der Ergebnisseite verwenden. Hier wird besonders deutlich, dass das Gleichheitszeichen eben nicht rein mathematisch, sondern als Zuweisungszeichen verwendet wird.

Mathematisch würde  $a = a + 1$  keinen rechten Sinn ergeben. In OPL bedeutet es, dass nach der Addition von eins zum Ursprungswert von "a" das Ergebnis wieder in "a" zu stehen kommt. Diese Art der Anwendung kommt häufig in Zählschleifen zum Einsatz, kann aber auch in jeder anderen Berechnung angewendet werden.

Hält man sich an die Bereichsgrenzen der Variablentypen, ist das Rechnen mit ihnen nicht besonders aufregend. Um den Überblick zu behalten, sollte man Rechnungen nur innerhalb eines Variablentypes durchführen.

Sind Typveränderungen notwendig, wandelt man besser vor oder nach der Rechnung getrennt um, das vereinfacht auch eine eventuelle Fehlersuche.

Folgende arithmetische Operatoren stehen zur Verfügung:

Addition:	$a + b$
Subtraktion:	$a - b$
Multiplikation:	$a * b$
Division:	$a / b$
Prozent:	$100 + 10\%$ (100 plus 10 Prozent: 110) $100 - 10\%$ (100 minus 10 Prozent: 90) $100 * 10\%$ (10 Prozent von 100: 10) $10 / 10\%$ (10 ist 10 Prozent von welcher Zahl?: 100)
Potenzrechnung:	$a ** b$

## Arrays

Arrays sind Variablen, die mehrere Werte des gleichen Typs speichern können. Die Identifizierung der Einzelwerte erfolgt über einen Index, der an der Variablen in Klammern vermerkt ist, Beispiel:

```
x% (1) = 17
x% (2) = 19
x% (3) = 22
```

Für Arrays wird synonym der Begriff "Variablenfeld" verwendet.

## String-Variablen

Zeichen-Variablen nehmen zum Unterschied von numerischen Variablen auch mehr als einen "Wert" auf, quasi eine Aneinanderreihung von Einzelwerten. Das erklärt den englischen Begriff "string". Als Zeichen gelten Buchstaben, Ziffern und die meisten Sonderzeichen. Schon von weitem sind Strings durch den Namenssuffix "\$" erkennbar. Die Zuordnung von Werten erfolgt genauso einfach wie bei den numerischen Variablen:

```
stadt$= "Berlin" oder land$="Deutschland"
```

Eine Besonderheit gibt es jedoch. Der Wert der Variablen wird in Anführungszeichen übergeben. Der Grund ist einsichtig: Verwechslungen mit anderen Variablentypen oder Prozedur-Namen müssen vermieden werden.

Will man Anführungsstriche einbetten, ist das etwas umständlich (" ""Hallo!"" "), besser verwendet man der Übersicht halber gleich die einfachen Anführungsstriche (" 'Hallo!' ").

Zuweisungen von Variable zu Variable sind ebenfalls gängige Praxis:

```
hauptstadt$= stadt$
```

Aufgrund der Eigenschaften, mehrere Zeichen enthalten zu können, lassen sich Strings auch mit Hilfe des "+"-Zeichens verknüpfen und zuweisen:

```
satz$= stadt$ + " ist die Hauptstadt von " + land$
```

Auch die folgende Zuweisungsart gilt nicht nur für die numerischen Variablen, sondern eben auch für Strings:

```
name$= "Willi"
name$= name$ + " Mustermann"
```

Genau wie numerische Variablen können Strings in Feldern angeordnet sein. Das Adressieren und Ansprechen der Feldwerte erfolgt ebenfalls über den Index:

```
name$(1)= "Willi"
name$(2)= "Walter"
name$(3)= "Wilfried"
```

## Variablen deklarieren

Bevor die beschriebenen Variablen das erste Mal benutzt werden können, müssen sie deklariert werden. Dadurch werden sie gewissermaßen dem Programm bekannt gemacht und später von ihm auch anerkannt.

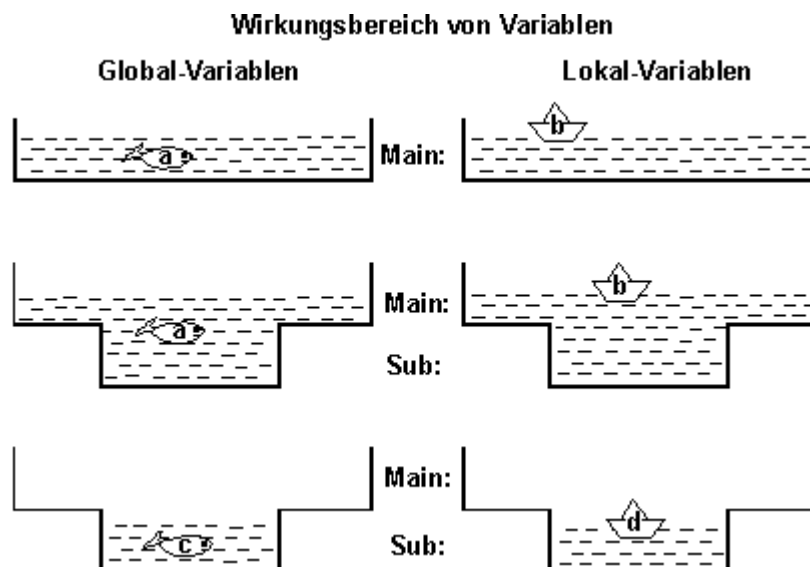
Neben den bereits beschriebenen "technischen" Variablentypen unterscheidet man zwei "verwaltungstechnische" Typen: globale und lokale Variablen.

Der Gültigkeitsbereich lokaler Variablen beschränkt sich jeweils auf die Prozedur, in der sie deklariert werden. Unterprogramme bzw. andere aufgerufene Prozeduren kennen diese Variablen nicht und können ihnen weder Werte zuweisen noch sonst irgendwie mit ihnen arbeiten. Im Gegensatz dazu ist das aber bei globalen Variablen möglich. Jede Prozedur, die von einer "über" ihr stehenden Prozedur aufgerufen wird, kann mit deren Globalvariablen umgehen und sie manipulieren.

Es ist wichtig, dass Sie den Unterschied genau kennen, ich erkläre das ganze deshalb noch einmal an einem Beispiel. Betrachten Sie einmal das folgende Programm und die zugehörige Abbildung. Programm und Unterprogramm enthalten nur die Deklarationen und machen sonst nichts weiter:

<pre>PROC Main:   GLOBAL a   LOCAL b   Sub:   ENDP</pre>	<pre>PROC Sub:   GLOBAL c   LOCAL d   ENDP</pre>
--	--

Stellen Sie sich einen Buddelkasten vor. Wenn Sie ihn mit Wasser füllen, können Sie darin Fische aussetzen und Papier-Schiffchen schwimmen lassen. Sofern Sie noch keine Löcher in den Sand gegraben haben, steht dieses Bild für ein Programm mit nur einer einzigen Prozedur. Die Schiffchen übernehmen die Rolle der lokalen, die Fische die der globalen Variablen. Innerhalb dieses Teiches ist es offenbar egal, ob man den Variableninhalt durch Fische oder Schiffe transportieren lässt, beide sind der Aufgabe gleichermaßen gewachsen.



Lassen Sie nun das Wasser ab, graben Sie ein Loch und fluten Sie das ganze wieder. Das Loch steht für das Unterprogramm *Sub*:. Wie Sie leicht erkennen, plätschert das Schiff nur an der Oberfläche herum, seine Ladung ist für den tiefen Graben unerreichbar. Oder anders: Die lokalen Variablen von *Main*: sind in *Sub*: unbekannt und lassen sich deshalb nicht benutzen, ihr Gültigkeitsbereich ist auf *Main*: beschränkt. Die Fische allerdings kommen mit ihren Fähigkeiten überall hin. Das bedeutet: Die in *Main*: deklarierte globale Variable ist auch in *Sub*: verwendbar – ihr Gültigkeitsbereich umfasst sowohl *Main*: als auch *Sub*:.

Richten Sie nun Ihr Augenmerk ausschließlich auf *Sub*: – senken Sie den Wasserspiegel auf dieses Niveau. Weder Fische noch Schiffe können je das höher liegende Niveau des Buddelkastens erreichen. Globale und lokale Variablen bleiben *Main*: verborgen und können dort nicht verarbeitet werden!

Offenbar scheint die Deklaration von globalen Variablen gleich am Anfang eines Programms eine optimale Lösung zu sein. Diese Vermutung ist nur bedingt richtig. Die Bequemlichkeit ist zugleich auch eine unangenehme Fehlerquelle, wenn Programme eine gewisse Größenordnung überschreiten. Lösungsansätze zu diesem Problem lernen wir später kennen. Nur noch soviel: Vermeiden Sie generell gleiche Variablennamen auf globaler und darunter liegender lokaler Ebene – innerhalb einer Prozedur werden doppelte Namen sowie so nicht akzeptiert.

Deklarationen müssen stets am Anfang einer Prozedur erfolgen. Die Erstwerte der Variablen werden automatisch stets auf Null gesetzt. Für Stringvariablen bedeutet diese "Initialisierung", dass sie "Nichts" enthalten (nicht einmal ein Leerzeichen) und damit einen sogenannten "Leer-" oder "Nullstring" bilden.

Einfache numerische Variablen werden so deklariert: Nach der Nennung des Variablentypes gibt man lediglich ihren Namen an. Das kann in einer kommagetrennten Reihe geschehen:

```
GLOBAL kosten, anzahl&,nummer%
```

Besonders bei größeren Programmen empfiehlt sich eine Einzeldeklaration, dadurch wird die Kommentierung des Verwendungszweckes erleichtert:

```
PROC
  REM es folgt die Deklaration
  GLOABL x%    REM x-Koordinate des Anfangspunktes
  GLOBAL y%    REM y-Koordinate des Anfangspunktes
  ...
ENDP
```

Der Kommentar wird mit *REM* (remark, Bemerkung) eingeleitet. Zwischen *REM* und dem eigentlichen Kommentar muss sich wenigstens ein Leerzeichen befinden. Steht *REM* hinter einem Befehl (und nicht allein in einer Zeile), muss auch hier vor dem *REM* wenigstens ein Leerzeichen stehen.

Stringvariablen gibt man zwecks Reservierung von Speicherplatz eine Längenangabe in Klammern mit, maximal ist ein Wert von 255 zulässig:

```
LOCAL name$(40)
```

Weitere Angaben werden bei Verwendung von Arrays benötigt, nämlich die Anzahl der Felder. Im folgenden Beispiel werden ein Integervariablenfeld mit 6 Feldern und ein Stringvariablenfeld mit ebenfalls 6 Feldern, von denen jedes 40 Zeichen aufnehmen kann, deklariert:

```
LOCAL nummer%(6), name$(6,40)
```

Bei der Stringvariablenfeld-Deklaration stolpert man immer wieder über die Reihenfolge, in der die Werte anzugeben sind. Am einfachsten merkt sie sich im Analogschluss: Denken Sie an numerische Felder, bei denen muss nur die Feldanzahl angegeben werden, der quasi "erste" Wert ist also die Feldanzahl. So ist es auch bei den Stringvariablenfeldern, dazu gesellt sich dann nur noch die Stringlänge ...

## Mit numerischen Variablen arbeiten

OPL enthält ein umfangreiches Reservoir an mathematischen Funktionen bereit. Sie sind im einzelnen im Teil 2 des Buches mit der alphabetischen Befehls- und Funktionsübersicht aufgeführt.

Neben allgemeinen Funktionen (EXP, LN, LOG, PI, SQR, RND, RANDOMIZE) finden sich statistische (MAX, MIN, MEAN, SUM, STD, VAR), trigonometrische (SIN, COS, TAN, ASIN, ACOS, ATAN, RAD, DEG) und Funktionen zur Typ- und Artwandlung (ABS, IABS, INT, FLT, HEX\$, FIX\$, GEN\$, NUM\$, SCI\$).

Die Anwendung ist weitgehend identisch: Der Funktion wird eine Variable als Argument übergeben und das Ergebnis in einer anderen Variablen wieder aufgenommen:

```
lognat=LN(x)
```

Einige wenige Funktionen kommen völlig ohne Argument aus und liefern eigenständig einen Wert zurück:

```
zahl_pi=PI
```

Wieder andere sind auf mehr als ein Argument angewiesen:

```
maxWert= MAX(x1,x2,x3, .. xn)
```

## Mit Stringvariablen umgehen

Während numerische Variablen eine weitgehend vertraute Materie sind, hat man im Alltag mit Stringvariablen wenig zu tun. Den Einsteiger mögen daher die vielen String-Verarbeitungsmöglichkeiten erstaunen. Ausführliche Beschreibungen finden Sie in der alphabetischen Befehlsübersicht.

Eine der wichtigen Informationen über einen String ist sicherlich seine aktuelle Länge (*LEN*). Zum Extrahieren von Teilen eines Strings dienen *LEFT\$*, *MID\$* und *RIGHT\$*. Strings oder Teile davon vervielfältigt man mit *RPT\$*. Eine Wandlung in ausschließlich große oder kleine Buchstaben erreichen die Anweisungen *LOWER\$* bzw. *UPPPER\$*. Den Wert des ersten Zeichens eines Strings ermittelt *ASC* – der umgekehrte Weg (*CHR\$*) macht aus dem Zeichencode einen String, der das zugehörige Zeichen enthält. Aus einem String, der eine echte Zahl darstellt, lässt sich der Wert dieser Zahl mit *VAL* bestimmen, mit *EVAL* werden sogar ganze Formeln berechenbar. Und schließlich benutzt man *LOC*, um zu erfahren, ob sich ein einzelnes Zeichen oder eine Zeichenkette in einem String wiederfindet.

## 3 Einfache Datenverarbeitung

---

### Daten für das Programm

Vorbemerkung: Aus Platzgründen werden in diesem Buch oft Beispiele verwendet, die einen bestimmten Sachverhalt näher erläutern, aber im Aufbau nicht vollständig sind. Wenn Sie solche Code-Schnipsel gleich ausprobieren wollen, müssen Sie immer dafür sorgen, dass sie in einer Prozedur stehen (*PROC test: ... ENDP*) und die Variablen deklariert sind!

Wie Werte zu den verschiedenen Variablen innerhalb des Programmes zugeordnet werden, haben Sie gerade erfahren. Die verwendete Art und Weise ist vielleicht für Programmierer noch akzeptabel, wie aber übergibt der Benutzer Daten an ein fertiggestelltes Programm?

Na klar, irgendwie mit Hilfe der Tastatur. Aber die Abfrage der Tastatur muss im Programm vorbereitet sein.

Für numerische und Stringvariablen gibt es dazu zwei getrennte Methoden.

Da ist zunächst einmal die *INPUT*-Anweisung:

```
INPUT wert(1)  REM erwartet Fließkommazahl-Eingabe
INPUT anzahl%  REM erwartet Integerzahl-Eingabe
```

Das Programm stoppt an der Stelle der Anweisung, gibt eine blinkende Eingabemarkierung, den Cursor, auf dem Bildschirm aus und wartet auf die Eingabe einer Zahl und ein abschließendes <Enter>. Die Eingabe muss zu der hinter *INPUT* angegebenen Variablen passen, sonst springt der Cursor in die nächste Zeile und es wird ein Fragezeichen ausgegeben. Ist die Eingabe korrekt, wird sie durch <Enter> in die Variable übernommen und das Programm mit dem nächsten Befehl fortgesetzt. Mit den eingetragenen Werten kann nun beispielsweise gerechnet werden.

Über *INPUT* können auch Strings eingegeben werden, wenn die zugehörige Variable eine Stringvariable ist. Übersichtlicher ist es jedoch, die zweite, nur für Strings reservierte Eingabeweise zu benutzen:

```
EDIT name$  REM erwartet eine String-Eingabe
```

Das Prinzip ist das gleiche wie bereits erläutert und doch gibt es kleine Unterschiede. Wird die deklarierte Länge des Strings überschritten, piepst der Rechner. Außerdem ist es möglich, die Eingabe mit <Enter> abzuschließen, ohne einen Wert eingegeben zu haben. Das Ergebnis ist ein Leerstring. Über *EDIT* können auch Ziffern und Zahlen eingegeben werden, die werden aber als Zeichen interpretiert und man kann mit ihnen nicht rechnen, jedenfalls nicht unmittelbar.

Über die beiden Anweisungen hinaus lassen sich auch einzelne Tastendrücke an das Programm weitergeben. Man muss dazu nicht einmal die <Enter>-Taste betätigen:

Bei *GET* und *GET\$* stoppt das Programm und wartet auf den Druck einer (fast) beliebigen Taste. Ausgenommen sind Tasten, die lediglich modifizierenden Charakter haben, also z.B. <Strg> und <Shift>.

Der Tastendruck liefert in *GET* den Tastaturcode zurück. Wahlweise kann man sich auch das eigentliche Zeichen in einem String, der nur ein Zeichen lang ist, in *GET\$* zurückgeben lassen. Bestimmte Tasten lassen sich später aber nicht unmittelbar am Bildschirm darstellen, Beispiel: die <Enter>-Taste (Wert 13).

Das Ergebnis wird passend deklarierten Variablen übergeben:

```
tastencode%= GET
zeichen$= GET$
```

Während des Programmlaufes wird *GET* gern benutzt, um Entscheidungen abzufragen ("Weitermachen? J/N") oder um dem Benutzer Zeit zu geben, etwas in Ruhe auf dem Bildschirm anzusehen und das Programm dann durch einfachen Tastendruck weiterlaufen zu lassen. Eine Werteübergabe ist im letzteren Fall nicht erforderlich, es reicht also, einfach *GET* in eine Zeile zu schreiben.

Die beiden Anweisungen *KEY* und *KEY\$* leisten das gleiche, das Programm hält jedoch nicht an, sondern nimmt sich den Wert aus dem Tastaturpuffer. Der sammelt nämlich zunächst alle Eingaben, leert sich aber auch sofort wieder, sowie die Eingabe vom Betriebssystem verarbeitet ist. Wenn der Tastaturpuffer gerade keinen Wert enthält, ist das Ergebnis in *KEY* Null bzw. in *KEY\$* ein Leerstring.

## Daten sichtbar machen

Die Datenverarbeitung – wie auch jedes unserer Programme – funktioniert nach dem EVA-Prinzip: Eingabe – Verarbeitung – Ausgabe. Die Eingabe haben Sie kennengelernt, die Verarbeitung kann eine einfache Berechnung sein. Wie aber bekommt man nun die verarbeiteten Daten zu Gesicht?

Hauptausgabeeinheit ist natürlich der Bildschirm. Der lässt sich recht einfach mit dem *PRINT*-Befehl "bedrucken". Nach *PRINT* muss man lediglich noch angeben, was dargestellt werden soll. Diese Angabe kann sowohl der Inhalt einer Variablen oder auch ein direkter Wert (eine "Konstante") sein, selbst Ausdrücke werden in einem Schritt verarbeitet und dargestellt:

```
PRINT "Hallo!"           REM eine Text(String)-Konstante
PRINT name$              REM eine Variable
PRINT anzahl% * hoehe/2  REM ein Ausdruck
```

Nach erledigtem *PRINT*-Befehl wird der Cursor, das ist die Markierung, ab der auf dem Bildschirm geschrieben wird, an den Anfang der nächsten Zeile versetzt. Das entspricht dem Weiterschalten der Zeile und dem Rücklauf des Wagens an einer dieser guten alten Schreibmaschinen (carriage return, Wagenrücklauf) ...

Üblicherweise sieht man den Cursor nicht. Bei Bedarf lässt er sich aber mit den Anweisungen *CURSOR ON* / *CURSOR OFF* an- und wieder abschalten.

Will man mehrere Werte in einer Zeile ausgeben, trennt man die Variablen bzw. Konstanten jeweils durch ein Komma voneinander. An der Stelle des Kommas wird dann auf dem Bildschirm ein Leerzeichen ausgegeben. Wenn das Leerzeichen stört, benutzt man ein Semikolon, dann wird ohne Zwischenraum weitergeschrieben.

```
PRINT name$;" :", alter%,"Jahre"
```

Setzt man Komma oder Semikolon an das Ende einer *PRINT*-Zeile, wird der Cursor nicht versetzt und der nächste *PRINT*-Befehl schreibt in der gleichen Zeile weiter. Das nächste Beispiel bewirkt die gleiche Ausgabe wie das Beispiel zuvor:

```
PRINT name$;" :",
PRINT alter%,
PRINT "Jahre"
```

Die erste *PRINT*-Ausgabe nach einem Programmstart beginnt immer ganz links in der ersten Bildschirmzeile zu schreiben. Wird der untere Rand des Bildschirmes erreicht, verschwindet mit dem nächsten *PRINT*-Befehl die erste Zeile, wodurch die unterste Zeile für die neue Ausgabe frei wird.

Das wirkt nicht besonders elegant. Abhilfe schafft der Cursor-Positionierungsbefehl *AT*, mit dem man dem System sagt, an welcher Stelle des Bildschirmes man schreiben möchte:

```
AT x%,y%
```

Im Beispiel sind x%,y% die x- und y-Koordinaten des Cursors, also die Angabe von Spalte und Zeile. *AT 1,1* steht dabei für die linke obere Ecke des Bildschirms. Das ist die Position, die der Cursor nach dem Programmstart einnimmt. Hier verharrt er auch nach dem Befehl *CLS*, der für das Löschen des Bildschirms sorgt.

```
PROC Liste:
  AT 10,3 : PRINT "Werkzeugliste"
  AT 13,5 : PRINT "1. Hammer"
  AT 13,6 : PRINT "2. Säge"      REM usw.
  GET
  CLS
  PRINT "Linke obere Ecke"
  GET
ENDP
```

In diesem Beispiel sehen Sie, dass man mehrere Befehle in eine Zeile schreiben darf, wenn man sie durch eine Folge von Leerzeichen-Doppelpunkt-Leerzeichen voneinander trennt. Aus Gründen der Übersicht meidet man das in den meisten Fällen, aber gelegentlich fördert diese Schreibweise auch die Übersicht.

Die zu wählenden Koordinaten sind keine festen Werte, sondern hängen von Bildschirm- und Schriftgröße ab.

Zusammenfassendes Beispiel – eine Kontoabhebung:

```
PROC Konto:
  LOCAL name$(255), kontostand, abbuchung
  kontostand=1234.56
  PRINT "Name eingeben:",
  EDIT name$
  PRINT "Abbuchungsbetrag eingeben:",
  INPUT abbuchung
  CLS
  AT 10,2 : PRINT "Hallo Herr/Frau " + name$
  AT 10,3 : PRINT "Ihr Kontostand:", kontostand
  AT 10,4 : PRINT "Erleichtert um:", abbuchung
  kontostand= kontostand - abbuchung
  AT 10,5 : PRINT "Neuer Bestand:", kontostand
  AT 10,7 : PRINT "Weiter mit Taste!"
  PAUSE -100
ENDP
```

Der Befehl *PAUSE* sorgt mit Hilfe der Zeitangabe "100" für rund 5 Sekunden Pause, bevor das Programm beendet wird, d.h., der Wert "1" steht für eine zwanzigstel Sekunde. Gibt man den Pausenwert wie im Beispiel negativ an, beendet jeder Tastendruck die Pause vorzeitig.

Dieses einfache Programm lässt noch nicht den optischen Gestaltungsspielraum erkennen, den man in der Praxis hat. Durch die Anweisungen *FONT* und *STYLE* sind Sie in der Lage, andere Schriften zu verwenden und deren Darstellung zu beeinflussen.

```
FONT font&,stil%
```

erlaubt die Angabe der Font-Nummer. Das Thema ist etwas umfangreicher, ich komme später noch darauf zurück. Benutzen Sie vorerst folgende Werte:



font&	Font-Name	Pixelhöhe
4	Courier	8
5	Times	8
6	Times	11
7	Times	13
8	Times	15
9	Arial	8
10	Arial	11
11	Arial	13
12	Arial	15
13	Tiny	4

Die Darstellung der Schrift wird zusätzlich durch den Stil beeinflusst:

stil%	Aussehen
0	normal
1	fett
2	unterstrichen
4	invers
8	doppelte Höhe
16	gleicher Zeichenabstand
32	kursiv

Die Werte dürfen durch Addieren kombiniert werden.

Was zunächst nach einer bequemen Gestaltungsmethode aussieht, ist es am Ende leider nicht, denn jede *FONT*-Anweisung löscht zuallererst einmal aus technischen Gründen den Bildschirm. Damit ist eine Mischung verschiedener Zeichensätze bzw. Buchstabengrößen nicht möglich. Zumindest nicht auf diese Art.

Die einzigen Änderungen, die ohne Bildschirmlöschen noch möglich sind, werden mit der *STYLE*-Anweisung vorgenommen:

```
STYLE stil%
```

Leider steht nicht die volle Palette wie für *FONT* zur Verfügung, lediglich diese zwei Werte:

```
stil%= 2 - unterstrichen und
stil%= 4 - invers.
```

Bis Sie die anderen Möglichkeiten kennenlernen, kommen Sie für das Nachvollziehen der folgenden Beispiele sicherlich mit dem Standardfont (Courier, 8 Pixel hoch) aus. Der Vollständigkeit halber aber noch dieser Hinweis:

Das Raster der Spalten und Zeilen ist je nach Font-Art unterschiedlich groß. Auf Pixel umgerechnet, nutzt das Maschennetz nicht immer den gesamten Bildschirm aus. Dadurch resultieren bei den einzelnen Fonts unterschiedliche Abstände der ersten Buchstaben vom oberen und linken Bildschirmrand. Mehr über die gerade aktuellen Werte erfährt man mit Hilfe des *SCREENINFO*-Befehls (s. alphabetische Befehlsübersicht). Die jeweils notwendige Rekonfiguration des unsichtbaren Koordinaten-Netzes ist auch der Grund, weshalb nach jeder *FONT*-Anweisung der Bildschirm gelöscht wird.

Wie man mit *SCREEN* Einfluss auf die aktive Bildschirmgröße nimmt, lesen Sie bitte ebenfalls in der alphabetischen Befehlsliste nach.

## 4 Verzweigungen

---

### Entscheidungen treffen

Unsere Programme bestehen nicht nur aus einer geraden Folge von Anweisungen und Funktionen. Oft ist es notwendig, ganz bestimmte Anweisungen auszuführen, wenn die Bedingungen das erfordern. In einem Taschenrechner-Programm z.B. muss das Programm wissen, was es eigentlich mit den eingegebenen Zahlen machen soll: addieren, multiplizieren, Wurzel ziehen ... Es muss also Möglichkeiten geben, gezielt auf spezifische Programmstücke zu verzweigen. Dabei spielt es keine Rolle, ob der Anlass dazu von aussen oder aus dem Programm heraus kommt.

In OPL gibt es dazu die *IF .. ELSEIF .. ELSE .. ENDIF*-Konstruktion. Die einfachste Verzweigung sieht so aus:

```
PRINT "Programm beenden?j/n"
t$= GET$
IF t$="j"
    STOP      REM beendet das Programm
ENDIF
REM es geht weiter
```

Nach Beantwortung der Frage durch einen Tastendruck wird mit *IF* geprüft, ob es sich bei dem Tastendruck um die Taste "j" handelte. Die Prüfbedingung wird immer hinter *IF* in derselben Zeile notiert. Nur wenn die Bedingung erfüllt ist, wird das Programmstückchen zwischen *IF* und *ENDIF* ausgeführt. Alle anderen Tastendrucke führen zur Weiterführung des Programmes hinter *ENDIF*.

Noch ist eine "Verzweigung" nicht so recht zu erkennen. Das wird aber sofort deutlich:

```
PRINT "Programm beenden?j/n"
t$= GET$
IF t$="j"
    PRINT "Programm endet in 2 Sekunden"
    PAUSE 40
    STOP      REM beendet das Programm
ELSE
    PRINT "Nicht angehalten"
ENDIF
REM hier es geht weiter
```

Nun haben wir zwei echte Zweige. Wenn die Taste "j" gedrückt wurde, wird der Programmtext zwischen *IF* und *ELSE* abgearbeitet, ansonsten das Programm zwischen *ELSE* und *ENDIF* ausgeführt, egal, welche andere Taste gedrückt wurde. *ELSE* darf in einer *IF .. ENDIF*-Konstruktion nur je einmal verwendet werden. Nach Abarbeitung einer der beiden Zweige geht es normal hinter *ENDIF* weiter.

In anderen Anwendungsfällen reicht diese einfache entweder-oder-Verzweigung nicht aus, man benötigt eine ganze Palette an Auswahlmöglichkeiten. Mit *ELSEIF* lässt sich das verwirklichen:

```

PRINT "Datenbankmenü"
PRINT "(n)euer Eintrag"
PRINT "(s)uchen"
PRINT "(l)öschen"
t$= GET$
IF t$="n"
    REM neuen Datensatz erstellen
ELSEIF t$="s"
    REM Datensatz suchen
ELSEIF t$="l"
    REM Datensatz löschen
ELSE
    REM nichts tun
ENDIF
REM hier es geht weiter

```

*ELSEIF* darf, wie ersichtlich, mehrfach verwendet werden. Der Bereich der einzelnen Zweige wird jeweils durch die Schlüsselwörter abgegrenzt, der eigentliche Programmtext wird hier durch die Kommentare angegeben.

Wenn die Programmteile in den einzelnen Zweigen zu umfangreich werden, ist es angebracht, diese in eigenen Prozeduren unterzubringen. So behält man am besten die Übersicht.

Da *ELSEIF* und *ELSE* optionale Bestandteile der *IF .. ENDIF*-Konstruktion sind, ist auch ein Aufbau der Form *IF .. ELSEIF .. ENDIF* (also ganz ohne *ELSE*) erlaubt.

Insgesamt dürfen bis zu acht *IF .. ENDIF*-Entscheidungen ineinander verschachtelt sein, sinngemäß etwa so:

```

IF bedingung1
    IF bedingung2
        IF bedingung3
            REM ...
        ENDIF
    ENDIF
ENDIF

```

Im Laufe der Zeit werden Sie diverse Beispiele für die Nutzung von *IF .. ENDIF* zu Gesicht bekommen. Darunter werden sich auch ein paar scheinbar ungewöhnliche Ausführungen der *IF*-Abfrage befinden. Deshalb sehen wir noch einmal etwas genauer drauf.

*IF* prüft den Ausdruck, der die Prüfbedingung enthält, auf seinen Wahrheitsgehalt. Ist die abgefragte Bedingung erfüllt, also WAHR (engl. TRUE), wird der *IF*-Zweig durchlaufen. Andernfalls lautet das Prüfergebnis FALSCH (engl. FALSE) und das Programm nimmt einen andern Weg. Der Computer kennt aber kein begriffliches Ergebnis WAHR oder FALSCH, sondern nur Zahlen. Deshalb liefert die Prüfung für WAHR den Wert -1 und für FALSCH den Wert Null an den Fragesteller (*IF*) zurück. *IF* begreift aber auch jede andere Zahl außer Null als WAHR und führt den entsprechenden Programmzweig aus.

Mit diesem Wissen im Hinterkopf ist es leichter, sich beim Programmieren vorzustellen, wie *IF* und *ELSEIF* reagieren und fremde Programmtexte sind leichter zu verstehen, z.B.:

```

wert%= 22
IF wert%
    PRINT wert%
ENDIF

```

Scheinbar gibt es gar keine Bedingung, nach der gefragt wird. Und doch: Gefragt wird nach dem Wahrheitswert von *wert%*. Da *wert%* nicht Null ist, ist das Ergebnis der Prüfung WAHR und der Inhalt von *wert%* wird auf dem Bildschirm ausgegeben.

Hinter *IF* müssen also immer auswertbare Formulierungen stehen. Das folgende geht beispielsweise deshalb nicht:

```
IF a$
```

Beim Einsatz von vergleichenden Operatoren gelten diese Regeln:

```
a=b   ist WAHR, wenn a und b gleich groß
a<>b  ist WAHR, wenn a und b verschieden sind
a>b   ist WAHR, wenn a größer als b ist
a>=b  ist WAHR, wenn a größer als oder gleich b ist
a<b   ist WAHR, wenn a kleiner als b ist
a<=b  ist WAHR, wenn a kleiner als oder gleich b ist
```

Daneben gibt es die logischen Operatoren *AND*, *OR* und *NOT*, die den Wahrheitsgehalt der einzelnen Angaben auf verschiedene Weise prüfen. Es gilt:

```
a AND b   ist WAHR, wenn a und b nicht Null sind
a OR  b    ist WAHR, wenn a oder b nicht Null ist
NOT a     ist WAHR, wenn a FALSCH (also Null) ist
```

Dabei dürfen a und b selber wiederum berechenbare Ausdrücke sein, z.B.:

```
(a>b) AND (b<=24)
```

Achtung: Die logischen Operatoren werden zu bitweise arbeitenden Operatoren, wenn Integerzahlen beteiligt sind! Die beiden folgenden Beispiele liefern wegen der verwendeten Integerzahlen unterschiedliche Ergebnisse, mit Fließkommazahlen wären beide Fälle WAHR:

```
a%=21 : b%=2
IF a% AND b%          REM ist Null! = FALSCH
IF (a%>0) AND (b%>0)  REM ist -1    = WAHR
```

Das bitweise "UNDieren" ergibt mit den angegebenen Werten Null, daher ist das Prüfergebnis der ersten Zeile auch Null. Wäre a%=22, ergäbe das bitweise "UNDieren" 2, das Prüfergebnis wäre WAHR.

Für die Verknüpfung von Bits gelten folgende Regeln:

```
AND
1 AND 1 --> 1
1 AND 0 --> 0
0 AND 1 --> 0
0 AND 0 --> 0
```

```
OR
1 OR 1 --> 1
1 OR 0 --> 1
0 OR 1 --> 1
0 OR 0 --> 0
```

```
NOT rechnet in dieser Weise:
NOT 0 --> -1
NOT -1 --> 0
```

Die bitweisen Operationen werden gern dazu benutzt, um auf gesetzte Bits zu prüfen. So liefert der Befehl *KMOD* ein oder mehrere gesetzte Bits in eine Integervariable zurück, wenn eine Modifiziertaste (Strg, Shift, ..) gedrückt wird. Die anschließende Analyse findet heraus, welche Taste es war und behandelt den Tastendruck entsprechend:

```

PROC Entscheid:
  LOCAL taste$(1),mod%
  taste$=GET$
  mod%= KMOD
  IF mod% AND 2
    PRINT "Shift gedrueckt"
  ELSEIF mod% AND 4
    PRINT "Strg gedrueckt"
  ELSEIF mod% AND 128
    PRINT "Fn gedrueckt"
  ENDIF
  GET
ENDP

```

Wenn beispielsweise die Shifttaste mitgedrückt wurde, sieht das binär wie folgt aus:

```

mod%=2    0000 0000 0000 0010
AND 2     0000 0000 0000 0010
ergibt    0000 0000 0000 0010

```

Das Ergebnis ist WAHR.

## 5 Schleifen

---

### Wiederholungen leicht gemacht

Schleifen sollen dem Programmierer die Mühe ersparen, gleichartige Anweisungen mehrfach aufschreiben zu müssen.

Um die Zahlen von 1 bis 10 auf dem Bildschirm auszugeben, wäre beispielsweise eine Folge von *PRINT*-Anweisungen denkbar:

```
PRINT "1"
PRINT "2"
...
PRINT "10"
```

Eine Schleife erledigt das viel einfacher. Mit Worten ausgedrückt, muss folgendes geschehen, um das gleiche Ergebnis zu erhalten:

- definiere eine Zählvariable *n%*
- gib ihr einen Anfangswert
- beginne eine Schleife:
  - gib den Wert von *n%* aus: *PRINT n%*
    - erhöhe *n%* um eins: *n% = n% + 1*
  - prüfe, ob das Schleifenende erreicht ist:
    - wenn *n%* größer als 10 ist, beende die Schleife
    - sonst wiederhole die Schleife
- weiter mit dem Programm

Was sich so umständlich beschreibt, ist als Programmtext einfach und elegant:

```
PROC UNTILschleife:
  LOCAL n%
  n% = 1
  DO
    PRINT n%
    n% = n% + 1
  UNTIL n% > 10
  GET
ENDP
```

Die Schleife ist in die Anweisungen *DO* und *UNTIL* eingeschlossen. In der *UNTIL*-Zeile wird die sogenannte Abbruchbedingung ("wenn *n%* größer als 10 ist") geprüft. Die Prüfung erfolgt ähnlich wie bei *IF*: Fällt die Prüfung positiv aus, liefert sie *TRUE* (-1) zurück und die Schleife wird beendet, das Programm setzt seinen Gang in der Zeile nach *UNTIL* fort. Bei *FALSE* (0) werden alle Anweisungen in der Schleife erneut durchlaufen, bis die Prüfung endlich *TRUE* ergibt.

Hier ist unbedingt ein Stopperstein zu beachten! Formuliert man die Abbruchbedingung falsch oder vergißt man, die Zählvariable zu verändern, läuft die Schleife unter Umständen unendlich. Meist bleibt dann nur der manuelle Programmabbruch mit der Tastenkombination <Strg><Esc> übrig.

Neben der *DO ..UNTIL*-Schleife existiert eine weitere Schleifenform, die durch die Schlüsselwörter *WHILE* und *ENDWH* eingegrenzt wird.

In diesem Falle wird die Abbruchbedingung bereits am Schleifenbeginn abgefragt. Solange das Ergebnis *TRUE* ist, läuft die Schleife. Unser vorheriges Beispiel umformuliert:

```
PROC WHILEschleife:
  LOCAL n%
  n%= 1
  WHILE n%<11
    PRINT n%
    n%= n%+1
  ENDWH
ENDP
```

Diese Konstruktion scheint keine Vorteile zu bieten, warum also zwei Schleifenarten?

Es gibt einen wesentlichen Unterschied: Eine *DO..UNTIL*-Schleife wird mindestens einmal durchlaufen, da die Prüfung auf die Abbruchbedingung am Schleifenende erfolgt, während eine *WHILE..ENDWH*-Schleife unter Umständen gar nicht erst "betreten" wird. Bei langen Programmtexten in der Schleife kann daher *WHILE..ENDWH* die bessere Wahl sein – es ergibt sich eine Zeitersparnis. Der andere Unterschied: *DO..UNTIL* bricht ab, wenn die Prüfung *TRUE* ist, während *WHILE..ENDWH* den Schleifendurchlauf fortsetzt, solange die Bedingungsprüfung *TRUE* ergibt. Daraus resultiert gelegentlich eine besser verständliche Formulierung der Abbruchbedingung, was einem bei späterer Wiederverwendung des Programmes das Einarbeiten erleichtert.

Die bisher beschriebenen Schleifen haben einen deutlichen Anfangs- und einen durch die Prüfbedingungen vorgegebenen Endpunkt. Man spricht daher von endlichen Schleifen. Es kommen aber auch absichtlich offene (Endlos-) Schleifen vor, die ebenfalls beendet werden müssen. Man findet häufig solche Konstruktionen:

```
DO
  ...
UNTIL 0    REM = FALSCH

oder:

WHILE 1    REM = WAHR
  ...
ENDWH
```

Die Abbruchbedingung wird dann im Inneren der Schleife formuliert. Wird sie erreicht, sorgt die parameterlose *BREAK*-Anweisung dafür, dass sofort abgebrochen und das Programm nach *UNTIL* bzw. *ENDWH* fortgesetzt wird.

```
PROC Satz:
  LOCAL text$(255),a%
  WHILE 1
    a%=GET
    IF a%= 27    REM Esc
      BREAK
    ENDIF
    Text$= text$ + CHR$(a%)
    REM CHR$() wandelt Zeichencode in String um
  ENDWH
  PRINT text$
  GET
ENDP
```

Die Puristen der "strukturierten Programmierung" sind mit dem Sprungbefehl *BREAK* nicht so recht glücklich: *BREAK* lässt sich durchaus mehrfach in der Schleife anordnen und führt so zu unübersichtlichen und schlecht

pflegbaren Programmen. Das Thema wird uns noch beschäftigen. Achten Sie also darauf, dass Schleifen nur einen Eingang und einen einzigen Ausgang haben.

Auch *CONTINUE* ist ein Sprungbefehl, den Sie aus Gründen einer einsehbaren Programmierung sparsam einsetzen sollten. *CONTINUE* springt innerhalb einer Schleife jeweils sofort zu den Prüfanweisungen, also zu *WHILE* oder *UNTIL*. Die abzuarbeitenden Programmtexte der Schleife lassen sich so unter bestimmten Bedingungen reduzieren, was wiederum die Bearbeitungszeit beträchtlich verringern kann.

Ein Beispiel zeigt, wie es geht:

```
WHILE a%>32
  ...
  IF n%<>44
    CONTINUE
  ENDIF
  ...
ENDWH
```

Um das Gelernte zu vertiefen, sehen wir uns noch ein Beispiel an, für das wir Zufallszahlen benötigen. Die stellt uns die Funktion *RND* zur Verfügung. Sie liefert Zahlen von 0 (einschließlich) bis 1 (ausschließlich):

```
a= RND
```

Damit man ganzzahlige Zufallszahlen für einen bestimmten Bereich bekommt, verfährt man nach diesem Muster:

```
grenzwert%=10
a%= 1 + INT(grenzwert% * RND)
```

*INT* macht aus Fließkommazahlen Integers, indem es Kommastellen einfach abschneidet. Es handelt sich also nicht um eine mathematische Rundung! Das Beispiel erzeugt Zufallszahlen von 1 bis 10.

Letztlich liefert kein Standardcomputer echt zufällige Zahlen. Um nun bei gleichen Startbedingungen nicht immer die gleichen Zufallszahlen zu erhalten, verändert man die Startbedingungen für *RND* durch die irgendwo davor zu setzende Anweisung *RANDOMIZE*. Der Parameter dahinter muss ein sich zeitlich verändernder Wert sein. Am einfachsten gewinnt man den aus dem eingebauten Timer, der Werte für Stunden, Minuten und Sekunden mit Hilfe der zugehörigen Funktionen liefern kann.

```
RANDOMIZE HOUR + MINUTE + SECOND
grenzwert%=10
a%= 1 + INT(grenzwert% * RND)
```

Das Ergebnis für *a%* ist nun bei jedem Programmstart ein anderes. *RANDOMIZE* braucht man nur einmal aufzurufen, am besten beim Programmstart.

Nun das Beispiel. Es zeigt, wie man mit Schleifen und Entscheidungen bereits wirkungsvolle Programme schreiben kann.

```
PROC Mathe:
  REM Einmaleins-Trainer
  LOCAL taste$(1)  REM gedruckter Buchstabe
  LOCAL a%,b%,c%   REM Rechenvariablen
  LOCAL erg%
```

(Fortsetzung nächste Seite)



```

RANDOMIZE MINUTE + SECOND
DO
  CLS
  AT 10,2 : PRINT "EINMALEINS TRAINER"
  AT 10,4 : PRINT "Auswahl:"
  AT 12,5 : PRINT "(T)esten"
  AT 12,6 : PRINT "(B)eenden"
  a%= 1 + INT(10*RND)
  b%= 1 + INT(10*RND)
  taste$= LOWER$(GET$)
  IF taste$= "t"
    c%= a%*b%
    AT 10,8 : PRINT "Wieviel ist",a%,"*",b%,
    INPUT erg%
    AT 10,9
    IF erg%=c%
      PRINT "Das ist richtig!"
    ELSE
      PRINT "Das ist falsch!"
    ENDIF
    GET
  ENDIF
UNTIL taste$="b"
ENDP

```

## 6 Prozeduren

---

### Einfache Prozeduraufrufe

Das obige Programm *Mathe*: ist so kurz und übersichtlich, dass man es noch nicht besonders gliedern muss. Stellen Sie sich trotzdem vor, es wäre umfangreich. Unter solchen Umständen behält man die Übersicht, indem man das Programm in mehrere Prozeduren aufgliedert, die man abhängig von ihrer Funktion aufruft. Sie haben am allerersten Beispiel ("Hallo Welt!") schon kennengelernt, dass man dazu nur den Namen mit einem Doppelpunkt angeben muss – ein weiteres Beispiel:

```
PROC Mathe1:
...
  ZeigeMenue:
...
ENDP

PROC ZeigeMenue:
...
  PRINT "Einmaleins-Trainer"
  PRINT "(T)esten"
  PRINT "(B)eenden"
...
ENDP
```

Es gibt aber mehr als nur diese eine Art des Aufrufes, denn man muss manchmal sowohl Daten an die aufgerufene Prozedur übergeben als auch von ihr zurückbekommen können.

### Parameterübergabe

Zum Übergeben von Daten hängt man sie an den Prozedurnamen als "Parameter" in Klammern an. Die aufgerufene Prozedur (das "Unterprogramm") muss so vorbereitet sein, dass sie die Parameter übernehmen kann. Deshalb stehen hinter ihrem Namen genauso viele "Auffang"-Variablen, wie die aufrufende Prozedur übergibt:

```
PROC Mathe2:
...
  a%=2 : b%=7
  Ausgabe: (a%,b%)
...
ENDP

PROC Ausgabe: (x%,y%)
...
  PRINT "Ergebnis:",x%*y%    REM Bildschirmausgabe
...
ENDP
```

Dass dabei die "Auffang"-Variablen (also die Eingabe-Parameter im Unterprogramm) vom gleichen Typ sein müssen, versteht sich fast von selbst. Sie dürfen und sollten sogar wegen der Unterscheidbarkeit für den Programmierer andere Namen besitzen. Es ist nicht nötig, diese Variablen im Unterprogramm zu deklarieren, das geschieht gewissermaßen automatisch. Mit ihnen kann man in üblicher Weise Berechnungen anstellen, allerdings lassen sie es nicht zu, dass ihr Wert verändert wird. Man betrachte sie daher als lokale Variable mit

konstantem Wert, was in OPL der Definition einer Konstanten gleichzusetzen ist. Der umgekehrte Weg – diese Konstante einer anderen Variablen zuzuordnen – funktioniert natürlich.

## Funktionen

Unterprogramme sind ideal geeignet, Funktionen aufzubauen. Funktionen füttert man mit Argumenten, die sie verarbeiten und ein Ergebnis wieder herausgeben. Die Übergabe in die Hin-Richtung haben Sie eben schon kennengelernt, zur Übergabe in die Rück-Richtung wird im Unterprogramm die Anweisung *RETURN* benutzt.

```
PROC Mathe3:
...
a%=2 : b%=7
c%= Berechne%:(a%,b%)
PRINT c%
...
ENDP

PROC Berechne%:(x%,y%)
...
z%= x%*y% : RETURN z%
...
ENDP
```

Prozeduren, die einen Wert zurückgeben sollen, sind am Ende ihres Namens mit einem Typ-Identifikator zu versehen. In unserem Beispiel wird eine Ganzzahl zurückgegeben, im Prozedurnamen findet sich folglich das "%" -Zeichen wieder. Das gilt dann sinngemäß auch für die Rückgabe von Longintegers ("&") und Strings ("\$"). Unterprogramme, die Fließkommazahlen zurückgeben, haben naturgemäß keine zusätzliche Kennzeichnung.

Die Verwendung dieser Methode ist keineswegs steif, ganz im Gegenteil. So ist das letzte Beispiel auch noch viel kürzer zu fassen:

```
PROC Mathe3a:
...
a%=2 : b%=7
PRINT Berechne%:(a%,b%)
...
ENDP

PROC Berechne%:(x%,y%)
...
RETURN x%*y%
...
ENDP
```

## Pro & Contra GLOBAL

Einen Schönheitsfehler hat die Rückgabe mit *RETURN*: Es kann nur ein einziger Wert zurückgegeben werden. Wer mehr braucht, ist gezwungen globale Variablen zu verwenden, die jeder Prozedur, die "unter" der aufrufenden Prozedur liegt, bekannt und deshalb verarbeitbar sind.

Sie mögen sich fragen, warum man sich eigentlich überhaupt mit Werteübergaben auseinandersetzt, anstatt einfach generell globale Variablen zu verwenden. Es existieren zwei einfache, aber desto stärkere Argumente gegen die globalen Variablen.

Zum einen lassen sich "sauber" geschriebene Unterprogramme mit einer "Schnittstelle" und ohne globale Variablen völlig eigenständig auf ihre Funktion testen. Das erspart eine Menge Arbeit bei der Fehlersuche! Zudem sind solche Programmstückchen ("Module") wesentlich einfacher wiederzuverwenden, vorausgesetzt, sie sind sinnig geschrieben.

Zum anderen verliert sich mit der Anzahl von Variablen und Programmzeilen die Übersicht über die Funktion der einzelnen Variablen. So kommt es immer wieder zu Programmfehlern durch unabsichtliche Manipulationen, nach deren Quelle man oft stundenlang fahndet.

Muss man aber doch zu globalen Variablen greifen (es lässt sich selten ganz umgehen), notiere man sich deren Bedeutung unbedingt in den Deklarationszeilen und vermeide jede "Fremdbenutzung". Außerdem ist auch eine möglichst eindeutige Namensgebung hilfreich.

## Z.B.: Lotto

Zur Anschauung folgt ein Beispiel, in dem sowohl für die "Werteübergabe" zum als auch für die "Werterückgabe" vom Unterprogramm globale Variablen benutzt werden. Wir wollen uns ein Programm zimmern, in dem uns ein Vorschlag für das Ankreuzen eines Lotto-Tipps in der Spielart "6 aus 49" unterbreitet wird.

Um die Übersicht zu behalten, bauen wir das Programm modular auf – jede Funktion bekommt ihre eigene Prozedur und braucht nur noch aufgerufen zu werden. Damit man das Gesamtprogramm für andere Lottoarten wiederverwenden kann, wird es von vornherein etwas allgemeiner gefasst. So bedarf es nur weniger Handgriffe (Änderung von Variablenwerten), um es anzupassen.

```
PROC Lotto:
  REM Vorschlag fuer die naechste Lottoziehung
  GLOBAL anz%          REM Anzahl zu ziehender Zahlen
  GLOBAL max%          REM max. Wert der Ziehungszahlen
  GLOBAL tipp%(6)      REM gezogene Lottozahlen
  LOCAL taste%         REM fuer Tastendruck
  RANDOMIZE MINUTE + SECOND
  anz%= 6 : max%= 49   REM "6... aus 49"
  DO
    CLS
    Ziehung:
    Sortieren:
    Ausgabe:
    PRINT "Weiter=Enter, beenden=Esc"
    taste%= GET
  UNTIL taste%=27
ENDP
PROC Ziehung:
  REM Ziehung & Kontrolle auf Doppel
  REM Ergebnisse landen im globalen Array tipp%()
  LOCAL n%,z%          REM Zaehlvariablen
  n%=1
  DO
    tipp%(n%)= 1 + INT(max%*RND)
    REM Kontrolle auf doppelte Werte:
    z%=1
    WHILE NOT(tipp%(z%)= tipp%(n%))
      z%= z% + 1
    ENDWH
    IF n%=z%
      REM letzter Wert nicht doppelt, also weiter
      n%= n% + 1
    ENDIF
  UNTIL n%>anz%
ENDP
```

```

PROC Sortieren:
  REM sortiert globales Array tipp%() aufsteigend
  LOCAL l%      REM aktuelles linkes Fach
  LOCAL r%      REM aktuelles rechtes Fach
  LOCAL temp%   REM Hilfsvariable beim Wertetausch
  l%=1
  DO
    r%=l%+1
    DO
      IF tipp%(r%)<tipp%(l%)
        REM Werte vertauschen
        temp%=tipp%(l%)
        tipp%(l%)=tipp%(r%)
        tipp%(r%)=temp%
      ENDIF
      r%=r%+1
    UNTIL r%>anz%
    l%=l%+1
  UNTIL l%=anz%
ENDP

PROC Ausgabe:
  REM gibt die Werte des globalen Arrays tipp%() aus
  LOCAL n%      REM Zaehlvariable
  n%= 1
  DO
    PRINT tipp%(n%)
    n%=n%+1
  UNTIL n%>anz%
ENDP

```

Den Quelltext sollten Sie mit dem bisher Gelernten gut selbst interpretieren können. Zur Unterstützung trotzdem noch ein paar Erläuterungen.

In *PROC Ziehung*: lautet die Bedingung hinter *WHILE*:

```
NOT(tipp%(z%) = tipp%(n%))
```

Man erinnere sich, dass die Bedingung so interpretiert wird: "Die Schleife läuft SOLANGE, wie die Bedingung erfüllt ist."

Die neu gezogene Zahl wird mit den vorhergehenden bereits gezogenen Zahlen verglichen. Sollte es eine Identität geben, wird die *WHILE*-Schleife abgebrochen, der letzte Ziehungswert verworfen und noch einmal gezogen. Die Schleife läuft also solange, wie die Vergleichswerte NICHT identisch sind. Jedoch, wenn n% und z% gleich groß sind, wird die neu gezogene Zahl mit sich selber verglichen. Als Folge wird die Schleife abgebrochen, wobei die gezogene Zahl mit Sicherheit nun nicht doppelt vorliegt und zur nächsten Ziehung übergegangen werden kann (n% wird um eins erhöht).

Es folgt das Sortieren. Da wir es hier nicht mit einem Riesenberg von Zahlen zu tun haben, sind wir in der Auswahl des Sortierverfahrens nicht begrenzt und greifen aus dem Angebot die Bubble-Sort-Methode heraus. Hier werden systematisch Werte verglichen und die jeweils kleineren und größeren in entgegengesetzte Positionen verschoben. Die großen steigen dabei quasi wie Blasen nach oben auf, was der Methode den anschaulichen Namen gegeben hat.

Wir lassen die "Blasen" nicht steigen, sondern arbeiten – genauso anschaulich – horizontal: Stellen Sie sich eine Reihe von links nach rechts angeordneter Fächer vor, in denen man Zettel ablegen kann. Notieren Sie in Gedanken die Ergebnisse der Ziehung auf je einen Zettel und legen Sie diese der Ziehungs-Reihenfolge nach in ein Fach.

Wir wollen erreichen, dass im weitesten links befindlichen Fach die kleinste Zahl liegt. Das Prinzip geht so:

Nimm aus dem ersten Fach den Zettel mit der gezogenen Zahl. Vergleiche sie mit der in Fach zwei. Ist die Zahl aus Fach zwei kleiner, lege sie ins Fach eins, die andere in Fach zwei. Nimm wieder die Zahl aus Fach eins und vergleiche sie diesmal mit der in Fach drei. Ein Austausch ist nur erforderlich, wenn die Zahl in Fach drei kleiner ist. So vergleicht man nach und nach die aktuelle Zahl in Fach eins mit den Werten in den anderen Fächern. Nach Vergleich mit Fach sechs endet die erste Runde.

Im Ergebnis hat man nun mit Sicherheit im Fach eins die kleinste Zahl! Daher kann man dieses Fach nun auch getrost vergessen. Gedanklich stellt man sich nun vor Fach zwei, vergleicht Zug um Zug mit den weiter rechts liegenden Zahlen und tauscht nach obigem Prinzip die Plätze, soweit das erforderlich ist. Nach diesem Durchgang liegt nun die zweitkleinste Zahl in Fach zwei. Setzen Sie das Gedankenspiel selber weiter fort. Beachten Sie: Um die Werte austauschen zu können, wird eine temporäre Hilfsvariable benötigt. Beim Versuch, direkt auszutauschen, geht wenigstens ein Wert verloren!

## Aufruf per @-Operator

Eine besondere Art von Prozeduraufrufen gestattet es, je nach Benutzereingabe oder aufgrund anderer Entscheidungen, verschiedene Prozeduren anzusprechen. Ein Anwendungsbeispiel findet sich im Kapitel "Menüs". Damit ist man sehr flexibel und kommt ohne ellenlange und langsame *IF .. THEN*-Konstruktionen aus.

Der übliche Aufruf einer Prozedur

```
...
up:      REM Unter-Prozedur up aufrufen
...
```

wird ersetzt durch

```
...
prozname$="up"
@(prozname$):
...
```

Die Übergabe von Parametern funktioniert wie üblich:

```
...
@(prozname$):(a,b%,c%)
...
```

Funktionen ruft man in der normalen Form so auf:

```
...
intzahl%= up%:
...
```

Mit @-Operator wird daraus:

```
...
prozname$="up"
intzahl%= @(prozname$):
...
```

oder direkt aufgeschrieben:

```
...
intzahl%= @%("up"):
...
```

Der Typsuffix "%" wandert also an den @-Operator und wird im Prozedurnamen nicht noch einmal genannt – obwohl die aufgerufene Funktion "up%:" heißt! Die gleichen Prinzipien gelten natürlich für alle Funktionsaufrufe in dieser Form.

Noch einmal die allgemeine Übersicht:

```

proznahme$="up"
intzahl%= @(proznahme$):(p1,p2,..)
          REM ruft: up%:(a1,a2,..)
longint&= @&(proznahme$):(p1,p2,..)
          REM ruft: up&:(a1,a2,..)
fk_zahl  = @(proznahme$):(p1,p2,..)
          REM ruft: up:(a1,a2,..)
text$    = @$ (proznahme$):(p1,p2,..)
          REM ruft: up$:(a1,a2,..)

```

## 7 Menüs

---

### Das Prinzip

Bisher haben Sie einige wenige und dazu noch "selbstgestrickte" Menüs kennengelernt. Aber auch kleine Programme sollten, insbesondere, wenn man sie in andere Hände gibt, ein wenig vom gewohnten Charme der Standardprogramme ausstrahlen. Menüs zur Benutzerführung gehören zweifellos dazu. Sie sind auch gar nicht so schwierig zu programmieren. Darüber hinaus ermöglicht man die Benutzung des Stiftes, ohne selbst allzuviel dazu tun zu müssen.

Die Menüs sind im Prinzip gegliederte Listen, mit deren Hilfe sich Einstellungen und Aktionen bequem ansteuern und ausführen lassen. Sie werden im Programm durch die *minIT*-Anweisung eingeleitet, dann definiert und durch die abschließende *MENU*-Anweisung aktiviert.

Durch die Definition wird festgelegt, wie die einzelnen Menü-Punkte heißen und welche Shortcut-Tasten dazu gehören. Shortcut-Tasten ("Tastaturkürzel") sind Tastenkombinationen, mit denen man die hinter den Menüpunkten verborgenen Prozeduren direkt ansprechen kann, ohne die Menüs öffnen zu müssen.

Einfaches Beispiel:

```
minIT
  mCARD "Datei", "Neu", %n, "Öffnen", -%O, "Ende", %e
  mCARD "Extras", "Einstellen", %q, "Hilfe", %H, "Über ...", %a
m%=MENU
```

Die Definition jedes Menüs wird mit dem Schlüsselwort *mCARD* eingeleitet, unmittelbar dahinter folgt der Menü-Titel. Der Titel und die dann folgenden Menüpunkte werden als Strings angegeben und sind daher in Anführungsstriche zu setzen. Möchte man stattdessen Variablen verwenden, steht dem nichts im Wege.

Jedem Menüpunkt wird ein Shortcut zugeordnet, für "Neu" z.B. steht das "%n". Hinter der Schreibweise mit dem Prozentzeichen verbirgt sich die Abkürzung für die Funktion *ASC("n")*, die den ASCII-Wert des Buchstaben "n" ermittelt, hier also: 110. Obwohl sich hinter "ASCII" eine Liste der standardmäßig verfügbaren Zeichen auf einem Computer verbirgt, gilt sie streng genommen nur für die ersten 127 Werte. Die verschiedenen Hersteller füllen den Freiraum bis zur Nr. 255 meist in unterschiedlicher Weise.

In der Menüdefinition ließe sich ohne weiteres auch die konkrete Zahl einsetzen, anschaulicher ist aber die gewählte Schreibweise.

Durch die angegebenen Programmzeilen öffnet sich zunächst das Menü. Man kann darin beliebig wie gewohnt mit Stift oder Tastatur navigieren. Aber erst, wenn der Stift auf einen Menüpunkt tippt oder wenn man <Enter> drückt, wird das Ergebnis der "Operation Menü" in die Variable *m%* übernommen. In *m%* findet sich genau einer der Werte wieder, die durch die Shortcuts vorbestimmt wurden. Ausnahme: der Abbruch durch <Esc> oder ein Stift-Tipp auf einen Menü-fremden Bereich liefert in *m%* den Wert Null zurück.



## Komplettbeispiel

Vollständiges Beispiel mit Verarbeitung der Shortcuts:

```

PROC Menu1:
LOCAL m%, taste%, mod%
LOCAL mklein$(16), mgross$(16), jump$(32)
mklein$="oe"          REM Shortcut-Buchstaben
mgross$="HA"
PRINT "<Strg><E> beendet"
WHILE 1
  taste%= GET
  mod%= KMOD          REM Modifizierertaste
  IF taste%= 290      REM Menü-Taste
    mINIT
      mCARD "Datei", "Öffnen", %o, "Beenden", %e
      mCARD "Extras", "Hilfe", %H, "Über ..", %A
    m%=MENU
    IF m%<=%Z        REM Grossbuchstaben-Shortcut
      IF LOC(mgross$, CHR$(m%)) <> 0
        jump$="Sub_G_" + CHR$(m%)
        @(jump$):
      ENDIF
    ELSE             REM Kleinbuchstaben-Shortcut
      IF LOC(mklein$, CHR$(m%)) <> 0
        jump$="Sub_k_" + CHR$(m%)
        @(jump$):
      ENDIF
    ENDIF
  ELSEIF (mod% AND 4) REM Strg gedrückt
    REM Shortcuts direkt nutzen
    taste%=taste% + 64
    IF mod% AND 2     REM Shift gedrückt
      IF LOC(mgross$, CHR$(taste%)) <> 0
        jump$="Sub_G_" + CHR$(taste%)
        @(jump$):
      ENDIF
    ELSE
      IF LOC(mklein$, CHR$(taste%)) <> 0
        jump$="Sub_k_" + CHR$(taste%)
        @(jump$):
      ENDIF
    ENDIF
  ELSE
    REM alle and.Tasten/Kombinat. landen hier; hier steht der zugehoerige
    REM Programmtext, z.B.: STOP oder RETURN taste% oder anderes
  ENDIF
ENDWH
ENDP

PROC Sub_k_e:
  STOP REM Programmende
ENDP

PROC Sub_k_o:
  giPRINT "Menüpunkt 'Öffnen' gewählt",0
ENDP

PROC Sub_G_A:
  giPRINT "Menüpunkt 'Über..' gewählt",0
ENDP

PROC Sub_G_H:
  giPRINT "Menüpunkt 'Hilfe' gewählt",0
ENDP

```

Das vorliegende Beispiel verwendet Shortcuts mit Klein- und Großbuchstaben. Zur Prüfung, ob ein Buchstabe auch tatsächlich ein Shortcut, also eine echte und erlaubte Menü-Auswahl ist, wird *LOC* verwendet. *LOC* arbeitet jedoch nicht "case sensitive", unterscheidet also nicht zwischen Groß- und Kleinschreibung. Daher müssen zwei Vergleichsstrings angelegt werden. Der eine enthält die im Menü verwendeten Großbuchstaben (*mgross\$*), der andere die kleinen (*mklein\$*).

Sehen wir uns den Zweig *IF taste%=290* an.

Das Menü wird in üblicher Weise aufgebaut, in *m%* steht nach der Auswahl der Rückgabewert. Die *IF*-Analyse von *m%* entscheidet, welcher der Strings zur Gültigkeitsprüfung herangezogen wird. War die Auswahl korrekt, wird das jeweilige Unterprogramm aufgerufen. Der Name der aufzurufenden Prozedur wird aus einem selbst zu benennenden Präfix (*Sub\_G\_*, *Sub-k\_*) und dem Auswahl-Buchstaben gebildet. Der sorgt schließlich für den Aufruf.

Menüs lassen sich aber auch wieder schließen, ohne eine Auswahl zu treffen. Dazu drückt man die <ESC>-Taste oder tippt mit dem Stift in einen anderen Bildschirmbereich. In diesem Falle landet der Wert Null in *m%*. Diese Nicht- Auswahl wird automatisch durch den Zweig *IF m%<=%Z* erfasst. Da dem Ergebnis von *CHR\$(0)* kein Zeichen in dem String *mgross\$* zuzuordnen ist, wird das Menü ohne Reaktion verlassen.

Handelt es sich bei dem Tastendruck um einen Shortcut (Zweig: *ELSEIF mod% AND 4*), muss zunächst der Buchstabenwert wieder hergestellt werden. Grund dafür ist, dass beim Drücken von <Strg> und einer Buchstabentaste nicht mehr der Zeichencode in *taste%* aufgenommen wird, sondern die Ordnungszahl des Buchstabens im Alphabet. Die interne Bildungsregel

```
buchstabenwert% AND NOT(60)
```

sorgt beispielsweise dafür, dass sowohl "A" als auch "a" den Wert "1" ergeben. Groß- und Kleinschreibung werden also nicht unterschieden. Hier muss man *KMOD*, in unserem Beispiel also *mod%*, zusätzlich heranziehen.

Damit aber wieder ein Prozedurname geformt werden kann, wird der aktuelle *taste%*-Wert in einen echten Tastaturcode zurückverwandelt: *taste%=taste% + 64* (siehe Zeichencode-Tabelle im Anhang).

## Kaskaden

OPL unterstützt auch kaskadierte Menüs.

Kaskaden verwendet man für eine weitere Aufgliederung der Menüs. Ihre Definition geschieht nach dem Schlüsselwort *mCASC* und muss erfolgt sein, bevor sie in *mCARD* benutzt werden.

```
mINIT
  mCASC "Ausrichtung", "links",%l,"rechts",%r
  mCARD "Text", "Ausrichtung",>,%A, "Zeilenabstand", %N
m%= MENU
```

In der *mCARD*-Zeile wird der Menüpunkt, hinter dem sich noch ein Untermenü versteckt, durch eine spitze Klammer (>) markiert. Der Name dieses Menüpunktes findet sich als (nicht gezeigter) Titel in der *mCASC*-Zeile wieder. Der restliche Teil wird wie bei *mCARD* gehandhabt.

Menüpunkte der Kaskade lassen sich auch noch weiter kaskadieren, doch vermindert sich dabei die Bedienfreundlichkeit – man sehe also besser davon ab.

Gerade in Kaskaden werden oft Optionslisten angeboten, in denen man einfach auswählen kann. Die Verwendung von Shortcuts ist dann nicht besonders sinnvoll. Deshalb lassen sich auch Menüeinträge ohne Shortcuts darstellen. Dazu benutzt man Werte von 1 .. 32 anstelle der Buchstaben.

## Spielregeln

Bei der Auswahl der Shortcut-Buchstaben orientieren Sie sich am besten an den eingebauten Menüs. Es gibt zwar keine ausdrücklichen Festlegungen, welche Buchstaben für welchen Einsatzzweck zu vergeben sind, aber der Benutzer wird es Ihnen danken, wenn er sich in Ihrem Programm auf Anhieb zurechtfindet. Insbesondere achten Sie aber darauf, dass das Programm mit dem Menüpunkt "Beenden"/"Exit" oder dem Shortcut <Strg><E> verlassen werden kann. Weitere gängige Shortcuts:

<Strg><N> - Neu ...

<Strg><O> - Öffnen

<Strg><S> - Speichern

<Strg><Shift><S> - Speichern als

<Strg><P> - Drucken

<Strg><J> - Toolbar an/aus

Auch bei den Titeln der Menüs richte man sich nach den bekannten Vorbildern. Links wird es fast immer ein Hauptmenü "Datei" geben, gefolgt von "Bearbeiten" und "Ansicht". Rechts findet man überwiegend das "Extras"-Menü, in dem solche Punkte wie "Hilfe" und "Über" (Programm und Autor) untergebracht sind.

## Optionen

Wenn Sie sich vorhandene Menüs ganz bewusst ansehen, werden Sie weitere optische und funktionelle Einzelheiten entdecken, die sich auch unter OPL nachbilden lassen.

So zieht man unter bestimmte Menüpunkte abtrennende Linien, indem man die Shortcutangabe mit einem Minuszeichen versieht:

```
mCARD "Datei", "Speichern", -%s, "Beenden", %e
```

Einträge werden grau dargestellt, indem man den in der untenstehenden Tabelle genannten Wert zum Shortcutwert addiert oder ODERiert. Man nutzt diese Darstellung, um zu signalisieren, dass das Menüelement zwar vorhanden ist, im Moment aber nicht genutzt werden kann. Tatsächlich ist ein grau dargestelltes Menüelement nicht auswählbar.

```
mCARD "Datei", "Datei öffnen", %o + $1000, "Beenden", %e
mCARD "Datei", "Datei öffnen", %o OR $1000, "Beenden", %e
```

Bei derartigen Manipulationen wird gern die hexadezimale Zahlendarstellung verwendet, weil sie dem Geübten übersichtlicher ist. Siehe im Teil 2 das Anhangs-Kapitel "Binär- und Hexzahlen".

Menüeinträge lassen sich mit einem Häkchen markieren, wenn man "Checkboxes" verwendet. So kann man sehen, ob eine Einstellung aktiviert oder deaktiviert ist. Aber Achtung! Durch die "Steuerwerte" wird nur das optische Erscheinungsbild des Menüs beeinflusst, die Verwaltung der dahinterstehenden logischen Funktion müssen Sie selbst organisieren!

Das gleiche gilt für die Optionsbuttons, die für die Einstellung sich ausschließender Bedingungen zuständig sind.

Steuerwert	Eigenschaft
\$1000	Eintrag grau
\$0800	Eintrag hat Checkbox
\$0900	Kennzeichnung für ersten Optionsbutton
\$0A00	Kennz.f.mittl. Optionsbutton (mehrere mögl.)
\$0B00	Kennzeichnung für letzten Optionsbutton
\$2000	Symbol an (Checkbox/Optionsbutton)
\$4000	Symbol unbestimmt

Beispiel:

PROC Checkbox:

```

...
mINIT
  mCARD "Option","Tbar",%j OR $0800 OR (k%*$2000)
m%= MENU
IF m%= %j
  Wechselschalter:
ENDIF
...
ENDP

```

PROC Wechselschalter:

```

IF k%
  k%= 0
  REM von hier aus auch Aufruf eines Unter-
  REM programms zum Ausschalten des Toolbars
ELSE
  k%= 1
  REM von hier aus auch Aufruf eines Unter-
  REM programms zum Einschalten des Toolbars
ENDIF
ENDP

```

## PopUp-Menüs

Unabhängig vom Standardmenü stellt OPL eine weitere Menüart zur Verfügung: die PopUp-Menüs. Dabei handelt es sich um frei platzierbare Menüs, Beispiel:

```
m%=mPOPUP (x%,y%,bezug%,"DEM",%m,"Euro",%u)
```

Diese Menü-Anweisung muss und soll nicht in eine *mINIT ... MENU*-Struktur eingebettet sein. Die Variablen *x%* und *y%* liefern die pixelgenauen Koordinaten für die Anordnung, *bezug%* bezeichnet den Eckpunkt des Menüs, auf den sich die *x%/ y%*-Koordinaten beziehen. Es gilt:

bezug%	Ort
0	linke obere Menüecke
1	rechte obere Menüecke
2	linke untere Menüecke
3	rechte untere Menüecke

Alle oben genannten Parameter zur Veränderung der Ansicht und Funktion gelten auch für die PopUp-Menüs, Ausnahme: sie können nicht kaskadiert werden. Das Ergebnis wird wie bei *MENU* in eine Variable (*m%*) zurückgeliefert, ein Abbruch ergibt Null.

## 8 Dialoge

---

### Das Prinzip

Dialoge erhöhen die Bequemlichkeit der Programmbedienung. Anfangs mussten wir unsere Daten mühsam über *INPUT* und *EDIT* eingeben und mit *PRINT* sichtbar machen. Mit Hilfe von Dialogen erhalten wir nun wohlgeformte Ansichten, um die Eingabe auf elegantere Art zu bewerkstelligen.

Wie auch die Menüs verlangen Dialoge einen formellen Aufbau. Sie werden mit *dINIT* eingeleitet und enden mit *DIALOG*. Allein diese beiden Befehle reichen aus, um eine leere Dialogbox auf die Bildschirmmitte zu zaubern, die sich mit dem Stift verschieben lässt.

Zwischen den beiden Anweisungen stehen in gewünschter Reihenfolge und in der Anzahl nur durch den Bildschirm begrenzte Dialog-Elemente.

Alle Elemente benötigen wenigstens zwei Parameter. Der eine enthält einen Anzeigetext, der andere ist eine globale oder lokale Variable. Besitzt die Variable bereits einen Wert, wird dieser ebenfalls angezeigt und dient gleichzeitig zur Aufnahme des Rückgabewertes.

Einige Dialog-Elemente erfordern aber auch mehr als nur die beiden Parameter.

### Textausgabe

Das einzige nur-Ausgabe-Dialogelement ist *dTEXT*, daher wird auch keine Rückgabefunktion benötigt.

```
dINIT
  dTEXT betreff$,text$
d%=DIALOG
```

Der Inhalt von *betreff\$* wird linksseitig dargestellt, *text\$* gleich rechts daneben. Wie angedeutet, darf man hier Textvariablen benutzen, kann aber durchaus Strings auch direkt verwenden:

```
dTEXT "Name: ", "Hannes Meier"
```

Während *betreff\$* auch leer sein kann, muss *text\$* wenigstens ein Leerzeichen enthalten. Zusätzlich bestehen Formatierungsmöglichkeiten durch einen dritten Parameter:

```
dTEXT betreff$,text$,format%
```

Zur Textausrichtung (nur wenn *betreff\$* ein Leerstring ist) kann *format%* folgende Werte annehmen:

format%	Ausrichtung
0	linksbündig (Standardwert)
1	rechtsbündig
2	zentriert

Weitere Effekte:

format%	Effekt
\$200	zieht eine Linie unter dieses Element
\$400	der Betreff wird selektierbar
\$800	Zeile wird zum Textseparator

Unter bestimmten Umständen lässt sich so der Text-Dialog auch als Menü definieren, denn die ausgewählte Zeilennummer wird nach `d%` zurückgegeben.

```
titel$= "Auswahl"
dINIT titel$
  REM dTEXT betreff$,text$,format%
  dTEXT "1)","Datei öffnen", $400
  dTEXT "2)","Datei speichern", $400
  dTEXT "3)","Beenden", $400
d%=DIALOG
```

Die Auswahl von Punkt "3)" mit dem Stift oder der Tastatur liefert in `d%` eine drei zurück, wenn der Dialog mit `<Enter>` abgeschlossen wird, während `<Esc>` eine Null in `d%` bewirkt. Das ganze funktioniert nur, wenn der Betreff kein Leerstring ist, denn nur dann kann die Zeile überhaupt angewählt werden.

Dem Dialog wurde diesmal ein Titel mitgegeben, der in der Dialogbox im oberen grauen Bereich dargestellt wird. Das allgemeine Aussehen einer Dialogbox ist noch weiter beeinflussbar (die Button-Parameter besprechen wir später):

```
dINIT titel$,format%
```

format%	Wirkung
1	Buttons rechts statt unten
2	Titel wird nicht angezeigt
4	ganzen Bildschirm ausnutzen
8	Box kann nicht verschoben werden
16	Dialog dicht packen, Zeilen gewinnen, keine Buttons

Fasst man die Dialogbox mit dem Stift im oberen grauen Bereich an, kann man sie über den Bildschirm verschieben. Möchte man das verhindern, muss man die Titelanzeige unterdrücken (`format%=2`) oder die Verschiebemöglichkeit sperren (`format%=8`).

Die einzelnen Formatwerte dürfen kombiniert werden.

Wer seine Dialogbox anders platzieren möchte, verwende die Positionierungsanweisung `dPOSITION` irgendwo zwischen `dINIT` und `DIALOG`:

```
dPOSITION x%,y%
```

Die Positionierung erfolgt nur grob gegliedert:

x%= -1	links
x%= 0	Mitte
x%= 1	rechts
y%= -1	oben
y%= 0	Mitte
y%= 1	unten

Die folgenden Dialogelemente widmen sich jeweils spezifischen Datentypen.

## Zahleneingabe

Für Fließkommazahlen und Integers werden getrennte Eingabeanweisungen benutzt:

```
dLONG longint&,betreff$,min&,max&
dFLOAT fliessk,betreff$,min,max
```

Die Rückgabevariablen (longint&, fließsk) sind den Zahlentypen angepasst, ebenso die beiden Parameter, die die Ober- und Untergrenze des Eingabebereiches festlegen (min&, max&,min,max). Ein separater Dialog für normale Integers existiert nicht, hier muss bei Bedarf eine Typumwandlung vorgenommen werden. Durch die Gestaltung von min& und max& kann man im Vorfeld dafür sorgen, dass der entsprechende Zahlenbereich gar nicht erst überschritten wird.

## Datums- und Zeiteingaben

Zur Datumseingabe steht die Anweisung

```
dDATE datum&,betreff$,min&,max&
```

zur Verfügung. Das eingetragene Datum wird als "Tage seit dem 1.1.1900" in die Variable zurückgegeben. Auch die Grenzwerte rechnen in dieser Form. Der 1.1.2100 wäre dann datum&= 73049 und der Wert von max& entsprechend anzupassen.

Ein auf "heute" voreingestelltes Datum bekommt man, wenn man DAY, MONTH und YEAR zur Hilfe nimmt, vorausgesetzt, die Systemzeit auf dem Gerät ist korrekt eingestellt.

```
PROC Dialog1:
  LOCAL d%, min&, max&, datum&, tag%, monat%, jahr%
  datum&= DAYS(DAY,MONTH,YEAR) REM heute
  max&= DAYS(1,1,2100)
  min&= datum&
  dINIT "Datums-Dialog"
  dDATE datum&,"Datum",min&,max&

  IF DIALOG
    REM Rueckgabewerte ausgeben:
    PRINT datum&
    REM lesbare Rueckwandlung der Eingabe
    DAYSTODATE datum&,jahr%,monat%,tag%
    PRINT tag%;".";monat%;".";jahr%
    GET
  ENDIF
ENDP
```

Für die nicht besprochenen Anweisungen s. alphabetisches Befehlsverzeichnis.

Eine Editierbox für Zeiteingaben wird auf diese Art definiert:

```
dTIME zeit&,betreff$,format%,min&,max&
```

Die Variable zeit& enthält nach Eingabe und Beendigung der Dialogbox die Zeit als "Sekunden seit Mitternacht". Der Parameter format% sorgt für die Anzeigeweise:

format%	Anzeige
0	abs. Zeit ohne Sekunden
1	abs. Zeit mit Sekunden
2	Zeitdauer ohne Sekunden
3	Zeitdauer mit Sekunden
4	abs. Zeit in Minuten innerhalb der Stunde
8	24h-Uhr, nur interessant für Systeme mit am/pm-Darstellung

Was da eigentlich im Dialog angezeigt und abgefragt wird, teilt man dem Nutzer sinnvollerweise über `betreff$` mit.

```
PROC Dialog2:
  LOCAL zeit&, min&, max&, format%
  LOCAL stunden%, minuten%, sekunden%
  zeit&= HOUR*3600.0 + MINUTE*60+ SECOND  REM jetzt
  max&= 23*3600.0 + 59*60 + 59
  min&= 0
  format%=3
  dINIT "Zeit-Dialog"
    dTIME zeit&,"Zeit",format%,min&,max&
  IF DIALOG
    stunden%=INT(zeit&/3600)
    minuten%=INT((zeit&-3600.0*stunden%)/60)
    sekunden%=INT(zeit&-3600.0*stunden%-60.0*minuten%)
    PRINT "Rückgabewert:",zeit&  REM Sekunden
    PRINT "Zeit:",stunden%;": ";minuten%;": ";sekunden%
  ENDIF
  GET
ENDP
```

Man informiere sich in diesem Zusammenhang auch über *DATETOSEC* und *SECTODATE* im alphabetischen Befehlsverzeichnis.

## Stringeingaben

Darunter fallen einfache Texteingaben ähnlich *EDIT*, verdeckte Eingaben und die Eingabe von Dateinamen. Die Dialogelemente definiert man so:

```
dEDIT text$,betreff$ [,laenge%]
```

Mit dem optionalen Parameter `laenge%` bestimmt man die Weite des Editierfeldes.

```
dXINPUT text$,betreff$
```

Der Text wird verdeckt dargestellt (Sternchen), so dass sich diese Eingabeform insbesondere für Passwörter eignet.

```
dFILE dateiname$,betreff$,format% [,uid1&,uid2,uid&]
```

Der Parameter `format%` steuert die Nutzungsart.

Weil im *dFILE*-Dialog automatisch auch eine Laufwerks- und Ordnerauswahl angeboten wird, sollte auch `betreff$` drei anstatt nur einer Bezeichnung enthalten:

```
dFILE dateiname$, "Datei,Ordner,Laufwerk",format%
```

Weiteres siehe in der alphabetischen Befehlsliste.

## Aus einer Liste wählen

Eine Auswahl ist eine bequeme Methode, um auf eine Liste vordefinierter Begriffe zurückzugreifen. In OPL wird das über *dCHOICE* organisiert.

```
dCHOICE wahl$,betreff$,liste$
```

Ein Beispiel:



```

PROC Auswahl:
  LOCAL wahl%, d%
  wahl%= 2
  dINIT "Zoom einstellen"
  dCHOICE wahl%, "Faktor:", "2x, 3x, 4x"
  IF DIALOG
    IF wahl%=1
      ...
    ELSEIF wahl%=2
      ...
    ELSE
      ...
    ENDIF
  ENDIF
ENDP

```

Das durch wahl% angegebene Listenelement wird im Dialog angezeigt, hier also "3x". Nach getroffener Auswahl wird die neu selektierte Elementnummer nach wahl% zurückgegeben und kann geeignet verarbeitet werden. Sollte die Auswahlliste sehr lang sein, gibt es Vorschriften zur Auftrennung in mehrere Zeilen – s. alphabetische Befehlsliste.

## Auswahl per Checkbox

Für eine einfache Auswahl ("ja/nein", "an/aus") eignen sich Checkboxes – das sind diese kleinen Kästchen mit oder ohne Häkchen – sehr gut.

```

dCHECKBOX schalter%,betreff$
dCHECKBOX toolbar%,"Toolbar an"
dCHECKBOX farbe%,"Farbe an"

```

Die Integer-Variablen nehmen die Werte TRUE oder FALSE an, je nachdem, ob die Checkbox angehakt wurde oder nicht. Werden die Variablen vor dem Dialog auf TRUE gesetzt (z.B. farbe%= -1), wird das Häkchen angezeigt. Für die entsprechenden Veränderungen im Hintergrund sind Sie natürlich selbst verantwortlich.

## Nutzerfreundliche Dialoge

Normalerweise werden Dialoge durch <Enter> oder <Esc> beendet, Stiftbedienung ist zunächst nicht vorgesehen. Hier schafft *dBUTTONS* Abhilfe:

```

dBUTTONS text1$,taste1% [,text2$,taste2%,text3$,taste3%]

```

Durch *dBUTTONS* werden am unteren Dialogrand einer oder mehrere Buttons angezeigt, die auch auf Stiftdruck reagieren. Die Parameterpaare (text\$,taste%) bestimmen deren Aussehen und Funktionalität. Der Inhalt des Strings wird auf dem Button dargestellt, die Integervariable sagt, welche Taste für diesen Button zuständig ist. Der Tastenwert wird als Buchstabe unter dem Button dargestellt. In Kombination von <Strg> und <Taste> wird der Button dann ausgelöst, sofern er nicht mit Hilfe des Stiftes bedient wird. Gehört der Tastaturcode zu einer Spezialtaste, wird deren Name automatisch angezeigt (13=Enter, 9=Tab, 32=Leertaste, 27=Esc).

Die *dBUTTONS*-Anweisung darf beliebig zwischen *dINIT* und *DIALOG* angeordnet sein.

```

dINIT
  dEDIT name$, "Name:"
  dXINPUT passwort$, "Passwort"
  dBUTTONS "Abbruch", 27, "OK", 13
d%= DIALOG
...

```

Bei einem Abbruch mit <Esc> werden die Eingabewerte nicht in die Variablen übernommen, evtl. vorher vorhandene Werte bleiben erhalten. In allen anderen Fällen werden die Eingaben übernommen und in d% steht der zur ausgewählten Taste definierte Code, im Falle von <Esc> der Wert Null.

Sollen andere Tasten anstelle von <Esc> als Abbruchtaste reagieren, müssen der Tastaturwert und mögliche Modifikatorwerte negativ angegeben werden:

```
dBUTTONS "Abbruch", - (%a+512) , "OK" , 13
```

Hier bewirkt der Modifikatorwert "512", dass man bei Tastenbedienung nicht wie sonst üblich die Strg-Taste zusätzlich halten muss, um den Button auszulösen – der Druck auf die Buchstabentaste reicht aus.

In Abhängigkeit von dINIT und seinen Formatierungsparametern lassen sich die Buttons auch am rechten Rand der Dialogbox darstellen. Mehr in der alphabetischen Befehlsliste.

## Alarm!

Mit dem Befehl *ALERT* kann eine bis zu zwei Zeilen umfassende Mitteilung ausgegeben werden. *ALERT* definiert eine Dialogbox, die ohne die formellen Anweisungen *dINIT* und *DIALOG* auskommt. S. alphabetische Befehlsliste.

## 9 Einfache Grafik

---

### Cursor positionieren

Bisher haben Sie den Bildschirm nur als Wiedergabemedium für Text und Zahlen kennengelernt. Für die Belange dieser Darstellung reichte die Einteilung des Bildschirmes in Zeilen und Spalten völlig aus. Aber natürlich wissen Sie, dass die Anzeige auf der Ansteuerung einzelner, feingerasterter Punkte beruht. Für ein einfaches *PRINT "F"* wird am Serie 5 ein Bereich von 6\*8 Pixel (Abk. für picture cell, Bildpunkt)) zur Darstellung des Buchstaben verwendet.

Mit Grafikkommandos lässt sich nun jeder einzelne Punkt auf dem Bildschirm aktivieren, ein pixelgenaues Arbeiten wird möglich. Das trifft auch auf "grafische Schrift" zu, zunächst werfen wir aber einen Blick auf die Zeichenbefehle. Man erkennt sie bereits von weitem an dem vor jeden Befehl gestellten Buchstaben "g".

Bevor ein grafisches Element gezeichnet werden kann, muss sein Anfangspunkt festgelegt werden:

```
gAT xpos%, ypos%
```

Die absoluten Positionierungsangaben in x- bzw. y-Richtung werden in Pixel angegeben. Der Ursprung des Koordinatensystems liegt in der linken oberen Ecke des Bildschirmes. Es ist der Punkt 0,0.

Je nachdem, welches Gerät Sie besitzen, stehen Ihnen folgende Ausdehnungen zur Verfügung (Breite\*Höhe)

Serie 5xx: 640\*240 (halb-VGA)

Revo: 480\*160

netBook/S7 640\*480 (VGA)

Am Serie 5 ist also die linke untere Ecke durch die Koordinaten 0,239, die rechte obere Ecke durch 639,0 beschrieben. Es gibt keine Fehlermeldung, wenn Sie mit *gAT* in einen Bereich außerhalb dieser Maße positionieren. So ist z.B. auch die Angabe

```
gAT -10, -10
```

statthaft, was der nicht sichtbaren Position 10 Pixel links und oberhalb vom Ursprung entspricht.

Die *gAT*-Anweisung arbeitet absolut. Jedesmal, wenn sie ausgeführt wird, setzt sich ein nicht sichtbarer Cursor wie eine Markierungsfahne beim Golf auf die genannte Position. Der nächste Zeichenbefehl bezieht sich darauf. Beim Programmstart steht der Cursor in der Grundposition 0,0. Wollen Sie den Cursor anzeigen lassen, reicht der Befehl:

```
CURSOR 1
```

Näheres finden Sie in der alphabetischen Befehlsliste im Teil 2 des Buches.

Neben der absoluten Positionierung kennt OPL auch eine relative. Ausgehend von der aktuellen Position, wird der Cursor um einen bestimmten Pixelbetrag nach rechts (dx%) und unten (dy%) verschoben, ohne auf dem Bildschirm eine Spur zu hinterlassen.

```
gMOVE dx%,dy%
```

Negative Parameterwerte verschieben nach links (-dx%) und oben (-dy%). Überschreiten des Bildschirmrandes verursacht auch hier keine Fehlermeldung.

## Geometrische Elemente

Nach dem Festlegen des Startpunktes können wir den grafischen Befehl angeben – hier geht es um eine Linie:

```
gLINETO x%,y%
```

Die Parameter benennen den Endpunkt der Linie in absoluten Werten. Auch von diesem Befehl existiert eine relativ-Variante.:

```
gLINEBY dx%,dy%
```

Für die Wirkung von Vorzeichen gilt das oben gesagte.

In beiden Fällen steht der Cursor nach Ausführung der Anweisung am Endpunkt der Linie.

Die relative Positionierung hat gewisse Vorteile. Wenn Sie eine Figur per Unterprogramm zeichnen, können Sie diese an einer anderen Cursorposition identisch zeichnen lassen, ohne die Koordinaten neu bestimmen und einsetzen zu müssen. Lediglich der Startpunkt ist neu festzulegen.

Das folgende Beispiel zeichnet zwei unterbrochene Linien:

```
PROC Grafik_1:
  gAT 20,20 : Linie:
  gAT 20,40 : Linie:
  GET
ENDP

PROC Linie:
  gLINEBY 100,0
  gMOVE 20,0
  gLINEBY 100,0
ENDP
```

Das einfachste grafische Element ist ein Punkt. In OPL wird er als Sonderfall des Linie-Befehls betrachtet. Um einen Punkt zu zeichnen, lassen Sie die Linie einfach "auf der Stelle treten":

```
PROC Grafik_2:
  PRINT gX,gY
  gAT 20,20
  gLINEBY 0,0
  PRINT gX,gY
  GET
ENDP
```

Die Position des Cursors wird dabei nicht verschoben, den Beweis bekommt man mit den Funktionen *gX* und *gY*, die die aktuellen Cursor-Koordinaten enthalten. Halten Sie vorsichtshalber eine Lupe bereit, wenn Sie den Punkt aus dem Beispiel zu Gesicht bekommen wollen ...

Um eine Folge von Linien direkt aneinander zu zeichnen, existiert der Befehl *gPOLY*. Der ist allerdings so ungelenk zu handhaben, dass ich nur im alphabetischen Befehlsverzeichnis darauf eingehe. Für kleine Mehr-ecke tut es eine Eigenkonstruktion viel übersichtlicher. Beispiel: eine Viereck-Routine, mit der sich beliebig geformte geschlossene Vierecke zeichnen lassen:

```
PROC Grafik_3:
  gAT 50,50 : Viereck: (20,0,10,30,-40,0)
  gAT 100,50 : Viereck: (20,0,10,30,-20,0)
  GET
ENDP

PROC Viereck:(x1%,y1%,x2%,y2%,x3%,y3%)
  LOCAL xStart%,yStart%
  xStart%= gX : yStart%= gY REM akt. Cursorpos. merken
  gLINEBY x1%,y1% : gLINEBY x2%,y2%
  gLINEBY x3%,y3% : gLINETO xStart%,yStart%
ENDP
```

Noch einfacher erhält man rechtwinklige Vierecke, indem man den *gBOX*-Befehl benutzt:

```
gBOX b%,h%
```

Die aktuelle Cursorposition bildet die linke obere Ecke des Rechtecks, das sich in einer Breite *b%* und einer Höhe *h%* nach rechts unten aufspannt. Negative Parameterwerte wirken in die entgegengesetzte Richtung. Die Cursorposition wird bei Befehlsausführung nicht verlagert.

```
PROC Grafik_4:
  gAT 100,100
  gBOX 50,50
  gBOX 25,25
  PRINT gX,gY
  GET
ENDP
```

Ähnlich einfach zeichnen sich Kreise und Ellipsen:

```
gCIRCLE radius%
gELLIPSE horiz_radius%,vert_radius%
```

Die Parameter von *gELLIPSE* geben jeweils den Abstand zum horizontalen bzw. vertikalen Rand der Ellipse an, gemessen vom Mittelpunkt. Die Cursorposition bleibt erhalten.

## Elemente füllen

Die Darstellung der grafischen Grundelemente lässt sich großzügig beeinflussen. So ist unter anderem auch eine Füllung möglich:

```
gFILL b%,h%,modus% (anstelle von gBOX)
gPATT muster%,b%,h%,modus%
```

Der Parameter *modus%* bei *gFILL* beeinflusst, wie das Rechteck gefüllt wird:

```
modus%   Pixel werden:
0        gesetzt
1        gelöscht
2        invertiert
```

Die Farbe bzw. der Grauwert der Füllung wird durch die aktuelle Vordergrundfarbe, der Farbe des "Pinsels", bestimmt. Bei Start des Programmes wird automatisch mit schwarzer Farbe gezeichnet und auch gefüllt. Wie man andere Farben ins Spiel bringt, siehe weiter unten im Abschnitt "Farbe".

`gPATT` füllt einen rechteckigen Bereich mit einem Muster. Der Befehl kennt als weiteren Wert für `modus%` die Zahl 3. Während bei den `modus%`-Werten 0 .. 2 nur die gesetzten Pixel des Musters übertragen werden, bewirkt `modus%=3`, dass auch die nicht gesetzten (weißen) Pixel mit zur Füllung beitragen. Der EPOC-Computer verfügt über nur ein einziges intern vorbereitetes Muster, das mit dem Wert `muster%= -1` angesprochen wird. Andere Muster lassen sich beliebig erzeugen und verwenden. Das folgende Beispiel werden Sie erst voll interpretieren können, wenn Sie die Kapitel über "Fenster und Rahmen" und "Bitmaps" gelesen haben.

```
PROC Muster:
  LOCAL muster%
  muster%= gCREATEBIT(10,10)
  gAT 3,3 : gBOX 5,5
  gUSE 1
  gAT 10,10 : gPATT muster%,100,100,0
  GET
ENDP
```

Die Befehle für gefüllte Kreise und Ellipsen lauten:

```
gCIRCLE radius%,gefüllt%
gELLIPSE hradius%,vradius%,gefüllt%
```

Weicht der Parameterwert `gefüllt%` von Null ab, werden Kreis und Ellipse mit der aktuellen Zeichenfarbe (s.u.) gefüllt. Die Cursorposition wird auch hier nicht beeinflusst.

```
PROC Grafik_5:
  gAT 100,100
  gELLIPSE 60,10,1
  gFILL 50,50,2
  gPATT -1,25,25,3
  GET
ENDP
```

Wenn Sie erst mit dem Programmieren beginnen, werden Sie oft noch keine Verwendung für bestimmte Befehle haben – aber die Wünsche wechseln! In die Kategorie der "Keine Ahnung, wofür ich das mal brauche" fallen die folgenden Anweisungen.

Mit

```
gINVERT breite%,hoehe%
```

können Sie ab der Cursorposition einen durch `breite%` und `hoehe%` spezifizierten Bereich invertiert darstellen.

```
PROC Invert:
  gAT 50,50
  gFILL 50,50,0
  gAT 30,30
  gINVERT 50,50
  GET
ENDP
```

## Verschieben, kopieren, entfernen

Eine Lagekorrektur des Bildschirminhaltes (oder auch nur eines Teils davon) wird möglich durch

```
gSCROLL dx%,dy%
gSCROLL dx%,dy%,x%,y%,b%,h%
```

Wenn nur dx% und dy% gegeben sind, wird der gesamte Bildschirminhalt um die Differenzbeträge verschoben. Gibt man die Koordinaten und die Ausdehnung eines rechteckigen Bereiches zusätzlich an, wird nur dieser Bereich verschoben.

```
PROC Scroll:
  PRINT "Hallo"
  gAT 20,20 : gFILL 50,50,0
  GET : gSCROLL 50,50
  GET : gSCROLL -50,-50,50,50,50,40
  GET
ENDP
```

In die beschriebene Kategorie "wozu das denn?" fällt auch *gCOPY*, den ich erst erkläre, wenn die zugehörigen Voraussetzungen besprochen sind (Kapitel "Bitmaps"). Ein Beispiel können Sie jetzt schon nachvollziehen:

```
PROC Copy:
  LOCAL x%,y%,b%,h%,modus%
  PRINT "Hallo"
  gAT 20,20 : gFILL 50,50,0
  GET
  x%=0 : y%=0 : b%=70 : h%=70
  modus%=0
  gAT 100,0
  gCOPY 1,x%,y%,b%,h%,modus%
  GET
ENDP
```

Über die Befehle *gBORDER* und *gXBORDER*, die eigentlich ebenfalls in dieses Kapitel gehören, berichte ich im Kapitel "Fenster und Rahmen".

Zum *gBUTTON*-Befehl, der etwas Wissen zu Bitmaps voraussetzt, lesen Sie bitte im alphabetischen Inhaltsverzeichnis nach. Ebenso zu *gCLOCK* – mit diesem Befehl werden Uhren verschiedener Gestaltung dargestellt.

Wenn Teile einer Zeichnung nicht gelungen sind oder nicht mehr benötigt werden, haben die bisher beschriebenen Befehle genug Leistungsfähigkeit, um durch geschicktes Überschreiben bestimmte Bereiche wieder auszuradieren. Soll aber der gesamte Bildschirm gelöscht werden, erreicht man das durch den parameterlosen Befehl

```
gCLS
```

## Schnellere Grafik

Zu einer komplexen Grafik gehört zumeist eine stattliche Anzahl von Befehlen. Alle Zeichen- und Schreibbefehle werden unmittelbar ausgeführt, d.h., die Bildschirmansicht wird nach jedem Befehl neu aufgebaut – ein "update" wird ausgeführt. Bei mehreren Befehlen bedeutet das einen durchaus spürbaren Zeitverlust, der sich verkleinert, wenn man *gUPDATE* benutzt. Ein

```
gUPDATE OFF
```

verhindert, dass der Bildschirminhalt nach jedem Befehl aufgefrischt wird. Die Ausführung wird im Speicher gepuffert. Erst, wenn alle Grafik- und Schreibbefehle abgearbeitet wurden, erzwingt man einmalig die Erneuerung des Bildschirminhaltes durch

```
gUPDATE .
```

Der Puffer ist aber nicht unendlich groß. Ist er voll, wird der Bildschirminhalt regeneriert. Es tragen auch andere Befehle und das Betriebssystem dazu bei, dass eine Regenerierung auch zwischendurch stattfindet. Trotzdem bringt das Benutzen von *gUPDATE OFF* einen messbaren Zeitvorteil. Lassen Sie das folgende Programm mit und ohne *gUPDATE OFF* laufen:

```
PROC Update:
  LOCAL x%,y%
  x%=10 : y%=20
  REM gUPDATE OFF
  PRINT "Start:",MINUTE;";";SECOND,"  ",
  WHILE y%<(50)
    WHILE x%<(gWIDTH-10)
      gAT x%,y%
      gLINEBY 0,0
      x%=x%+1
    ENDWH
    x%=10 : y%=y%+1
  ENDWH
  gUPDATE
  PRINT "Ende:",MINUTE;";";SECOND
  gUPDATE ON
  GET
ENDP
```

Nach abgeschlossener Aktion stellt man immer den Originalzustand wieder her:

```
gUPDATE ON
```

## Farben

OPL arbeitet in der Grundeinstellung mit vier "Farben" (schwarz, weiß, 2 Graustufen) bei Geräten mit Grau-Bildschirmen und mit 256 Farben auf Farb-Bildschirmen. Will man mehr oder weniger Farben sehen, verwendet man gleich zu Programmstart den Befehl *DEFAULTWIN modus%*, dadurch wird die verwendbare Farbzahl des Basisfensters bestimmt. Die Parameter stehen für folgende Grauwert- bzw. Farbzahl:

```
modus%= 0      2 Grauwerte
modus%= 1      4 Grauwerte
modus%= 2      16 Grauwerte
modus%= 3     256 Grauwerte
modus%= 4      16 Farben
modus%= 5     256 Farben
```

Die ersten drei Werte ergeben auf Farbgeräten immer noch eine farbige Darstellung. Wie man sich das Programm am besten über die jeweiligen Farbfähigkeiten selber informieren lässt, lesen Sie am besten im alphabetischen Befehlsverzeichnis unter *gCOLORINFO* nach. Auch über *gINFO32* bekommen Sie entsprechende Informationen.

Achtung: Je mehr Farben erlaubt sind, desto größer ist der Stromverbrauch des Gerätes. Beschränken Sie sich im Modus am besten auf so wenig Farben wie nur irgend möglich.

Die aktive Farbe, mit der gezeichnet und gefüllt wird, kann mit Hilfe verschiedener Befehle festgelegt werden. Dabei ist *gGREY* nur im Vierfarbmodus brauchbar (s. alphabetische Befehlsliste), während *gCOLOR* eine gute Abstufung sowohl in Grautönen als auch in Farbe (für das netBook) zulässt.

Als Parameter werden Farbtonwerte zwischen 0 (schwarz) und 255 (weiß) mitgegeben:

```
gCOLOR rot%,gruen%,blau%
```



Sind die Werte von `rot%`, `gruen%` und `blau%` gleich groß, werden auch auf einem Farbbildschirm Grautöne angezeigt. Wird die Farbe nicht explizit gesetzt, ist die Zeichenfarbe schwarz.

```
PROC Grafik_6:
  LOCAL c%
  DEFAULTTWIN 2
  gFILL 288,50,0
  gBOX 288,100
  c%=0
  WHILE c%<256
    gCOLOR c%,c%,c%
    gAT c%+16,10
    gFILL 16,80,0
    c%=c%+16
  ENDWH
  GET
ENDP
```

Anstatt mit weiß zu zeichnen, ist es gelegentlich einfacher, die Art, wie die Pixel auf den Bildschirm geschrieben werden, zu ändern. Mit

```
gGMODE modus%
```

nimmt man Einfluss, ob alle nachfolgenden Grafikbefehle die Pixel setzen, also sichtbar machen (`modus%=0`), oder ob an derselben Stelle gelöscht werden soll (`modus%=1`). Dabei spielt es keine Rolle, welche Eigenschaft der Pixelpunkt bereits hat.

Spannend wird es mit `modus%=2`. Der Zeichenbefehl sieht sich das bereits Gezeichnete an und dreht jeden Punkt in seinem Farbwert um. Aus schwarz wird weiß, aus dunkelgrau hellgrau – und umgedreht. Insbesondere am Farbbildschirm des `netBooks` sind die Effekte manchmal recht unerwartet.

Befehle, die bereits einen Modus-Parameter verlangen, werden von `gGMODE` nicht beeinflusst. Das trifft z.B. auf `gFILL` zu.

Andere Einstellungen sind, offenbar ungewollt, in der Lage, die `gGMODE`-Anweisung auszuhebeln. Setzt man die Breite des imaginären Zeichenstiftes mit

```
gSETPENWIDTH linienbreite%
```

auf eine andere Breite als "1", funktioniert die Invertierung nicht korrekt. Probieren Sie es:

```
PROC Grafik_7:
  DEFAULTTWIN 2
  gCOLOR 64,64,64
  gFILL 100,100,0
  gGMODE 2          REM variieren!
  gSETPENWIDTH 5    REM variieren!
  gAT 25,25
  gBOX 50,50
  GET
ENDP
```

Bei einer `linienbreite%=0` wird naturgemäß gar nicht erst gezeichnet, obwohl das System keinen Fehler meldet.

# 10 Grafischer Text

---

## Text – pixelgenau gesetzt

Wie die Grafien selber ist auch grafischer Text auf den Pixel genau platzierbar. Das bietet Vorteile, der Aufwand jedoch, bis der Text in der gewollten Art an der richtigen Stelle auf dem Bildschirm steht, ist etwas größer.

So erfordert bereits die Platzierung des Textes Aufmerksamkeit. Positioniert wird wieder mit dem *gAT*-Befehl. Wissen muss man, dass das System die linke untere Ecke eines Buchstabenfeldes an die aktuelle Cursorposition setzt. "Buchstabenfeld" schließt die Extremmaße des gesamten Fonts mit ein, so dass "unten" die Kante der Unterlänge von Buchstaben wie dem "g" meint.

Nach dem Programmstart, bei dem der Grafik-Cursor bekanntlich an der Position 0,0 steht, bewirkt der folgende grafische PRINT-Befehl scheinbar gar nichts:

```
gPRINT "Hallo!"
```

In Wirklichkeit ist das Ergebnis nur nicht zu sehen. Bereits ein *gPRINT* "Guten Tag!" würde zumindest die Unterlänge des "g" hervorblitzen lassen. Besser wäre also:

```
gAT 20,20 : gPRINT "Hallo!"
```

Die *gPRINT*-Anweisung hat gegenüber dem Standard-*PRINT*-Befehl weitere Besonderheiten. So wird langer Text am rechten Bildschirmrand nicht mehr automatisch umgebrochen. Der Programmierer also ist selbst verantwortlich für die passende Aufbereitung des Textes. Welche Hilfsmittel man dazu einsetzt, werden wir später noch kennenlernen.

Anders als beim normalen *PRINT*-Befehl stellt ein erneuter *gPRINT*-Befehl den Cursor nicht in eine neue Zeile. Es wird dort weitergeschrieben, wo der Cursor aktuell steht. Wurde vorher bereits Text geschrieben, wird nahtlos weitergeschrieben. Ein einfaches Beispiel:

```
PROC Textgrafik_1:
  gAT 0,16
  gPRINT "Hallo "
  gLINEBY 20,0
  gPRINT "Welt!"
  GET
ENDP
```

Im Gegensatz zu normalem Text darf und kann bei der grafischen Variante jeder Buchstabe einzeln formatiert werden. Verschiedene, vor der Textausgabe voranzustellende Formatierungsbefehle nehmen Einfluss auf die Darstellung. Dazu gehört zunächst einmal die Auswahl eines Fonts:

```
gFONT fontnummer&
```

Die zur Verfügung stehenden Fonts, identifiziert durch eine Longintegerzahl, habe ich im Anhang gelistet. So wird z.B. der 13 Punkt große Font "Aria" als fette Schrift eingestellt:

```
gFONT 268435953, oder auch
font&= 268435953 : gFONT font&
```

Darüber hinaus bestimmt im wesentlichen der Stil die Text-Ausgabe:

```
gSTYLE stil%
```

Diese Effekte sind möglich:

stil%	Schriftstil
0	normal (Standard)
1	fett
2	unterstrichen
4	invers
8	doppelte Höhe
16	gleicher Zeichenabstand
32	kursiv

Jede Stilart wird durch ein einzelnes Bit repräsentiert, die Addition der Werte zwecks Kombination verschiedener Stile ist daher erlaubt und gegebenenfalls sogar erforderlich.

```
stil%= 1+32 : gSTYLE stil%
```

Im Beispiel wird die aktuelle Schrift fett und kursiv dargestellt. Hat man bereits einen fetten Font gewählt, wird dieser noch um eine Stufe fetter abgebildet.

Der dritte einflussnehmende Befehl bestimmt den Schreibmodus:

```
gTMODE modus%
```

Mit ihm wird für alle nachfolgenden grafischen Textausgabebefehle festgelegt, wie die einzelnen Pixel der Buchstaben auf dem Bildschirm dargestellt werden. Damit ist dieser Befehl das Pendant zum *gGMode*-Befehl, der für den Rest der Grafik zuständig ist. Es gilt:

modus%	Pixel werden ...
0	gesetzt
1	gelöscht
2	invertiert
3	ersetzt

Ist `modus% = 3`, werden alle Pixel eines Buchstabenfeldes, also "zeichnende" (schwarze) als auch "nicht-zeichnende leere" (weiße) gesetzt. In der Konsequenz werden unterliegender Text und unterliegende Grafik einfach überschrieben.

Damit sind die Möglichkeiten zur Textausgabe nicht erschöpft. Gerade hier haben sich die OPL-Entwickler richtig ausgetobt und eine Reihe weiterer Textausgabebefehle bereitgestellt.

In Verbindung mit anderer Grafik wird manchmal der Platz knapp. Meistens ist es unerwünscht, dass Text in eine bestehende Grafik schreibt und diese zerstört. Deshalb lässt sich die Textausgabe in der Länge begrenzen:

```
zeichenanzahl%= gPRINTCLIP(text$,weite%)
```

Wenn man die in Pixel angegebene Textbreiten-Vorgabe `weite%` überschreitet, wird der in String `text$` intelligent abgeschnitten. Das bedeutet, dass auch der letzte Buchstabe immer vollständig am Bildschirm erscheint. Die Anzahl der tatsächlich geschriebenen Zeichen wird nach `zeichenanzahl%` zurückgeliefert.

Nun gibt es durchaus Gründe, Text doch in eine bestehende Grafik laufen zu lassen. Gelegentlich schafft man sogar extra einen gefüllten Bereich, in dem die Schrift platziert werden soll. Wendet man dann den *gPRINT*-

Befehl an, verschwindet die Schrift im Untergrund, denn es werden nur die Pixel, die den eigentlichen Buchstaben ausmachen, gesetzt. Der folgende Befehl schafft sich deshalb vor der Textausgabe Platz und zeichnet erst eine leere Box und dann den Text:

```
gPRINTB text$,weite%
```

Der Parameter `weite%` begrenzt wie bei dem gerade zuvor behandelten Befehl die Ausgabebreite. Diese Anweisung ist aber weder in der Lage, die Anzahl der Zeichen zurückzugeben, noch gibt sie den letzten Buchstaben korrekt wieder. In ungünstigen Fällen ist er mitten drin abgeschnitten. Deshalb sollte man vor der Textausgabe die zu erwartende Pixel-Breite bestimmen und dann erst `gPRINTB` anwenden:

```
weite%=gTWIDTH(text$) : gPRINTB text$,weite% + 1
```

Überschreitet `weite%` ein im Einzelfall gerade erlaubtes Limit, muss man den Text eben kürzen oder anders darstellen.

Im Beispiel wurde bereits ein Pixel "Luft" zugegeben, trotzdem wirkt der ausgegebene Text noch immer ein-gequetscht. Um dieses Erscheinungsbild zu verbessern, stehen weitere Parameter zur Verfügung, die man Stück um Stück hinzufügen kann, die Reihenfolge jedoch ist einzuhalten:

```
gPRINTB text$,weite%, ausr%, oben%, unten%,rand%
```

Der Wert von `ausr%` bestimmt die Textausrichtung:

```
ausr%= 1      REM rechtebündig
ausr%= 2      REM linksbündig (Standard)
ausr%= 3      REM zentriert
```

Von dieser Einstellung hängt ab, wie die Randbreite in Pixel (`rand%`) interpretiert wird. Es entsteht ein leerer Raum links in der Breite der in `rand%` angegebenen Pixel, wenn `ausr%` auf "linksbündig" gesetzt ist, oder rechts, wenn "rechtsbündig" angewiesen wurde. Bei zentrierter Anordnung werden positive `rand%`-Werte dem Text links, negative rechts zugeschlagen. Die Parameter `oben%` und `unten%` bewirken Freiräume über und unter dem Text. Das Maß ist auch hier wieder "Pixel".

Sind die Anforderungen an die Textausgabe weniger kompliziert, kommt man auch mit einem einfacheren, aber doch wirksamen Befehl aus:

```
gXPRINT text$,modus%
```

Dabei hat `modus%` folgende Auswirkungen:

modus%	Textdarstellung
0	ähnlich wie bei <code>gPRINT</code>
1	invers
2	invers, Ecken gerundet
3	invers, schmal
4	invers, schmal, Ecken gerundet
5	echt unterstrichen
6	echt unterstrichen, schmal

Anders als bei `gPRINT` bewirkt der Befehl die Ausgabe sowohl der zum Buchstabenfeld gehörigen schwarzen als auch der "leeren" (weißen) Pixel.

Für `modus%= 0, 1, 2` und `4` verschafft zusätzlich ein 1 Pixel breiter Rahmen in Hintergrundfarbe etwas optischen Abstand zum nächsten Objekt. Damit ist die Voraussetzung für eine gute Lesbarkeit des Textes ge-

schaffen. In den "schmalen" Varianten fällt dieser Abstand weg. Beim Modus "Unterstreichen, schmal" rückt der Unterstrich noch um einen Pixel weiter an den Text heran, touchiert als "echter" Unterstrich aber höchstens die Unterlänge, während die Standard-Unterstreichung (*gSTYLE 2*) die Unterlänge schneidet.

Durch *gXPRINT* werden die eingestellten Schriftstile nicht außer Kraft gesetzt, daher erreicht man mit

```
gSTYLE 2 : gXPRINT text$,5
```

sogar einen doppelten Unterstrich.

```
PROC Textgrafik_2:
  LOCAL zeichen%, text$(255), n%
  text$= "Hallo Welt, jetzt komme ich!"
  gFILL 140,50,0
  n%=1
  WHILE n%<LEN(text$)
    gAT 20,30
    gPRINTB MID$(text$,n%,255),100
    PAUSE -4
    n%=n%+1
  ENDWH
  GET
ENDP
```

# 11 Fenster und Rahmen

---

## Fenster erzeugen und manipulieren

Bisher sind wir stillschweigend davon ausgegangen, dass der "Bildschirm" eine mehr oder minder feste Einrichtung ist, die sich gleichermaßen mit Textbefehlen (*AT*, *PRINT*, ...) als auch grafischen Befehlen (*gAT*, *gPRINT*, ...) zur Ausgabe verwenden lässt. Ab jetzt wollen wir aber sprachlich und technisch genauer unterscheiden.

Der Bildschirm ist lediglich eine physikalische Einheit, auf der einzelne Punkte, Pixel, angesteuert werden, die in ihrer Gesamtheit dem Auge die erwünschte Darstellung vermitteln. Der Bildschirm für sich ist also nur ausführendes Organ. Daten, die er darstellen soll, werden im Hintergrund durch ein ausgeklügeltes System aufbereitet und verwaltet. Dieses System "denkt" in "Fenstern".

Wenn ein OPL-Programm gestartet wird, sieht der Nutzer zunächst erst einmal das sogenannte Basisfenster, das sich über den gesamten Bildschirm erstreckt. Der Beweis: Legen Sie einen Rahmen um das Fenster und Sie bekommen einen optischen Eindruck:

```
PROC Rahmen:
  gBORDER 3
  GET
ENDP
```

Neben dem Basisfenster lassen sich insgesamt gleichzeitig weitere 63 Fenster auf den Bildschirm bringen oder, wie es im Fachchinesisch heißt, öffnen bzw. anlegen. Das Basisfenster allerdings ist das einzige, in dem mit Textbefehlen gearbeitet werden kann. Deshalb wird das Basisfenster synonym oft auch als Textfenster bezeichnet. Darüber hinaus ist es aber ein ganz normales Fenster, dessen Eigenschaften wie die aller anderen Fenster auch manipuliert werden können, z.B. in den Werten von Größe und Position:

```
PROC Fenster1:
  LOCAL x%,y%,b%,h%
  x%= 80 : y%= 40
  b%= gWIDTH - 2*x%
  h%= gHEIGHT - 2*y%
  PRINT "Alte Höhe/Breite:",gWIDTH, gHEIGHT
  gSETWIN x%,y%,b%,h%
  gBORDER 3
  PRINT "Neue Höhe/Breite:",gWIDTH, gHEIGHT
  GET
ENDP
```

*gWIDTH* und *gHEIGHT* enthalten die jeweils aktuelle Fensterbreite und -höhe und werden hier zur Berechnung herangezogen, während *gSETWIN* die eigentlichen Änderungen veranlasst.

Durch die Verkleinerung scheint plötzlich das "dahinter" liegende Fenster des Editors hervor. Üblicherweise lässt man daher das Textfenster in dem Zustand, in dem es sich beim Start befindet.

Neue Fenster legt man nun mit dem *gCREATE*-Befehl an:

```
id%= gCREATE(x%,y%,b%,h%,sichtbar%)
```

Die Koordinaten *x%* und *y%*, an denen das Fenster mit der linken oberen Ecke platziert wird, sind Pixelangaben, die sich immer auf das Bildschirmkoordinatensystem beziehen. Das Fenster wird nach rechts unten mit

der Breite *b%* und der Höhe *h%* aufgespannt. Obwohl negative Werte für *b%* und *h%* keine Fehlermeldung ergeben, so sind sie doch nicht zulässig. In diesem Falle ist das Fenster zwar vorhanden, aber nicht sichtbar.

Hat der Parameter *sichtbar%* den Wert 1, wird das Fenster sofort dargestellt, bei *sichtbar%=0* nicht. Im letzteren Fall erscheint das Fenster erst auf dem Bildschirm, wenn der Befehl *gVISIBLE ON* gegeben wird.

Ein frisch angelegtes Fenster kommt optisch "oben" zu liegen und ist gleichzeitig das aktuelle, d.h., (fast) alle weiteren Befehle beziehen sich bei der Ausführung jetzt auf dieses.

Damit man auch andere Fenster ansprechen, also zum aktuellen machen und damit arbeiten kann, vergibt das System Identitätsnummern. Um sich umständliches Zählen zu ersparen, erfasst man sie in einer Variablen. Im Beispiel zuvor ist das "*id%*". Hat man mehrere Fenster zu verwalten, sucht man sich zu jedem einen möglichst einprägsamen Variablen-Namen oder verwendet Variablenfelder.

Welches Fenster nun das aktuelle sein soll, teilt man dem System durch den Befehl

```
gUSE id%
```

mit. Das allerdings bringt ein Fenster immer noch nicht optisch in den Vordergrund, sondern erst das:

```
gORDER id%,pos%
```

Durch *id%* wird das Fenster benannt, das an die Position *pos%* gestellt werden soll. Dabei ist *pos%=1* die oberste, unverdeckte Position. Will man ein Fenster in die allerunterste Position bringen, also im Normalfall durch das Basisfenster verdecken, gibt man *pos%=64* an. Auch wenn weniger Fenster als die erlaubten 64 geöffnet sind, ergibt das keinen Fehler, sondern stellt das aktuelle Fenster eben nur in die letzte Position.

Die Position des aktuellen Fehlers lässt sich mit *gRANK* ermitteln. Weitere Informationen zum aktuellen Fenster bekommt man mit *gIDENTITY*, das die Identitätsnummer enthält. *gORIGINX* und *gORIGINY* geben Auskunft über die Position der linken oberen Ecke des Fensters, während man mit *gX* und *gY* die aktuelle Cursorposition erfährt. *gWIDTH* und *gHEIGHT* lassen Sie wissen, wie breit und hoch das Fenster ist. Alle Angaben sind im Pixel-Maß. Wer noch mehr wissen muss, benutzt *gINFO32* – siehe alphabetische Befehlsübersicht.

Braucht man ein Fenster nicht mehr, schließt man es besser, als dass man es lediglich mit *gORDER* in den Hintergrund schiebt oder mit *gVISIBLE OFF* unsichtbar macht – das entlastet den Prozessor:

```
gCLOSE id%
```

Der Parameter ist wieder die Fensternummer. Dabei muss es sich um die Nummer eines offenen Fensters handeln, ansonsten wird ein Fehler ausgegeben und das Programm stoppt! Werden eines oder mehrere Fenster geschlossen, ist im Anschluss daran die aktuelle Fensternummer die "1", also das Basisfenster. Das allerdings ist das einzige, das sich nicht schließen lässt.

```
PROC Fenster2:
  LOCAL b%,h%, win1%,win2%,win3%
  b%=100 : h%=100
  win1%=gIDENTITY
  PRINT "Basisfenster-ID:",win1%
  win2%=gCREATE (30,30,b%,h%,0) REM unsichtbar
  gBORDER 0
  gAT 5,16
  gPRINT "Fenster 2"
  gAT 5,36
  gPRINT "Pos:",gORIGINX,gORIGINY
  win3%=gCREATE (50,50,b%,h%,1) REM gleich sichtbar
  gBORDER 0
  gAT 5,16
  gPRINT "Fenster 3"
  GET
```

(Fortsetzung nächste Seite)

```

gUSE win2%
gVISIBLE ON      REM Fenster 2 sichtbar
GET
gORDER win2%,1   REM Fenster 2 in Vordergrund
GET
gORDER win2%,64  REM Fenster 2 hinter Basisfenster
PRINT "Position von Fenster",win2%,"=",gRANK
GET
gCLOSE win3%     REM Fenster 3 schliessen
GET
ENDP

```

Bei Programmende werden alle Fenster automatisch geschlossen.

## Fenster-Rahmen und Schatten

Ohne besondere Anweisung werden Fenster völlig rahmenlos dargestellt – manchmal ist das erwünscht, aber oft auch nicht. Daher wendet man diesen Befehl zum Zeichnen eines Rahmens an:

`gBORDER modus%,` oder auch `gBORDER modus%,breite%,hoehe%`

Der Parameter `modus%` erlaubt folgende Einstellungen:

modus%	Wirkung
0	einfacher Rahmen
1	Rahmen + Schatten 1 Pixel breit
2	Rahmen + "weißer" Schatten 1 Pixel breit
3	Rahmen + Schatten 2 Pixel breit
4	Rahmen + "weißer" Schatten 2 Pixel breit
\$100	Rahmen mit 1 Pixel Randabstand
\$200	rundere Ecken

Schatten wird durch schwarze Zusatzlinien am rechten und unteren Rand im Inneren des Fensters gebildet. Bei "weißen" Schatten wird der Schatten mit "weiß" gezeichnet. In der Praxis entsteht so auf dunklem Hintergrund ein sichtbarer Abstand.

Die Werte \$100 und \$200 können untereinander und mit den anderen Werten kombiniert werden, die Kombination der Werte 0 .. 4 macht jedoch keinen Sinn.

Beachten Sie, dass sämtliche auf diese Weise erzeugten Rahmen innerhalb der Grenzen des Fensters liegen und zu dessen Bestandteil gehören. Das bedeutet auch, dass durch den Löschbefehl `gCLS` nicht nur der Inhalt des Fensters verschwindet, sondern auch der Rahmen weg ist und im Bedarfsfall neu gezeichnet werden muss!

Gibt man bei `gBORDER` auch noch `breite%` und `hoehe%` an, werden Rahmen und Schatten entsprechend groß gezeichnet – die Wirkung ist dem Zeichnen eines Rechteckes mit `gBOX` ab der Position `gAT 0,0` vergleichbar, nur sind die Ecken einmal mehr und einmal weniger abgerundet.

Schießt man bei der Angabe der Werte über die Maße des Fensters hinaus, wird ein Teil des Rahmens einfach nur nicht sichtbar, eine Fehlermeldung ergibt das jedoch nicht.

Der `gBORDER`-Befehl wurde noch für die Vorversion des Betriebssystems ausgelegt, dadurch ist er nicht in der Lage, den halbseidenen Schatten zu bilden, der für die Menüs und Dialoge ab der Serie 5 typisch ist. Diese Funktionalität findet sich jetzt aber im erweiterten `gCREATE`-Befehl wieder:

```
id%= gCREATE(x%,y%,b%,h%,sichtbar%,modus%)
```



Neben der Schattenbildung ist `modus%` auch verantwortlich für die Anzahl der im Fenster verwendbaren Farben. Der Übersichtlichkeit halber werden hier Hexadezimalzahlen verwendet. Die "Einer-Stelle" steht für die Farbanzahl:

```
modus%= $0  REM  2-Graustufen-Modus
modus%= $1  REM  4-Graustufen-Modus
modus%= $2  REM  16-Graustufen-Modus
modus%= $3  REM  256-Graustufen-Modus
modus%= $4  REM  16-Farben-Modus
modus%= $5  REM  256-Farben-Modus
```

Farbgeräte stellen, auch wenn ein Graumodus gewählt wird, immer Farben dar. Man informiere sich in der alphabetischen Befehlsübersicht unter *gCOLORINFO* und *DEFAULTWIN* über die Problematik oder auch im Kapitel "Grafik", Abschnitt "Farben".

Die zweite Stelle lässt erkennen, ob ein Schatten dargestellt werden soll oder nicht:

```
modus%= $10      REM mit Schatten
modus%= $00      REM kein Schatten
```

Wenn ein Schatten gezeichnet werden soll, gibt die dritte Stelle an, um wieviele Pixel sich das Fenster über dem vorhergehenden Fenster befindet, die eigentliche Schattenbreite ist doppelt so groß.

```
modus%= $100     REM 2 Pixel Schatten
modus%= $400     REM 8 Pixel Schatten
```

Wird in der dritten Stelle eine Schattenbreite angegeben, aber die zweite Stelle ist Null, wird der Schattenwert ignoriert.

Der Endwert für `modus%` wird einfach durch Addition der Einzelwerte gebildet:

```
modus%= $2 + $ 10 + $300, oder einfach
modus%= $312
```

Im Beispiel wird ein 6 Pixel breiter Schatten dargestellt, das Fenster arbeitet im 16-Graustufen-Modus.

Mit Hilfe von `modus%` wird lediglich der Schatten, aber noch kein Rahmen für das Fenster angezeigt. Den muss man immer noch mit `gBORDER` erzeugen.

Einen besonderen Rahmeneffekt erreicht man mit

```
gXBORDER typ%,modus%
```

Mit dem Parameter `typ%` legen Sie die für die verschiedenen historisch existierenden Gerätegruppen charakteristischen Rahmentypen fest:

```
typ%= 0: Serie 3 Rahmen
typ%= 1: Rahmen der Serien 3a/3c
typ%= 2: Rahmen von Serie 5xx, Revo, netBook
```

Um das typische aktuelle Rahmendesign der EPOC32-Geräte zu gewährleisten, arbeitet man also immer mit `typ%=2`.

Für diesen Fall bestimmt dann `modus%` besondere Rahmeneffekte, die das Fenster wie einen herausragenden oder eingedrückten Druckknopf in verschiedenen Stadien erscheinen lassen. Das Fenster wird so zu einem frei form- und positionierbaren rechteckigen Button.

modus%	Effekt
0	kein Rahmen
\$01	einfacher Rahmen
\$42	flach eingesunken
\$44	tief eingesunken
\$54	tief eingesunken mit Outline
\$82	wenig herausragend
\$84	stark herausragend
\$94	stark herausragend mit Outline
\$22	ohne linken Rand
\$2A	ohne oberen Rand

Einer der folgenden Werte lässt sich jeweils hinzuaddieren, um bestimmte Eckeffekte zu erreichen – zumindest theoretisch. Auf einem Serie 5mxPro und einem Revo wollten sich die Effekte nicht einstellen:

```
modus%= modus% + $100 REM 1 Pixel Freiraum um den Rahmen
modus%= modus% + $200 REM rundere Ecken
modus%= modus% + $400 REM leicht rundere Ecken
```

Wie auch `gBORDER` lässt sich `gXBORDER` um Parameter für die Breite und Höhe des Rahmens ergänzen:

```
gXBORDER typ%,modus%,breite%,hoehe%
```

Sowohl `gBORDER` als auch `gXBORDER` sind nicht auf die Rahmung von Fenstern selber beschränkt, obwohl sie hier wohl ihre besten Dienste versehen. Sie lassen sich auch innerhalb des aktuellen Fensters beliebig positionieren. Ihre Wirkung ist vergleichbar dem `gBOX`-Befehl, nur dass bei entsprechendem Bedarf auch Schatten mitgezeichnet werden und die Ecken abgerundet erscheinen. Da der Rahmen nicht gefüllt wird, werden lediglich die Pixel des Rahmens selber in der üblichen Art gezeichnet, sogar die durch `gGMode` bestimmte Zeichenmethode ist wirksam.

```
PROC Fenster3:
  gAT 5,1   : gFILL 200,40,0
  AT 2,5    : PRINT "Normaler Text im Basisfenster"
  gAT 5,80  : gPRINT "Grafischer Text im Basisfenster"
  gGMode 2
  gAT 60,20 : gXBORDER 3,70,70
  GET
ENDP
```

# 12 Bitmaps

---

## Unsichtbare Fenster

Bitmaps – im deutschen Sprachgebrauch Bilder oder genauer auch Pixelbilder – sind den Fenstern verwandte Objekte. Viele bereits kennengelernte Grafik- und Fensterbefehle sind hier wieder verwendbar. Deshalb werden Fenster und Bitmaps in der Literatur auch in einem Atemzug als "drawables" (Zeichenflächen, Zeichenebenen) bezeichnet. Der wesentliche Unterschied: Bitmaps sind nicht sichtbar.

"Unsichtbare Fenster" scheinen keinen rechten Sinn zu ergeben, passen sich aber dennoch hervorragend in das OPL-Konzept ein.

Weil die Bitmaps unsichtbar sind, benötigen sie zu ihrer Erstellung keine Positionsangaben, können aber auf Angaben zu ihrer Ausdehnung nicht verzichten:

```
id%= gCREATEBIT(breite%,hoehe%)
id%= gCREATEBIT(breite%,hoehe%,modus%)
```

In einem dritten, optionalen Argument (modus%) wird der Grafikmodus festgelegt. Es gelten die gleichen Werte wie bei *gCREATE*.

Die Bitmaps reihen sich in die Anzahl der zu öffnenden Fenster ein. Die Gesamtanzahl einschließlich Basisfenster darf 64 nicht überschreiten.

Folgende Fenster-spezifische Befehle gelten auch für Bitmaps:

```
gBORDER, gXBORDER, gWIDTH, gHEIGHT, gIDENTITY,
gUSE, gCLOSE, gINFO, gINFO32.
```

Aus verständlichen Gründen liefern die folgenden Fenster-Befehle einen Fehler bei der Ausführung und dürfen deshalb nicht auf Bitmaps angewandt werden:

```
gSETWIN, gVISIBLE, CURSOR id%, gORIGINX, gORIGINY,
gRANK, gORDER.
```

Bitmaps sind nach dem Erstellen nichts als leere Flächen, die nur darauf warten, durch Anwendung der Zeichenbefehle zu echten Bildern oder Grafiken zu werden. Um diese Kunstwerke dann aufbewahren zu können, lassen sie sich als Bitmap-Datei abspeichern:

```
gSAVEBIT dateiname$
gSAVEBIT dateiname$,breite%,hoehe%
```

Geben Sie nur den Datei-Namen als Parameter an, wird das gesamte Bitmap gespeichert. Durch Setzen der Cursorposition (*gAT*) und Zuweisen von Werten für Breite und Höhe eines rechteckigen Bereiches wird die Speicherung nur eines Teils des Bitmaps veranlasst.

Der Befehl ist nicht auf Bitmaps beschränkt, sondern funktioniert mit allen Zeichenflächen!

Achten Sie zur eigenen Übersicht darauf, dass Sie dem Dateinamen die Endung "MBM" geben, der Computer macht das nicht automatisch. "MBM" steht für das EPOC-Bildformat "MultiBitMap". Solche Dateien können mehrere Bilder enthalten. Leider lassen sich derartige Dateien nicht von OPL aus erzeugen. Dazu muss man sich mit einem DOS-Programm quälen, das sich zumeist mit auf der zum Lieferumfang gehörenden PsiWin-CD befindet. Wenigstens kann man solche Multi-Bitmaps per OPL einlesen:

```
id%= gLOADBIT(dateiname$)
id%= gLOADBIT(dateiname$,aendern%)
id%= gLOADBIT(dateiname$,aendern%,index%)
```

Durch *gLOADBIT* wird automatisch ein in der Größe passendes, neues Bitmap angelegt, dessen Identitätsnummer in *id%* hinterlegt wird. Das zu öffnende Bild wird dorthinein geladen. Hat die Variable *aendern%* den Wert 1, darf man dieses Bild verändern und wieder abspeichern. Das klappt jedoch nicht, wenn *aendern%* Null ist. In diesem Falle darf die Bilddatei aber gleichzeitig auch von anderen Programmen im read-only-Modus benutzt, also angesehen werden.

Enthält die Bitmap-Datei mehrere Bilder, nennt *index%* die Nummer des Bildes, das geladen werden soll. Das allererste Bild hat den Indexwert Null. Wird kein Index angegeben, aber die Datei enthält mehrere Bilder, lädt das System automatisch das erste.

Bisher ist nach dem Laden des Bitmaps absolut nichts zu sehen. Der Inhalt muss erst in einen sichtbaren Bereich – ein echtes Fenster – übertragen werden. Dafür ist der *gCOPY-Befehl* vorgesehen:

```
gCOPY bm%,x%,y%,b%,h%,modus%
```

Das Ganze funktioniert ein wenig umständlich, wird aber mit etwas Übung zur Routine. Zuerst begibt man sich per *gUSE* in das Fenster, in dem die Kopie landen soll und benennt dort die Zielkoordinaten mit *gAT*. Dann folgt der Aufruf von *gCOPY*.

Die ersten fünf Parameter beziehen sich auf das geöffnete Bitmap: *x%* und *y%* sind Startposition, *b%* und *h%* die Breite und Höhe des rechteckigen Bereiches, der kopiert werden soll. Wollen Sie den Inhalt des gesamten Bitmaps kopieren, sind *x%* und *y%* Null – das ist die Cursorposition beim Öffnen/Laden oder Anlegen des Bitmaps. Breite und Höhe ermittelt man mit *gWIDTH* und *gHEIGHT*.

Schließlich muss man mit *modus%* noch festlegen, wie das Original ins Zielfenster geschrieben wird:

modus%	Pixel werden ..
0	gesetzt
1	gelöscht
2	invertiert
3	ersetzt (kopiert auch die "weißen" Stellen des Originals)

Mit *gCOPY* kopiert man auch aus Fenstern oder nicht gespeicherten Bitmaps. Allerdings empfiehlt der Softwarehersteller Symbian das Kopieren aus Fenstern heraus aus Geschwindigkeitsgründen nicht. Das Ziel – Fenster oder Bitmap – hat keinen Einfluss auf die Arbeitsgeschwindigkeit.

Die Empfehlung scheint mehr theoretischen Charakter zu besitzen – bei kleinen Objekten merkt man den Unterschied nicht.

Zusammenfassendes Beispiel auf der nächsten Seite:

```

PROC Bm1:
  LOCAL bm%,b%,h%,win%,win2%,modus%
  DEFAULTTWIN 1    REM 4 Farben
  win%= gIDENTITY
  PRINT "Testgrafik"
  gBOX 90,90
  gAT 10,20 : gFILL 50,50,0
  gGREY 1    REM grau
  gAT 30,30 : gFILL 50,50,0
  gAT 0,0 : gSAVEBIT "testgrafik.mbm",90,90
  BUSY "Taste für weiter!",1 : GET : BUSY OFF
  bm%=gLOADBIT ("testgrafik.mbm")
  b%= gWIDTH : h%=gHEIGHT
  gUSE win% : CLS
  modus%=0      REM Pixel setzen
  gAT 50,50
  gCOPY bm%,0,0,b%,h%,modus%
  PRINT "Bitmap ins Basisfenster kopiert"
  BUSY "Taste für weiter!",1 : GET : BUSY OFF
  win2%= gCREATE (160,50,90,90,1,$411)
  gBORDER 3
  REM gAT 0,0 : gFILL 90,90,0
  gAT 10,10
  modus%=0      REM 0 .. 3 erlaubt
  gCOPY bm%,0,0,b%/2,h%/2,modus%
  PRINT "Bitmap teilweise in Fenster",win2%,"kopiert"
  BUSY "Taste für Ende!",1 : GET : BUSY OFF
ENDP

```

Ist ein Bitmap einmal geladen oder geöffnet, lässt es sich problemlos mehrfach als Datenbasis benutzen. Wird es allerdings nicht mehr benötigt, sollte man es schließen, um Prozessor und Speicher zu entlasten.

# 13 Datenbanken

---

## Sparbüchse für Daten

Datenbanken (kurz: DBs) sind aus unserem Leben nicht mehr wegzudenken. Ihnen fallen sicher selber etliche Beispiele ein: diese ungeliebte Kartei mit den Pünktchen in Flensburg, die Bankkonten, Ihr Adressverzeichnis auf einem simplen elektronischen Merkzettel, die Datenbanken, die hinter Internetsuchmaschinen stehen, usw. So ist es kein Wunder, dass auch OPL einen Befehlssatz besitzt, mit dem man Datenbanken anlegen und verwalten kann. Beim näheren Hinsehen muss man sogar sagen, dass die Betriebssystem-Entwickler ganze Arbeit geleistet haben. Sie spendierten dem Psion ein ausgefeiltes Datenbank-Management-System (DBMS), auf das man auch mit OPL zurückgreifen kann. Damit lassen sich nicht nur einfache Datenbanken, sondern auch komplexe relationale DBs aufstellen und per SQL (Standard Query Language), der weit verbreiteten und bekannten Abfragesprache verwalten.

Obwohl wir nicht zu tief in die Materie eintauchen wollen, sollen Sie doch das Datenbankprinzip im folgenden kennen und verstehen lernen.

Daten können immer erst dann verarbeitet werden, wenn sie irgendwo elektronisch gesammelt wurden. Das ist zunächst trivial. Solange man aber selber noch nie vor einem Haufen unsortierter Fakten gestanden hat, ist einem die gelegentlich recht komplexe Problematik fern. In einem solchem Falle werden Sie jedoch recht schnell lernen, dass man schleunigst ein Ordnungsprinzip finden muss, sonst nutzen einem die schönsten Datensammlungen nichts.

Sie werden sich daher zuerst bemühen, den Datenhaufen in eine Tabellenform zu bringen. Damit haben Sie aber schon den Grundstock zur Datenbank gelegt! An die etwas anderen Bezeichnungen – Datensatz statt Zeile, Feldbezeichner anstelle der Spaltenüberschrift und Feld statt Zelle – gewöhnen Sie sich schnell.

Müssen Sie nur eine einzelne Tabelle zur Unterbringung Ihrer Daten bemühen, reden wir von einer einfachen Datenbank. Vereint die Datenbank mehrere Tabellen, die inhaltlich auch noch untereinander verknüpft sind, liegt eine relationale Datenbank vor. Die wesentlichen Vorteile: relationale DBs sparen Speicherplatz und pflegen sich leichter.

Ein Beispiel soll das Prinzip relationaler Datenbanken verdeutlichen.

Sie unterziehen sich der Mühe, Ihre Bücher elektronisch zu katalogisieren. Daher erfassen Sie deren Titel, Autor und Verlag. Vom Verlag notieren Sie aber nicht nur den Namen, sondern unter jeweils eigenen Feldbezeichnern auch die Postadresse, die Telefonnummer und eine Kontaktperson. Da Sie mit einiger Sicherheit mehrere Bücher von einem Verlag besitzen, tragen Sie immer wieder mühsam dieselben Verlagsdaten neu ein. Im Laufe der Zeit ändern sich gelegentlich Telefonnummern oder Anschriften der Verlage, so dass Sie mit neuen Büchern auch die frischen Daten eintragen. Die alten Einträge bleiben dabei unkorrigiert – das Durch-einander bei einer Abfrage ist somit bereits vorprogrammiert.

In einer relationalen Datenbank werden nun die Verlage mit den Zusatzangaben in einer eigenen Tabelle geführt. Die Büchertabelle enthält anstelle der vollständigen Verlagsdaten nur noch einen Verweis auf die Verlagstabelle. Ändern sich die Verlagsdaten, muss der zugehörige Datensatz nur noch an einer Stelle gepflegt werden und ist so immer auf dem neuesten Stand.

## Datenbanken und Tabellen anlegen

Die wesentlichen Dinge, die man mit einer Datenbank tun können muss, sind: Daten eingeben, verändern und abfragen. Bevor eine Datenbank überhaupt benutzbar ist, muss man sie erstellen. OPL hält dafür den *CREATE*-Befehl bereit.

```
CREATE "meineDB FIELDS Feld1,Feld2,Feld3 ... TO TabName",logName,f1,f2,f3,...
```

bzw.

```
DBname$= "meineDB"
CREATE DBname$ + " FIELDS Feld1,Feld2,Feld3 ... TO TabName",logName ,f1,f2,f3,...
      REM das Leerzeichen vor FIELDS ist wichtig!
```

Durch *CREATE* wird eine Datenbank mit dem Namen "meineDB" angelegt und geöffnet. In allgemeiner Form schreibt man das wie in der zweiten Ausführung des Befehls, wo ein String die direkte Eingabe des Dateinamens ersetzt. Sinnvollerweise enthält der Dateiname die volle Spezifikation einschließlich Pfad, also das Laufwerk, den Ordner und schließlich den Namen, z.B.: "C:\Dokumente\mdb1".

In der geöffneten Datenbank wird anschließend eine Tabelle erzeugt, die ihren Namen von "TabName" bezieht. Die Tabelle hat Felder mit den Bezeichnungen, die in der kommagetrennten Liste zwischen "FIELDS" und "TO" stehen – hier also: Feld1, Feld2, Feld3 ...

Nach der Spezifikation der Datenbank folgt ein logischer Name (logName), mit dem die Tabelle angesprochen wird. Verwendet werden dürfen die Buchstaben von "A" bis "Z". Daraus ist abzuleiten, dass zur gleichen Zeit bis zu 26 Datenbanken geöffnet sein dürfen.

Die im *CREATE*-Befehl benutzten Variablen(f1, f2, f3, ...), die nach dem logischen Bezeichner folgen, müssen – im Gegensatz zu anderen Variablen – vorab nicht deklariert werden. Sie sind passend zur Reihenfolge der Feldbezeichner und mit dem geeigneten Datentyp zu notieren. Es gehören also zusammen: f1 mit Feld1, f2 mit Feld2, usw.

Ein praktisches Beispiel zeigt es einfacher. Wir legen im Wurzelverzeichnis "C:\\" die Datenbank "mdb1" mit der Tabelle "Liste" an.

```
PROC Db1:
  LOCAL DBname$(255)
  DBname$= "c:\mdb1"
  CREATE meineDB$ + " FIELDS Index,Name(40),Vorname(40) TO Liste",A,i%,n$,v$
  CLOSE
ENDP
```

Die Tabelle "Liste" enthält Felder mit den Bezeichnern "Index", "Name" und "Vorname". Die Variablen i%, n\$ und v\$ lassen erkennen, dass "Index" eine Ganzzahl ist und dass es sich bei den beiden anderen um Strings handelt. Die geklammerten Zahlen hinter "Name" und "Vorname" begrenzen, ähnlich wie bei einer Deklaration, die Stringlänge auf 40 Zeichen. Gibt man keine Begrenzung vor, ist der Standardwert 255. Der logische Name lautet "A".

Die Datenbank bleibt zunächst einmal leer und wird sofort wieder mit *CLOSE* geschlossen. Um eine weitere Tabelle, diesmal mit dem Namen "Adresse", in die Datenbank einzubringen, wiederholt man den Vorgang. Das Beispiel finden Sie auf der nächsten Seite.

```

PROC Db2:
  LOCAL DBname$ (255)
  DBname$= "c:\mdb1"
  CREATE DBname$ + " FIELDS Index,Ort,Strasse,Telefon TO Adresse",A,i%,o$,s$,t$
  CLOSE
ENDP

```

Wollen Sie noch mehr Tabellen in dieselbe Datenbank einfügen, schließen Sie sie jeweils vorher wieder.

## Daten speichern

Tragen wir nun einen Datensatz in eine Tabelle ein. Dazu muss die DB mit dem Befehl *OPEN* geöffnet werden. Als Beispiel sieht das so aus:

```

PROC Db3:
  LOCAL DBname$ (255)
  DBname$= "c:\mdb1"
  OPEN DBname$ + " SELECT Index,Name,Vorname FROM Liste",D,i%,n$,v$

  INSERT
    D.i%= 1
    D.n$= "Muster"
    D.v$= "Margarete"
  PUT
  CLOSE
ENDP

```

Dem *OPEN*-Befehl gibt man zunächst wieder die Datenbankspezifikationen mit. Das sind neben Datei-Pfad und -Name die Feldbezeichnungen (hinter *SELECT*) und die Tabelle, zu der die Felder gehören (hinter *FROM*). Dann folgen wie bei *CREATE* der logische Name und die Variablen. Tabellename und Feldbezeichnungen müssen mit denen aus dem *CREATE*-Befehl übereinstimmen, die Variablennamen nicht, wohl aber die Variablentypen.

Die geöffnete Tabelle lässt sich nun verändern. In diesem Falle schreiben wir einen einzigen Datensatz in die Datenbank. Eingeleitet wird der Vorgang durch den Befehl *INSERT* ("einsetzen"). Die eigentlichen Daten werden zwischen den Befehlen *INSERT* und *PUT* den Feldvariablen zugeordnet. *PUT* ("ablegen") sorgt schließlich für das Schreiben in die Datenbank.

In der Praxis kommt es vor, dass Sie die Dateneingabe abbrechen wollen, die Feldvariablen aber bereits zugeordnet haben. In diesem Falle benutzen Sie statt *PUT* den Befehl *CANCEL*. Der letzte eingegebene Datensatz wird dann nicht in die Datenbank geschrieben. In Verbindung mit einem Dialog könnte das so aussehen:

```

PROC Db4:
  ...
  INSERT
  DINIT
  dEDIT D.n$,"Eingabe: "
  IF DIALOG
    PUT
  ELSE
    CANCEL
  ENDIF
  ...
ENDP

```

Wird die Eingabe im Dialog mit der Enter-Taste bestätigt, gelangt die Eingabe über "D.n\$" in die Datenbank, bei Abbruch mit der Esc-Taste wird die Eingabe nicht übernommen.



## Beschleunigung durch Transaktionen

Wie frische Daten prinzipiell in die Datenbanktabelle kommen, haben Sie nun gesehen. Sie können den Teil *INSERT ... INPUT* immer wieder verwenden, um weitere Daten einzubringen. Der Vorgang ist jedoch sehr langsam. Das fällt bei einer Dateneingabe über einen Dialog nicht weiter auf, kann aber beim Verarbeiten von Daten (Auslesen einer Datenbank und Schreiben in eine zweite) lästig und langweilig werden.

Die Lösung: Anwendung der Befehle *BEGINTRANS* und *COMMITTRANS*. Mit ihnen werden "Transaktionen" eingeleitet und durchgeführt. Der Geschwindigkeitszuwachs ist gewaltig. Vollziehen Sie unser Beispiel nach, indem Sie es einmal mit und einmal ohne *BEGINTRANS/COMMITTRANS* laufen lassen. Es wird eine neue DB angelegt und mit Daten gefüllt, die automatisch generiert werden, um die Eingabe per Hand zu ersparen:

```
PROC Db5:
  LOCAL DBname$ (255), n%
  DBname$= "c:\mdb1"
  IF EXIST DBname$
    DELETE DBname$ REM löscht vorhandene DB
  ENDIF
  CREATE DBname$ + " FIELDS Index,Name TO Liste",A,i%,n$

  REM BEGINTRANS

  n%= 1
  WHILE n%< 1000
    INSERT
      A.i%= n%
      A.v$= "ein Name"
    PUT
    n%= n% + 1
  ENDWH
  REM COMMITTRANS
  CLOSE
  PRINT "Ende" : GET
ENDP
```

Ob gerade eine Transaktion mit der aktuellen Tabelle läuft, prüfen Sie mit *INTRANS*:

```
i&= INTRANS
```

Ist *i&= -1*, findet eine Transaktion statt, bei *i&= 0* nicht.

Sollen alle seit dem Befehl *BEGINTRANS* geänderten Daten nicht in die DB übernommen werden, wendet man den Befehl *ROLLBACK* anstelle von *COMMITTRANS* an. Neue Transaktionen müssen danach wieder mit *BEGINTRANS* angeschoben werden.

Nach all den Aktionen und Transaktionen hat sich Ihre Datenbank gefüllt. Jetzt können Sie damit arbeiten.

Voraussetzung ist immer das Öffnen der Datenbank. Um genau zu sein: wer sagt, dass er eine Datenbank öffnet, meint, dass er die Inhalte einer oder mehrerer Tabellen darin dem verarbeitenden Programm zugänglich macht. Im Zusammenhang mit "geöffneten" Datenbank-Tabellen redet man auch von "Ansichten". Sie sind gewissermaßen teilweise oder vollständige virtuelle Kopien der Tabellen und mit diesen arbeitet man. Das im Hintergrund die Organisation übernehmende Datenbank-Management-System (DBMS) lässt es zu, dass von einer Tabelle platzsparend mehrere Ansichten geöffnet sein können. Jede Ansicht kann eine andere Zusammenstellung der Felder aufweisen. Für den Profi hat das Vorteile, der Einsteiger löst seine Probleme mit Sicherheit auch mit einfacheren Konstellationen.

## Suchen ... und finden

Ist die Datenbank einmal geöffnet, muss man Daten, die man darstellen oder manipulieren will, zunächst einmal finden. Dazu bedient man sich des Suchbefehls:

```
n%= FIND(string$)
```

Der Befehl sucht in der gesamten Ansicht nach dem Begriff, der in `string$` enthalten ist. Wird etwas gefunden, steht die Datensatznummer anschließend in `n%`. Der gefundene Datensatz wird gleichzeitig zum aktuellen, d.h., man kann jetzt mit dessen Daten arbeiten. Beispielsweise kann man nun den Inhalt oder Teile des Datensatzes darstellen:

```
PRINT A.name$
```

Fällt einem der zu suchende Begriff nicht mehr haargenau ein (Meyer, Maier, Meier ...), dürfen Wildcards (Ersatzzeichen) benutzt werden. Ein `""` steht dabei für eine beliebige Zeichenkette, ein `"?"` ersetzt dabei ein einzelnes Zeichen. Damit findet man mit Sicherheit das Gesuchte:

```
n%= FIND("M??er") .
```

Oder

```
n%= FIND("*gr??")
```

findet sowohl grau als auch grün, aber auch alle Kombinationen damit, z.B. hellgrau.

Gerade in Verbindung mit Wildcards fallen bei der Suche mehrere Ergebnisse an. *FIND* setzt die Suche aber nicht automatisch fort. Sie müssen zunächst die Datensatznummer mit dem *NEXT*-Befehl um eins erhöhen und dann die Suche erneut starten.

Da es passieren kann, dass der aktuelle Datensatz bereits der letzte war, müssen Sie das mit *EOF* prüfen, um die Suche abzuschließen:

```
ende%=EOF
```

Wenn `ende% = -1` (also WAHR) ist, ist das Ende der Datenbank erreicht, ansonsten enthält `ende%` eine Null. Eine einfache Abfrage könnte so aussehen:

```
WHILE NOT EOF
  FIND(string$)
  PRINT A.name$
  NEXT
ENDWH
```

*FIND* beginnt seine Suche stets ab dem aktuellen Datensatz und sucht aufwärts. Es können mit diesem Befehl nur Stringfelder durchsucht werden. Groß- und Kleinschreibung werden dabei nicht unterschieden.

*FINDFIELD*, der zweite Suchbefehl, kann ebenfalls nur in Stringfeldern suchen, ist aber aufgrund einstellbarer Parameter flexibler:

```
n%= FINDFIELD(string$,startfeld%,anzahl%,flag%)
```

Der Suchbegriff steht wieder in `string$`. In `startfeld%` steht die Nummer des ersten Stringfeldes, ab dem gesucht wird. Der Parameter `anzahl%` enthält die Anzahl der Stringfelder, die in jedem Datensatz durchsucht werden sollen, einschließlich dem ersten Feld. Durchsucht man nur ein einziges Feld, ist `anzahl%` also 1, der Maximalwert ist die Summe aller Stringfelder, die beim Anlegen der DB mit *CREATE* festgelegt wurden.

Mit `flag%` bestimmt man die Suchrichtung:

```

flag%= 0    REM rückwärts ab aktuellem Datensatz
flag%= 1    REM vorwärts ab aktuellem Datensatz
flag%= 2    REM rückwärts vom Ende der Ansicht
flag%= 3    REM vorwärts vom Anfang der Ansicht

```

Normalerweise wird auch bei *FINDFIELD* die Groß- und Kleinschreibung nicht berücksichtigt, addiert man zu *flag%* jedoch 16, wird die Unterscheidung aktiviert.

War die Suche erfolgreich, wird die Datensatznummer in *n%* zurückgeliefert und der Datensatz wird zum aktuellen. Die Weitersuche ist wie bei *FIND* zu organisieren.

Durch Suchaktionen verliert man leicht die Übersicht, welcher Datensatz in der Ansicht gerade der aktuelle ist. Deshalb gibt es Befehle, mit denen man sich in den Datensätzen bewegen kann. *NEXT* haben Sie schon kennengelernt. Der vorhergehende Datensatz wird mit *BACK* zum aktuellen. *LAST* benutzt man, wenn der letzte Datensatz zum aktuellen werden soll, mit *FIRST* wird es der erste. Direkt nach dem Öffnen einer Ansicht ist immer der erste Datensatz der aktuelle.

Hat man auf den ersten Datensatz positioniert, ergibt ein *BACK* keine Änderungen. Genauso verhält sich das mit *LAST* und *NEXT* – hier wird jedoch *EOF* auf WAHR, also -1 gesetzt.

In bestimmten Situationen will man sich die Position eines Datensatzes merken. OPL stellt dafür den Befehl *BOOKMARK* zur Verfügung. Mit

```
n%= BOOKMARK
```

merkt man sich die aktuelle Datensatzposition, allerdings steht in *n%* nicht die Position, sondern die Bookmark-Nummer. Der Datensatz lässt sich nun mit

```
GOTOMARK n%
```

wieder anspringen und zum aktuellen machen. Benötigt man die Markierung nicht mehr, entfernt man sie mit:

```
KILLMARK n%
```

## Änderungen

Nachdem Sie sich vertraut gemacht haben, wie man sich durch die Ansicht bewegt und wie man Daten findet, können Sie auch gezielt Änderungen vornehmen. Die einfachste Änderung wäre die Ergänzung des bestehenden Datensatzes per *INSERT ... PUT*. Neue Datensätze werden am Ende der geöffneten Ansicht eingefügt. Die mit *COUNT (n%= COUNT)* feststellbare Gesamtzahl der Datensätze in der Ansicht erhöht sich um eins.

Wollen Sie bestehende Datensätze ändern, funktioniert das nach dem gleichen Prinzip, Sie verwenden lediglich eine *MODIFY ... PUT* Kombination. Die Datensatzposition bleibt erhalten.

Muss ein Datensatz ganz verschwinden, reicht der Befehl *ERASE*. Der Datensatz wird gelöscht und der nächst folgende zum aktuellen. Steht der zu löschende Datensatz in der Position *LAST*, wird der Datensatz nur geleert, *COUNT* aber um eins vermindert und *EOF* auf WAHR gesetzt. Der leere Datensatz bleibt aber der aktuelle.

Wie Sie sehen konnten, ist das Arbeiten immer nur mit Daten möglich, die sich in aktuellen Datensätzen befinden. Ähnlich verhält es sich mit Ansichten. Sie können zur gleichen Zeit mehrere Ansichten im Speicher halten, aber immer nur mit einer arbeiten. Um die spezielle Ansicht zur aktuellen zu machen, verwendet man den Befehl *USE*:

```
USE D
```

Der Parameter ist der logische Bezeichner für eine der geöffneten Ansichten.

Nicht immer ist es wünschenswert, dass Ihr Datenbestand von einem Programm verändert wird. Benutzen Sie dann anstelle von *OPEN* den Befehl *OPENR* mit den gleichen Parametern wie oben beschrieben. Die Daten sind dann "read only" und könnten daher gleichzeitig durch ein anderes Programm benutzt werden.

Wenn Sie viele Daten in Ihre Datenbank eingegeben haben, sollten Sie diese hin und wieder komprimieren. Das Ergebnis ist verblüffend! Führen Sie den Befehl an der geschlossenen Datenbank aus:

```
COMPACT Dbname$
```

Wie beim Anlegen oder Öffnen sollte DBname\$ dabei den gesamten Pfad zur Datei enthalten, z.B. "C:\Dokumente\mdb1"

## Dateiverwaltung

Oftmals muss man für neue Datenbanken oder auch sonst neue Pfade anlegen, manchmal aber auch alte (Datenbank-) Dateien vom Programm aus löschen können. Dafür steht neben den oben bereits angeführten eine Reihe weiterer Befehle bereit, die an die ersten Erfahrungen mit DOS erinnern.

```
COPY quelle$,ziel$    REM Datei-Kopierfunktion
RENAME alt$,neu$      REM Datei umbenennen
DELETE dateiname$     REM vollst. Dateispezifikation
EXIST (dateiname$)    REM existiert die Datei?
DIR$(dateispezif$)    REM listet Dateien
MKDIR verz$           REM neues Verzeichnis anlegen
RMDIR verz$           REM Verzeichnis löschen
SETPATH$ verz         REM Standardverzeichnis f. Dateien
```

Die Befehle sind ausführlich im alphabetischen Befehlsverzeichnis beschrieben und bedürfen hier keiner weiteren Erläuterungen.

# 14 Fehler

---

## Fehlerarten

Wer mit dem Programmieren erst anfängt, wird sehr schnell feststellen, was professionelle Programmierer schon längst wissen: fehlerfreies Programmieren ist nicht möglich, insbesondere, wenn es um größere Projekte geht.

Es gibt zwei Arten von Fehlern: solche, die beim Schreiben die Syntax-Regeln verletzen und bereits während der Übersetzung bemerkt werden, und solche, die sich erst zur Laufzeit des Programmes melden.

## Syntax-Fehler

Sie sind die eher unproblematischen, denn das Programm kommt gar nicht erst zum Laufen, wenn sie nicht behoben werden. Sie machen sich bei der Übersetzung durch eine Fehlermeldung wie z.B. "Unbekannte Variable" oder "String zu lang" bemerkbar und die Übersetzung bricht ab. Die Fehlerausschriften sind kurz und nicht immer sehr aussagekräftig, meist entdeckt man aber die "faule Stelle" recht schnell. Besonders beliebt sind anfangs einfache Schreibfehler oder das Vergessen von Interpunktionszeichen sowie Strukturfehler – also das nicht korrekte Beenden von Prozeduren oder Schleifen.

## Fehler zur Programmlaufzeit

Diese Fehlerart ist besonders tückisch. Selbst bei intensivem Test findet man meist nicht alle. So ist immer noch der Nutzer des Programmes der beste "Tester" und nach Murphys Gesetz ("Wenn etwas schief gehen kann, geht es auch schief – irgendwann.") machen sich die Fehler natürlich erst bei ihm und nicht vorher bemerkbar. Ein Beispiel dafür ist die Gefahr, dass das Programm aufgrund einer Nutzereingabe versucht, eine Zahl durch Null zu teilen. Das wird dann auch prompt mit einer Fehlermeldung und Programmabbruch quittiert.

Für alle diese Fälle bietet OPL glücklicherweise Methoden zur Erkennung und Behandlung.

## TRAP – Fehler ignorieren

Bestimmte Befehle, die bereits ein hohes Fehlerpotential in sich bergen, lassen sich mit dem Vorsatz *TRAP* versehen (s. alphabetische Liste). Tritt dann ein Fehler auf, reagiert das Programm nicht mit Abbruch, sondern der Befehl wird einfach ignoriert, aber eben auch nicht ausgeführt. Eine Fehlermeldung wird ebenfalls unterdrückt.

Stellen Sie sich vor, Sie wollen ein Fenster, das bereits geschlossen wurde, noch einmal schließen. Fehlermeldung und Programmabbruch sind unvermeidlich – nicht jedoch, wenn Sie *TRAP* verwenden:

```
TRAP gCLOSE win%
```

Achtung, *TRAP* wirkt nur für diese eine Zeile!

Obwohl das Programm weiterläuft, registriert das System den Fehler und merkt sich die Fehlernummer.

## Fehleranalyse

Die Fehlernummer erfahren Sie über die Funktion *ERR*, die Fehlerbeschreibung selber mit *ERR\$* und die genaue Lokalisierung des Fehlers (Prozedur etc.) mit *ERRX\$*. Versuchen Sie einmal, das Basisfenster zu schließen und registrieren Sie den Fehler:

```
REM OPL-Programm, Name: "Fehler"
PROC Fehler1:
  LOCAL fehler%
  TRAP gCLOSE 1
  fehler%=ERR
  PRINT fehler%, ERR$(fehler%),ERRX$
  GET
ENDP
```

Bildschirmanzeige: "-2 Ungültige Argumente Fehler in FEHLER\FEHLER1" ("FEHLER" ist der Name des OPL-Programmes, "FEHLER1" der Prozedurname).

Das System erlaubt das Schließen des Basisfensters nicht, demzufolge ist "1" ein ungültiges Argument.

Versuchen Sie es einmal mit dem Argument "2". Dieses Argument ist zwar gültig, aber im Beispiel gibt es kein geöffnetes Fenster mit der Identitätsnummer 2, die Folge ist die Ausschrift: "-118 Zeichenfläche nicht geöffnet, Fehler in FEHLER\FEHLER1"

So wird durch *TRAP* der Programmabsturz verhindert und der Benutzer erhält einen Hinweis.

Während *TRAP* nur eine Zeile schützt, kann das Konstrukt *ONERR .. ONERR OFF* ganze Prozeduren im Auge behalten.

## Fehler behandeln

Ein Beispiel:

```
PROC Fehler2:
  LOCAL win%
  PRINT "Fensternummer?", : INPUT win% : CLS
  schliesse:(win%)
  BUSY "Beliebige Taste drücken"
  GET
  BUSY OFF
ENDP

PROC schliesse:(w%)
  ONERR fehlerbehandlung::
  gCLOSE w%
  RETURN
fehlerbehandlung::
  ONERR OFF
  IF ERR=-2 AND (w%<1 OR w%>64)
    PRINT "Fensterwerte <2 und >64 nicht erlaubt"
    PRINT "Basisfenster nicht schließbar!"
  ELSE
    PRINT "Fehler!"
    PRINT "Fehlernummer:",ERR
    PRINT "Fehlertext:",ERR$(ERR)
  ENDIF
ENDP
```

*ONERR* schützt hier die gesamte Prozedur *schliesse*:. Tritt ein Fehler auf, und der ist hier vorprogrammiert, springt die Prozedur zu dem Label, das hinter *ONERR* genannt ist und arbeitet das dort stehende Programmstückchen ab. Um Strukturfehler oder andere unbeabsichtigte Auswirkungen zu vermeiden, wird *ONERR* mit dem Gegenbefehl *ONERR OFF* deaktiviert und danach die Fehlerbehandlung ausgeführt.

Streicht man in diesem Beispiel die Zeilen, die für die Fehlerbehandlung zuständig sind, hat man praktisch ein TRAP, das nicht mehr nur auf eine Zeile beschränkt ist – aber das grenzt bereits hart an Missbrauch ...

## RAISE – mein Fehler!

Mit *RAISE* sind Sie in der Lage, Fehler selber hervorzurufen. Das scheint erst einmal absurd zu sein, hat aber seine Berechtigung. Einerseits können Sie Programmteile, die fehlerträchtig sind, gleich so schreiben, dass der Fehler nicht zum Tragen kommt, Sie aber trotzdem eine Fehlermeldung bekommen. Der Fehler trifft Sie also nicht mehr "unerwartet", sondern Sie sind darauf eingestellt und reagieren darauf in geeigneter Weise. Als Beispiel ändern wir die Prozedur "schliesse:(w%)" aus dem letzten Beispiel:

```
PROC schliesse:(w%)
  ONERR fehlerbehandlung::
    IF w%<2 OR w% >64
      RAISE -2
    ELSE
      gCLOSE w%
    ENDIF
  RETURN

fehlerbehandlung::
  ONERR OFF
  PRINT "Fenster",w%,"schließen - Fehler!"
  PRINT "Fehlernummer:",ERR
  PRINT "Fehlertext:",ERR$(ERR)
ENDP
```

Andererseits können Sie eigene Fehlermeldungen erzeugen. Systemfehlernummern sind immer negative Integerzahlen. Verwenden Sie positive Zahlen, lautet die Fehlerbezeichnung ERR\$ immer "unbekannter Fehler", Sie müssen daher selbst für die korrekte Meldung sorgen. Es folgt hier das Beispiel *schliesse2:(w%)*, das eine Abwandlung von *schliesse:(w%)* darstellt. Mit ihm lassen sich eigene Fehlermeldungen verarbeiten.

```
PROC schliesse2:(w%)
  ONERR fehlerbehandlung::
    IF w%=1
      RAISE 1
    ELSEIF w%<1 OR w% >64
      RAISE 2
    ELSE
      gCLOSE w%
    ENDIF
  RETURN

fehlerbehandlung::
  ONERR OFF
  IF ERR=1
    PRINT "Basisfenster nicht schließbar!"
  ELSEIF ERR=2
    PRINT "Werte <2 oder >64 nicht erlaubt."
  ELSE
    PRINT "Anderer Fehler, Nr.:",ERR,ERR$(ERR)
  ENDIF
ENDP
```

Sie können erkennen, dass sich so sehr viel informativere Meldungen erzeugen lassen. Und das ist besonders interessant während der Programmentwicklung. Nachdem man sicher ist, dass alles wie gewollt funktioniert, nimmt man die eigene Fehlererfassung wieder heraus oder lässt sie nur dort drin, wo es wirklich im fertigen Programm einen Sinn macht.

Haben Sie das Handbuch gerade nicht zur Hand, verschaffen Sie sich einen Überblick über die Fehlerliste mit folgendem Progrämmchen, das auch noch einmal *RAISE* nutzt (es ginge natürlich auch ohne):

```

PROC Fehlerliste:
  LOCAL n%
  n%= 0
  WHILE n%>-133
    n%=n%-1
    TRAP RAISE n%
    PRINT ERR,ERR$(ERR)
    PAUSE -1
  ENDWH
  GET
ENDP

```

## Korrektter Einsatz der Fehlerbehandlung

*ONERR* lässt sich durchaus mehrfach in einem Programm benutzen. Dabei ist aber immer nur das zuletzt aufgerufene *ONERR* das aktive. Bei ungeschickter Platzierung können Endlosschleifen aufgebaut werden, beachten Sie dieses "Unfallrisiko". Am besten fahren Sie, wenn Sie einzelne Prozeduren absichern und bei deren Verlassen sicherstellen, dass *ONERR* nicht mehr aktiv ist. Die Gefahr neben der Endlosschleife ist, dass ausgegebene Fehlermeldungen Nutzer und Programmierer irre führen.



# 15 Applikationen

---

## Der Weg nach oben

Bis hierher haben Sie alle Programmbeispiele abgeschrieben, übersetzt und ausführen lassen. Vielleicht haben Sie aber auch einfach ein übersetztes Programm (\*.OPO) vom Systembildschirm aus gestartet. Solche Programme können Sie durchaus auch an andere Interessierte weitergeben. Wenn die eigenen Programme aber immer "schönere" Gestalt annehmen, wächst der Wunsch, sie auch äußerlich den professionellen Programmen gleichen zu lassen. Das bedeutet, sie sollen in der Extras-Leiste als Programm erkennbar sein, um sie leichter starten zu können und nicht erst irgendwo im Speicher suchen zu müssen. Ihren Wunsch können Sie sich erfüllen, wenn Sie aus einfachen OPL-Programmen sogenannte Applikationen erstellen.

Der Weg dorthin ist im Grunde genommen lediglich an ein paar Formalismen gebunden. Anfangs wird es sich bei Ihren Werken um relativ einfache Arbeiten handeln. Daher müssen nicht alle Umstände berücksichtigt werden, an die ein Profi denken muss. Wenn Sie erst soweit sind, dass Ihre Programme mit Dokumenten und anderen Datendateien arbeiten können, brauchen Sie diese Ratschläge nicht mehr.

## Formalitäten

Der Formalismus sieht so aus, dass vor Ihr bestehendes OPL-Programm ein spezieller Anweisungsblock für den OPL-Übersetzer geklemmt wird. Der Übersetzer erzeugt dann daraus die besondere Form eines ausführbaren Programms: die Applikation (auch "Anwendung" oder einfach "Programm" genannt). Allgemein sieht der OPL-Quelltext so aus:

```
APP applikationsname, UID&
  REM hier steht eine spezielle Anweisungsliste
ENDA

PROC procname:
  REM Hier steht Ihr Programm
ENDP
REM es koennen weitere Prozeduren folgen
```

Die Liste zwischen APP und ENDA kann bzw. muss einige der wenigen nutzbaren Anweisungen (CAPTION, ICON und FLAGS) enthalten.

Wenn eine OPL-Datei in üblicher Weise übersetzt wird, landet die daraus entstehende OPO-Datei in dem Ordner, in dem sich auch die OPL-Datei befindet. Was aber passiert beim Übersetzen eines OPL-Programmes in eine Applikation? Das Produkt der Übersetzung landet diesmal automatisch im Verzeichnis SYSTEM\APP\.. Um eine bessere Übersicht zu gewährleisten, wird jeder Applikation dann in \APP auch noch ein eigener Ordner spendiert, der den Namen der OPL-Datei trägt. Achtung – Zum Systemordner haben Sie nur Zugang, wenn Sie vom Systembildschirm ausgehend das Menü "Einstellungen" benutzen und dort den Haken in der Zeile "Ordner 'System' anzeigen" gesetzt haben!

Im Ordner selber finden wir dann nicht nur eine Datei, sondern mindestens zwei. Der Namensanfang der beiden lautet wieder wie der Name der OPL-Datei. Die Endung nach dem Punkt sagt, dass es sich einerseits um die eigentliche Applikation handelt (\*.app) und andererseits um das Application Information File (\*.aif). Die AIF-Datei ist eine Hilfsdatei, die in eigener Form die umgesetzten Anweisungen aus dem Block APP ... ENDA aufnimmt. Zum Beispiel enthält sie auch die Grafiken für die Icons, die später in der Extras-Leiste dargestellt werden. Die Icon-Dateien müssen daher dem Programmnutzer nicht noch einmal extra mitgegeben werden.

Verteilt man später das eigene Programm, gibt man einfach den gesamten Ordner weiter. Der Anwender kopiert den gesamten Ordner bei sich wieder unter \SYSTEMAPPS\. Sofern dann alle Dateien im korrekten Verzeichnis untergebracht sind, installiert sich die Applikation ohne sein Zutun in der Extras-Leiste und lässt sich auch von dort bequem starten. Den Titel, der dort zu sehen ist, bezieht es aus dem Namen hinter der APP-Anweisung.

## Einzelheiten

An einem einfachen Beispiel stelle ich nun die Details dar. Der Name der OPL-Datei sei "MeinProg"

```
APP HalloWelt, &0100001
    CAPTION "English",1
    CAPTION "Deutsch",3
    ICON "icon24.mbm"
    ICON "mask24.mbm"
    ICON "icon32.mbm"
    ICON "mask32.mbm"
    ICON "icon48.mbm"
    ICON "mask48.mbm"
ENDA

PROC hw:
    PRINT "Hallo Welt"
    GET
ENDP
```

Geklärt ist bereits, dass sich der Name hinter APP als Titel in der Extras-Leiste wiederfindet. Die Long-Integerzahl dahinter ist die UID, die Unique IDentification number, die Programme unverwechselbar macht. Der Sinn ist, dass Programme, die mit Dateien arbeiten (wie WORD), mit diesen Dateien zuverlässig verknüpft werden können. In der Praxis geht das dann so: Sie tippen im Systembildschirm z.B. auf eine WORD-Textdatei, wodurch zugleich die zugehörige Applikation – hier eben WORD – startet. Damit es kein Durcheinander gibt, bedarf es also dieser UID.

Für den privaten Bereich steht nach Vorgaben von Psion/Symbian der Bereich &01000000 bis &0FFFFFFF beliebig zur Verfügung. Sowie Sie mit Ihren Programmen an die Öffentlichkeit gehen wollen, besorgen Sie sich (kostenfrei) bei Symbian eine der weltweit nur einmal vorkommenden Nummern, die in den Bereich ab &10000000 fallen. Somit wird Ihr Programm nie mit anderen öffentlichen Programmen kollidieren können. Die derzeit gültige Symbian E-Mail-Adresse lautet:

uid@symbiandevnet.com

Sollte sich das geändert haben, suchen Sie die Information im Entwicklerbereich auf der Website <http://www.symbian.com/>. Geben Sie dort in die Suchmaske einen Begriff wie " EPOC UID" ein, werden Sie bestimmt ein hilfreiches Info-Dokument finden. Bei der Bestellung von UIDs geben Sie in der E-Mail an:

- im Betreff: "UID Request"

- im weiteren Text:

Ihren Namen oder den Programm-Namen

Ihre E-Mail-Adresse

die Anzahl der benötigten UIDs (lieber eine mehr!)

## CAPTION

Wenn der Titel, der in der Extras-Leiste unter dem Icon angezeigt werden soll, einen Bezug zu der sprachlich lokalisierten Version des Psion haben soll, müssen Sie das Schlüsselwort CAPTION verwenden. In diesem Fall wird der Titel aus dem hinter CAPTION stehenden String genommen. Die Zahl am Ende ist die Kennung,

welcher Titel auf welchem Gerät benutzt werden soll. Im Beispiel steht "1" für ein englisch und "3" für ein deutsch lokalisiertes Gerät. Den Effekt können Sie sich ansehen, wenn Sie eine auf einem Serie5 geschriebene Applikation auf dem EPOC32-Emulator unter Windows laufen lassen – der Emulator ist praktisch ein englisch lokalisiertes Gerät.

Die Lokalisierungs-codes:

Englisch 1	Amerikanisch 10	Belgisch Flämisch 19
Französisch 2	Schweizer Französisch 11	Australisch 20
Deutsch 3	Schweizer Deutsch 12	Belgisch Französisch 21
Spanisch 4	Portugiesisch 13	Österreichisch 22
Italienisch 5	Türkisch 14	Neuseeländisch 23
Schwedisch 6	Isländisch 15	Internat. Französisch 24
Dänisch 7	Russisch 16	
Norwegisch 8	Ungarisch 17	
Finnisch 9	Niederländisch 18	

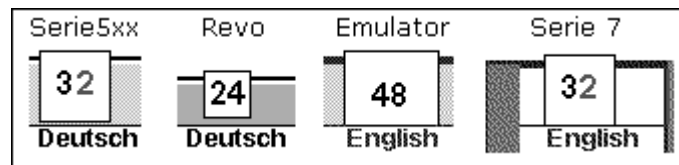


Abb.: Unterschiedliche Lokalisierung und Icon-Größen bei verschiedenen Geräten

## ICONS

Icons sind die kleinen Bildchen, die im Systembildschirm vor dem Dateititel und in der Extras-Leiste zu sehen sind.

Soll alles fachgerecht sein, werden drei verschieden groß bemessene Grafiken (24\*24, 32\*32 und 48\*48) und jeweils dazu passende Masken benötigt. Der Bedarf erklärt sich aus der Skalierbarkeit (Zoom) der Bildschirm-darstellung und aus den physisch unterschiedlichen Abmessungen der Bildschirme des Gerätespektrums. Stellen Sie nur eine Größe bereit, rechnet sich das Betriebssystem die fehlenden Größen zurecht, das Ergebnis ist allerdings selten berauschend.

Sie benötigen zur Erstellung keine gesonderte Software, denn das Programm SKIZZE unterstützt Sie bei der Arbeit. Nur der Revo hat dieses Programm nicht von Haus aus installiert, Sie können es sich jedoch kostenlos von Epocware besorgen (<http://www.epocware.com/sketch.html>).

Die nach Ihren Vorstellungen erzeugten Grafiken mit den genannten Abmessungen exportieren Sie als EPOC-Grafik mit der Endung \*.MBM. Die Bezeichnungen wählen Sie passend nach dem Schema, das oben angegeben ist: "icon24.mbm", usw.

Für das Aussehen der Icons ist es wichtig zu wissen, welche Rolle die "Maske" dabei spielt – betrachten Sie die Abbildung.

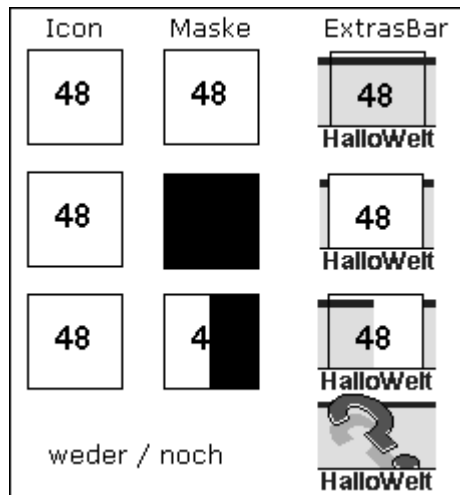


Abb.: Die prinzipielle Wirkung der Maske

Das eigentliche Icon lässt sich ohne Umschweife erzeugen, egal, ob schwarz/weiß/grau oder auch farbig (Serie7/netBook). Prinzipiell lässt sich nun diese Grafik auch als Maske verwenden, liefert aber nicht immer ein ansehnliches Ergebnis. Betrachten Sie die vorhandenen Icons auf Ihrem Gerät näher, werden Sie feststellen, dass es neben "durchsichtigen" auch "undurchsichtige" Bereiche gibt. Hier setzt die Wirkung der Maske ein.

Die Maske ist eine schwarz/weiß-Grafik, die dieselben Abmessungen wie das Icon haben muss. Sie sagt dem Computer, welche Pixel der Icon-Grafik zu setzen sind und welche nicht. An Stellen, die in der Maske schwarz sind, wird der Inhalt des zugehörigen Rasterpunktes des Icons gezeichnet, egal ob schwarz, weiß oder farbig. Die unterliegende Fläche wird deshalb an dieser Stelle überdeckt. Die weißen Stellen in der Maske sorgen dafür, dass keine Bildpunkte gezeichnet werden, der Hintergrund bleibt sichtbar, das Icon wirkt durchsichtig.

Haben Sie die Icons und Masken erstellt, verschieben Sie sie in den Ordner, in dem sich auch der Quelltext befindet, dann können Sie die Argumente hinter ICON so verwenden, wie oben angegeben. Ansonsten setzen Sie einfach noch den korrekt benannten Pfad zur Grafik mit ein.

Das Dateiformat der Bilder heißt "MBM" (MultiBitMap). In einer Datei können also auch mehrere Icon-/Maske-Paare untergebracht werden. So käme man mit einer einzigen ICON-Zeile aus, z.B.:

```
ICON "iconmask.mbm"
```

Das bedeutet aber, man muss die mit SKIZZE erzeugten Einzelbilder in eine solche Datei "stopfen". Psion liefert dazu auf der PsiWin CD das DOS-Programm "bmconv.exe" mit (auch zu finden im OPL-SDK). Man erfährt die anzuwendenden Befehlsparameter, wenn man in einem DOS-Fenster einfach "bmconv.exe" aufruft. Eine andere Lösung ist das Benutzen der hervorragenden Shareware (25 US\$) "MBMview" von Lieuwe de Vries, (hoffentlich noch) zu finden unter

<http://ourworld.compuserve.com/homepages/platodva/psion.htm>

Die Prozedur ist insbesondere mit dem DOS-Programm dermaßen umständlich, dass sich der Aufwand nicht lohnt. Da nach der Übersetzung unserer Programme in Applikationen die Icon-Dateien keine Belastung mehr darstellen (man muss auf sie nicht mehr wie auf eine Herde Schafe aufpassen, denn sie sind in der AIF-Datei aufgegangen), hat es sich bewährt, sie einzeln so zu verwenden, wie es das Beispiel zeigt.

Solange Sie in der Erprobungsphase sind und sich der Aufwand für die Icons noch nicht lohnt, können Sie natürlich auch den bequemen Weg gehen. Schreiben Sie nur diese beiden Zeilen als "Applikationsmacher":

```
APP applikationsname
ENDA
```

Als Standard-Icon bekommen Sie dann das schrägliegende Fragezeichen in der Extras-Leiste zu sehen.

## FLAGS

Die Benutzung von *FLAGS* ist nur für Programme interessant, die eigene Dateien erstellen können, wie das beispielsweise die Applikation WORD kann. In diesem Falle fügt man irgendwo zwischen *APP* und *ENDA* diese Zeile ein:

```
FLAGS 1
```

Gibt man statt dessen den Wert 2 an, erscheint die Applikation nicht in der Extras-Leiste. Die Benutzung dieser Option ist in den meisten Fällen nicht sinnvoll. Für Ihre ersten Programme ist es praktisch und erlaubt, *FLAGS* einfach wegzulassen.

Wenn Sie das oben aufgestellte Programm übersetzen lassen, sehen Sie in \SYSTEM\APPS\ den neuen Ordner "MeinProg", in dem sich die Dateien "MeinProg.aif" und "MeinProg.app" befinden. Auf einem Serie 5mxPRO sieht das dann aus, wie in der Abbildung oben dargestellt. Als Icon vor den Dateinamen wird die 24er Größe verwendet, in der Extras-Leiste kommt die 32er Größe zum Einsatz. Als Maske haben wir jeweils ein schwarzes Quadrat in der maximal zulässigen Größe gewählt. Der Programmtitel wird aus dem CAPTION-Eintrag für ein deutsch lokalisiertes Gerät ("Deutsch") genommen.

# 16 SIS-Dateien erstellen

---

## Wie man SIS-Dateien erstellt

SIS-Dateien sind die moderne Verpackung für fertige Anwendungen. Deren einzige Aufgabe besteht darin, dem Anwender, der die Applikation irgendwie zum Laufen bringen muss, bei der Installation möglichst unkompliziert zur Hand zu gehen. Und das klappt nach gewisser Vorbereitung auch ganz gut. Zumeist reicht am Windows-PC ein Klick auf die Datei und PsiWin, sofern installiert, verbindet zum angeschlossenen Psion, oder der Tipp mit dem Stift auf die Datei im Systembildschirm veranlasst die automatische Installation direkt auf dem Psion.

Das Erstellen der SIS-Dateien ist nicht besonders schwierig, aber doch reichlich umständlich, da einmal mehr mit der "Ausführen"-Befehlszeile im Windows Startmenü gearbeitet werden muss. Doch bevor Sie richtig loslegen können, müssen Sie sich das Hilfsprogramm Makesis.exe besorgen, das die SIS-Datei mit Hilfe des Windows-PCs zusammenstellt. Makesis.exe ist Bestandteil des OPL-SDKs (s. Kapitel "Einführung").

## Package-Datei anlegen

Eine Package-Datei (\*.PKG) ist eine Steuerdatei, die von Makesis.exe benötigt wird. Sie enthält Informationen, wie die SIS-Datei zusammengesetzt ist und einige zusätzliche Parameter, z.B. Angaben zur Sprachversion. Diese Datei soll sich im selben Verzeichnis auf dem PC befinden, wie die Makesis.exe; die wiederum darf in einem beliebigen selbstgewählten Verzeichnis stehen. Der Inhalt der Package-Datei lässt sich mit einem schlichten Texteditor anlegen und verändern – das Windows-Notepad ist dafür gut genug.

Ein Beispiel:

Sie haben ein Programm "Hallo" geschrieben, das aus drei Teilen besteht, die sich später auf dem Psion-Laufwerk C: oder D: im Verzeichnis \System\Apps wiederfinden sollen. Die Dateien heißen "hallo.app", "hallo.aif" und "hallo.hlp". Der Bequemlichkeit halber legen Sie auf dem PC das Verzeichnis C:\MAKE an, in das Sie sowohl Makesis.exe als auch die Dateien legen, die zu Ihrem Programm gehören. Das müsste nicht sein, spart aber im folgenden etwas Schreiarbeit. Im gleichen Verzeichnis erzeugen Sie nun eine Textdatei auf die Windows-übliche Art und geben ihr den Namen "hallo.pkg". Dort soll drinstehen:

```
#{"Hallo"} , (0x100002c3) , 2, 50
"hallo.app" - "!\System\Apps\Hallo\hallo.app"
"hallo.aif" - "!\System\Apps\Hallo\hallo.aif"
"hallo.hlp" - "!\System\Apps\Hallo\hallo.hlp"
```

Zeile 1 enthält den Programm-Namen, die UID-Nummer (hexadezimal) sowie die Programmversionsnummer (als Angabe von Haupt- und Unternummer, hier also 2.50 ). Die Angaben müssen mit denen aus dem OPL-Programm im Abschnitt *APP/ENDA* übereinstimmen!

Die UID-Nummer (unique identifier) holt man sich per kostenlosem Antrag via EPOC World (s. Kapitel "Applikationen").

Die Folgezeilen beinhalten jeweils die Benennung der Quelldatei (im Bedarfsfall inklusive Pfad) und die Zieldatei inklusive Pfad für den Psion. Das Ausrufungszeichen steht als Platzhalter für das Psion-Laufwerk. Bei der späteren Installation hat man deshalb die Wahl, wohin das Programm kopiert wird. Für spezielle Belange setzt man dort die Laufwerksangabe gleich konkret ein, also C oder D.

Halten Sie sich an die Schreibweise mit den Anführungszeichen; die Angaben für jede Datei müssen zusammenhängend in einer Zeile stehen!

Das Beispiel "hallo.pkg" ist wohl so ziemlich das kürzeste, was man machen kann. Zur "makesis.exe"-Datei gehört noch ein recht ausführlicher dokumentierender Text, der für alle weiteren Feinheiten eine Lösung aufzeigt.

## SIS-Datei erzeugen

Begeben Sie sich unter Windows über das Startmenü zur "Ausführen"-Befehlszeile. Tragen Sie dort für unser Beispiel ein:

```
C:\make\makesis hallo.pkg <Enter>
```

Das generiert die SIS-Datei.

Bei diesem Vorgehen wird das DOS-Fenster kurz aufgerufen. Eventuelle Fehlermeldungen werden hier angezeigt. Da sich das Fenster aber sofort wieder schließt, bleiben einem diese Informationen vorenthalten. Abhilfe schaffen Sie, indem Sie eine kurze Batch-Datei schreiben, bei deren Aufruf (Doppelklick reicht) das DOS-Fenster am Ende offen bleibt.

Geben Sie der Batch-Datei den Namen "Make-it.bat"; der Inhalt:

```
@ECHO OFF
c:\make\makesis.exe hallo.pkg
@ECHO Fertig!
```

## Politur

Da man nun der SIS-Datei nicht mehr ansieht, was sie enthält, erzeugen Sie noch eine Datei "liesmich.txt", die das notwendige Wissen für den Nutzer/Anwender enthält. Verpacken Sie schließlich die SIS-Datei und "liesmich.txt" gemeinsam in eine komprimierte Zip-Datei, dann ist Ihre Arbeit sauber getan und versandfertig.

## 17 Aussichten

---

Bis hierher und nicht weiter geht unsere gemeinsame Entdeckungsreise. Das bedeutet jedoch nicht, dass alle Möglichkeiten vom OPL erschöpft sind. Eine ganze Reihe von Programmier-Aufgaben können Sie mit dem bisher angebotenen Wissen bereits umsetzen, aber manchmal reicht's eben auch noch nicht. Die folgende Übersicht stellt grob zusammen, was Ihnen OPL noch zu bieten hat. Finden Sie also heraus, ob es sich für Sie lohnt, den eingeschlagenen Weg weiter zu gehen ...

### Input-/Output- (I/O-) Befehle

Bereits mit Ihrem jetzigen Wissenstand können Sie Daten in Ihre Programme hineinholen (input) und auch wieder ausgeben (output). Egal, woher die Daten stammen (aus einer Datei, von einem Modem, von der Tastatur ...) oder wohin sie geliefert werden (zum Bildschirm, an einen Drucker, an eine Datei, ...), es handelt sich immer um I/O-Prozesse, die dabei stattfinden. Die Standard-Befehle sind allerdings etwas spröde – meist lassen sie nur wenig Spielraum für Sonderwünsche. Beispiel:

Mit `LPRINT datei$` können Sie ganz bequem einen Text in eine Datei speichern. Das Dateiformat steht dabei weitgehend fest. Ein spezielles Format auf diese Art zu realisieren ist schwer möglich. Hier ließen sich mit Hilfe von I/O-Befehlen Dateien mit beliebigem Inhalt anlegen, beschreiben und wieder auslesen.

Zu den I/O-Befehlen gehören `IOOPEN`, `IOREAD`, `IOWRITE` und eine ganze Reihe weiterer (s. im alphabetischen Teil unter "I/O Befehle").

### Asynchrones Arbeiten

Anweisungen, die asynchrones Arbeiten erlauben, werden immer dann eingesetzt, wenn ein Vorgang angestoßen werden soll, auf dessen Ergebnis man aber nicht warten kann, weil gleichzeitig auch noch andere Aktionen weiterlaufen müssen.

Stellen Sie sich vor, Sie erwarten Daten über ein Modem. Wenn Sie das Problem auf herkömmliche Weise lösen, müssten Sie solange warten, bis die Daten ankommen und übertragen worden sind. In der Wartezeit wäre Ihr Gerät aber für alle weiteren Aktionen gesperrt. Sie könnten das Programm nicht einmal regulär abbrechen, weil die Tastatur nicht reagieren würde. Hier helfen asynchron arbeitende Befehle weiter. Zu Ihnen gehören unter anderem: `IOA`, `KEYA`, `GETEVENTA32`

### Gehobene Programmentwicklung

Wenn Sie eine ganze Weile in OPL programmiert haben, werden auch Module entstehen, die Sie wiederverwenden können. Solche Module laden Sie mit `LOADM` zum bestehenden Programm hinzu und erleichtern sich so die Arbeit.

Nun kommt es aber vor, dass Sie Module einbinden müssen, die zunächst nur im Plan, aber noch nicht als Quelltext existieren. Mit `DECLARE EXTERNAL` und `EXTERNAL` besteht dann trotzdem schon in der Entwicklungsphase die Möglichkeit, Variablen-Verträglichkeiten zur Übersetzungszeit zu prüfen und Prototypen einzubinden.



## OPX-Module

OPX-Module sind in C/C++ programmierte Programmstückchen, deren Funktionen sich nach Einbindung einfach aufrufen lassen. Mit Ihrer Hilfe werden die OPL-Funktionalitäten deutlich erweitert. Auf diese elegante Art und Weise ist sogar der Zugriff auf Betriebssystem-Funktionen möglich, vorausgesetzt, ein passendes OPX-Modul existiert.

Die Einbindung von OPX-Modulen geschieht mit dem `INCLUDE`-Befehl. Das OPX-Modul wird dabei über eine sogenannte Header-Datei dem OPL-Programm "bekannt" gemacht.

Die EPOC32-Psions verfügen bereits "von Geburt an" über 5 eingebaute OPX-Module:

Date.opx (für Datum- und Zeit-Handling)

System.opx (zum Aufruf von Systemfunktionen wie Zuschalten der Hintergrundbeleuchtung und die Steuerung von Applikationen)

Bmp.opx (Bitmap-Verwendung für Buttons und Sprites (animierte Grafiken))

Dbase.opx (erweiterte Konfigurations- und Nutzungsmöglichkeiten von Datenbanken)

Printer.opx (unterstützt Funktionen wie Formatieren und Druckvorschau)

Sollten Ihnen diese Möglichkeiten noch immer nicht reichen, sind Sie ein Kandidat, der C/C++ lernen sollte ...

# 18 Index

---

## @

---

@-Operator · 30

## A

---

Abbruchbedingung · 23  
 Applikationen · 73  
 Arrays · 10  
 asynchrones Arbeiten · 80

## B

---

BASIC · 4  
 Bildschirm-Ausgabe · 15  
 Bitmaps · 59  
 Bubble-Sort · 29  
 Buttons in Dialogen · 41

## C

---

Cursor positionieren · 43

## D

---

Dateiverwaltung · 68  
 Datenbanken · 62  
 Datenbanken und Tabellen · 63  
 Dateneingabe · 14  
 Dialoge · 37  
 Dialoge, nutzerfreundliche · 41  
 DO ..UNTIL-Schleife · 23

## E

---

Editor · 5  
 Emulator · 5

## F

---

Farben · 48  
 Fehler behandeln · 70  
 Fehler ignorieren · 69  
 Fehler zur Programmlaufzeit · 69  
 Fehleranalyse · 70  
 Fehlerarten · 69

Fenster · 54  
 Fließkommazahlen · 9  
 Font & Stil · 16  
 Funktionen · 13, 27

## G

---

Gleitkommazahlen · 9  
 Globalvariablen · 11  
 GLOBAL-Variablen · 27  
 Grafik entfernen · 46  
 Grafik kopieren · 46  
 Grafik verschieben · 46  
 Grafik, einfache · 43  
 Grafik, schnellere · 47

## I

---

I/O Befehle · 80  
 Icons · 75  
 IF-Entscheidungen · 18  
 Integerzahlen · 9

## K

---

Konstanten · 27

## L

---

Lokalvariablen · 11  
 Long-Integers · 9  
 Lotto · 28

## M

---

Masken · 75  
 Menü-Definition · 32  
 Menüpunkt · 32  
 Menüs · 32  
 Menüs, kaskadierte · 34

## N

---

Numerische Variablen · 9

---

**O**

Operatoren, bitweise · 20  
 Operatoren, logische · 20  
 OPL · 4  
 OPL-Programme · 6  
 OPL-SDK · 5  
 OPO-Datei · 7  
 OPX-Module · 81

---

**P**

Package-Datei · 78  
 Parameter · 26  
 PopUp-Menüs · 36  
 Programmierumgebungen, andere · 5  
 Programm-Schleifen · 22  
 Programm-Verzweigungen · 18  
 Prozeduren · 26

---

**R**

Rahmen · 56

---

**S**

Schatten · 56  
 Schleifen · 22  
 Shortcut-Tasten · 32  
 SIS-Files · 78  
 String-Variable · 8  
 suchen & finden · 66

Syntax-Fehler · 69

---

**T**

Tastaturkürzel · 32  
 Text, grafischer · 50

---

**U**

Unterprogramm · 6, 26, 27

---

**V**

Variable, String- · 8  
 Variablen · 8  
 Variablen deklarieren · 11  
 Variablen, globale · 11  
 Variablen, lokale · 11  
 Variablen, numerische · 8  
 Variablenfeld · 10  
 Variablennamen · 8  
 Variablentypen · 9  
 Verzweigungen · 18

---

**W**

WHILE..ENDWH-Schleife · 23

---

**Z**

Zahlenbereich · 9



**Ulrich Krinzner**

## **OPL-Programmierung für Psion-Handhelds**

Serie 5/mx/mxPro/, Serie 7, netBook und Revo

### **- Teil 2: Die OPL-Befehle -**

**Copyright © Ulrich Krinzner 2004**

**E-Mail: [krinzner@snafu.de](mailto:krinzner@snafu.de)**

Privatveröffentlichung, 1. Auflage, 04/2004

Dieses Buch wurde im Interesse einer noch immer großen Fan-Gemeinde produziert. Obwohl schon längst Rost und Motten die Produktionsstätten unserer Lieblinge zerfressen haben dürften, ist doch das Interesse an der leichten Programmierbarkeit dieser Geräte ungebrochen. In deutscher Sprache gedruckte Werke zum Thema sind selten. Das vorliegende Buch füllt diese Lücke für Geräte mit den Betriebssystemen EPOC32 r3 und r5, zuletzt auch als Symbian-OS 5 bezeichnet. Dieser Teil enthält die umfangreiche Beschreibung des gesamten Befehlsatzes. Zum besseren Verständnis bei der Analyse von Programmen auf EPOC 16-Basis sind auch die "alten" Befehle mit erfasst.

Mein Dank gilt der Psion Teklogix GmbH, Jakob-Kaiser-Strasse 3, D-47887 Willich, die mir gestattet hat, die Befehlsliste aus dem deutschen Programmierhandbuch zum Serie 3a mit zu verwenden. Ich habe partiell davon Gebrauch gemacht.

Im Sinne möglichst geringer Kosten für Interessenten, wurde mehr Wert auf den Inhalt als auf raffinierte Buchdruckerkunst gelegt.

Das vorliegende Werk ist in allen seinen Teilen urheberrechtlich geschützt. Alle Rechte vorbehalten. Teilweise oder vollständige Reproduktionen – in welcher Form auch immer-, jedwede Veröffentlichung, Übersetzung sowie Speicherung in elektronischen Medien ist nur mit ausdrücklicher schriftlicher Genehmigung des Autors zulässig.

Die Warenzeichen, Gebrauchsnamen, Handelsnamen usw. aller erwähnten Erzeugnisse und Firmen werden ausdrücklich als solche anerkannt. Ihre Verwendung dient ausschließlich der Information des Lesers und keiner kommerziellen Absicht.

Hier vorgestellte Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind lediglich für Lehrzwecke bestimmt.

Das vorliegende Material wurde unter großer Sorgfalt zusammengestellt, trotzdem ist Fehlerfreiheit nicht zu gewährleisten. Der Autor wird für solche möglichen Fehler und deren Folgen weder juristische Verantwortung noch sonstige Haftungen übernehmen.

# Inhaltsverzeichnis

---

<b>INHALTSVERZEICHNIS .....</b>	<b>2</b>
<b>OPL-BEFEHLE (ÜBERBLICK) .....</b>	<b>3</b>
VORBEMERKUNGEN .....	3
OPL32-BEFEHLE .....	3
<i>Programmsteuerung</i> .....	3
<i>Bildschirm- und Tastatursteuerung</i> .....	4
<i>Dateien</i> .....	5
<i>Speicherverwaltung, -adressierung, -nutzung</i> .....	6
<i>Drucken</i> .....	6
<i>Zahlen</i> .....	6
<i>Strings</i> .....	7
<i>Datum und Uhrzeit</i> .....	7
<i>Grafik</i> .....	8
<i>Menüs</i> .....	9
<i>Dialoge</i> .....	9
<i>Statusmeldungen</i> .....	9
<i>Applikationen</i> .....	10
<i>Fortgeschrittene Programmierung</i> .....	10
OPL16-BEFEHLE (NICHT FÜR EPOC32 VERWENDEN) .....	11
<b>OPL-BEFEHLE (ALPHABETISCHE LISTE) .....</b>	<b>13</b>
VORBEMERKUNGEN .....	13
ALPHABETISCHE LISTE DER OPL-BEFEHLE .....	15
<b>ANHANG 1, ZEICHENTABELLE &amp; CODES .....</b>	<b>112</b>
DIE TEXT-ZEICHEN .....	112
DIE STEUERZEICHEN .....	114
SPEZIALTASTEN .....	115
SONDERFALL .....	116
<b>ANHANG 2, BINÄR- UND HEX-ZAHLEN .....</b>	<b>117</b>
DEKADISCHES SYSTEM .....	117
BINÄRES SYSTEM .....	117
HEX-ZAHLEN .....	118
<b>ANHANG 3, FONTSLISTE .....</b>	<b>120</b>
<b>ANHANG 4, OPL-FEHLERNUMMERN .....</b>	<b>122</b>
<b>INDEX .....</b>	<b>127</b>

# OPL-Befehle (Überblick)

---

## Vorbemerkungen

Die OPL-Schlüsselwörter ("Anweisungen"), die Sie verwenden müssen, um Programme nach Ihren Vorstellungen zu erstellen, teilt man grob in zwei Gruppen ein in:

**Funktionen:** OPL-Anweisungen, die einen Wert zurückliefern können,

**Befehle:** Anweisungen, die keinen Wert zurückliefern, sondern "nur" etwas bewirken.

Auch Funktionen können in gewissen Fällen wie "normale" Befehle verwendet werden, Sie brauchen lediglich den Rückgabewert zu ignorieren. Im alltäglichen Sprachgebrauch werden daher oft die Begriffe "Funktionen", "Befehle", "Anweisungen" und "Schlüsselwörter" wild durcheinander benutzt ...

Sie sehen im Folgenden eine Liste aller OPL-Anweisungen. Die Gliederung der Liste unterstützt Sie dabei, für jede Aufgabe den richtigen Befehl möglichst schnell zu finden und dann im alphabetischen Teil nachzuschlagen.

Beim Übergang von den 16-Bit- zu den 32-Bit-Organizern, also von EPOC16 (SIBO) zu EPOC32, ist einerseits eine Reihe von Befehlen weggefallen, andererseits sind etliche neue hinzugekommen. Da das vorliegende Buch sich hauptsächlich an die Benutzer der EPOC32-Plattform wendet, liegt auch der Schwerpunkt der ersten Liste bei den "OPL32"-Befehlen. In einer zweiten Liste sind die für EPOC32 nicht mehr verwendbaren Befehle aufgeführt. Dadurch haben Sie die Gelegenheit, auch ältere Quelltexte zu verstehen.

## OPL32-Befehle

### Programmsteuerung

<b>Prozeduren, Programmeinstellungen</b>	
PROC .. ENDP	Definiert Start und Ende einer Prozedur
CONST	Deklariert eine (globale) Konstante
INCLUDE	andere Dateien in Quell-Programm einbinden
DECLARE OPX	Deklariert ein OPX-Modul
SETFLAGS	Nimmt Einfluss auf das Programmverhalten
CLEARFLAGS	Löscht alle mit SETFLAGS gesetzten Flags

<b>Schleifen, Verzweigungen, Sprünge</b>	
DO .. UNTIL	Eine Befehlsfolge wiederholen
WHILE...ENDWH	Eine Befehlsfolge wiederholen
BREAK	Bricht eine Schleife ab und springt zu deren Ende
CONTINUE	Setzt Schleife unmittelbar am Prüfbefehl fort
IF .. ELSEIF .. ELSE .. ENDIF	Programmverzweigung je nach Bedingung
GOTO	Springt zu einer Markierung
VECTOR ..ENDV	Springt zu einer Markierung aus einer Liste
@-Operator	Springt zu einem Unterprogramm
RETURN	Zur rufenden Prozedur zurückkehren
STOP	Das Programm beenden

<b>Fehlerbehandlung</b>	
RAISE	Einen Fehler künstlich hervorrufen
REM	Bemerkungen ins Programm einfügen
ONERR	Eine Routine zur Fehlerbehandlung installieren
TRAP	Fehler für einen Befehl ignorieren
ERR, ERR\$, ERRX\$	Aufgetretenen Fehler untersuchen (Fehlernummern und -text)

## Bildschirm- und Tastatursteuerung

CLS	Das Textfenster löschen
SCREEN	Größe/Position eines Fensters festlegen
SCREENINFO	Information über das Textfenster auslesen
AT	Den Text-Cursor positionieren
CURSOR ON / OFF	Den Cursor zeigen / verstecken
gUPDATE	Aktualisierung des Bildschirms kontrollieren
FONT, STYLE	Font und Schriftstil für das Textfenster setzen
GET, GET\$	Auf Tastendruck warten und Wert zurückliefern
KEY, KEY\$, GET, GET\$	Tastendruck auslesen
EDIT	Einen String anzeigen und editieren lassen
INPUT	Text, Zahlen, etc von der Tastatur einlesen
PRINT	Text, Zahlen, etc. ausgeben
PAUSE	Programmablauf f. bestimmte Zeit unterbrechen
BEEP	Einen Ton ausgeben
KMOD	Modifizierer-Tasten auslesen
ESCAPE ON/OFF	Abbruch über PSION-ESC an/aus
OFF	Computer ausschalten



## Dateien

<b>Generelle Dateiverwaltung</b>	
COPY	Datei kopieren
DELETE	Datei/ Datenbanktabelle löschen
RENAME	Datei umbenennen
EXIST	Auf Existenz einer Datei überprüfen
DIR\$	Vorhandene Dateien auflisten
PARSE\$	Dateispezifikation auslesen

<b>Verzeichnisse verwalten</b>	
MKDIR	Verzeichnis erstellen
SETPATH	Verzeichnis setzen
RMDIR	Verzeichnis löschen

<b>Datenbanken</b>	
CREATE	Eine neue Datei erstellen
OPEN, OPENR, CLOSE	Eine Datei öffnen oder schließen
USE	Auf eine bereits geöffnete Datei wechseln
COMPACT	Komprimiert DB-Datei
APPEND	Datensatz hinzufügen
UPDATE	Datensatz ändern
ERASE	Datensatz löschen
FIRST, LAST, NEXT, BACK, POSITION	Datensatz anwählen
FIND, FINDFIELD	Datensatz suchen
COUNT	Datensätze zählen
EOF	Auf Dateiende überprüfen
POS	Aktuelle Datensatznummer herausfinden
RECSIZE	Größe des aktuellen Datensatzes herausfinden
DELETE	Löscht eine Datenbanktabelle
INSERT	Setzt einen neuen leeren Datensatz in die DB ein
MODIFY	Modifiziert den aktuellen Datensatz
PUT	Überträgt Änderungen in die Datenbank
CANCEL	Verwirft gemachte Änderungen
BOOKMARK	Setzt Marke (Bookmark) auf aktuellen Datensatz
GOTOMARK	Springt zur Marke u. macht den DS zum aktuellen
KILLMARK	Entfernt eine Marke
BEGINTRANS	Beginnt eine Transaktion mit der aktuellen DB
COMMITTRANS	Führt die Transaktion aus
INTRANS	Prüft, ob akt. DB-Ansicht bei einer Transaktion ist
ROLLBACK	Bricht die aktuelle Transaktion ab

<b>OPL-Prozeduren und -Module</b>	
LOADM	Ein OPL-Modul laden, um die Prozeduren zu benutzen zu können
UNLOADM	Ein OPL-Modul aus dem Speicher entfernen

## Speicherverwaltung, -adressierung, -nutzung

GLOBAL, LOCAL	Variablen deklarieren
CONST	(Globale) Konstanten deklarieren
SPACE	Freien Speicher auf logischer Einheit herausfinden
ADDR	Adresse einer Variablen ermitteln
BYREF	Übergibt Variable an OPX-Modul per Referenz
POKE...	Einen Wert an eine bestimmte Adresse schreiben
PEEK...	Einen Wert von einer bestimmten Adresse auslesen
ALLOC	Zelle auf dem Heap reservieren
FREEALLOC	Zelle auf dem Heap freigeben
ADJUSTALLOC	Teil einer Zelle einfügen / löschen
REALLOC	Größe einer Zelle ändern
LENALLOC	Länge einer reservierten Zelle herausfinden
EXTERNAL	Deklariert Variablen und Prozeduren als extern
DECLARE EXTERNAL	Erlaubt, nicht deklar. Variabl. & Prozed. zu finden

## Drucken

LOPEN	Ausgabeeinheit festlegen und für Ausgabe öffnen
LCLOSE	Ausgabeeinheit wieder schließen
LPRINT	Drucken

## Zahlen

<b>Trigonometrie</b>	
COS, SIN, TAN, ACOS, ASIN, ATAN	Funktionen
RAD, DEG	Zwischen Grad und Bogenmaß konvertieren

<b>Allgemeine numerische Funktionen</b>	
EXP	Potenzieren
LN, LOG	Logarithmus
PI	Pi (Konstante)
SQR	Quadratwurzel
RND, RANDOMIZE	Zufallszahlen
UADD, USUB	Arithmetik mit vorzeichenfreien Integer-Werten

<b>Statistische numerische Funktionen</b>	
MAX, MIN	Größten, kleinsten Wert finden
MEAN	Durchschnitt bilden
SUM	Summe bilden
STD, VAR	Standardabweichung, Varianz

<b>Zahlenformat ändern</b>	
ABS, IABS	Absolutwert
INT, INTF	Ganzzahliger Anteil
FLT	Konvertiere Integer nach Floating Point
HEX\$	Konvertiere Integer in hexadezimalen String
FIX\$, GEN\$, SCI\$, NUM\$	Konvertiere eine Zahl in einen String
EVAL, VAL	Konvertiere einen String in eine Zahl

## Strings

LEFT\$, MID\$, RIGHT\$	Zeichen aus einem String kopieren
REPT\$	Einen String wiederholen
UPPER\$, LOWER\$	String in Groß- / Kleinbuchstaben umwandeln
LEN	Ermitteln der Stringlänge
ASC	Ermitteln des Zeichencodes des ersten Zeichens eines Strings
LOC	Ermitteln Position eines Teilstrings in einem String
VAL	Konvertiere String (aus Ziffern) in eine Zahl
FIX\$, GEN\$, SCI\$, NUM\$	Konvertiere eine Zahl in einen String
CHR\$	Einen Buchstaben aus dem Zeichencode generieren

## Datum und Uhrzeit

DATIM\$	Datum und Uhrzeit als String
SECOND, MINUTE, HOUR	Uhrzeit: Sekunde, Minute, Stunde
DAY, MONTH, YEAR	Datum: Tag, Monat, Jahr
DAYS	Anzahl der Tage zwischen zwei Daten
DOW, WEEK	Wochentag, Kalenderwoche
MONTH\$	1-12 als Monat
DAYNAME\$	1-7 als Wochentag
DATETOSECS	zwischen Zeitformaten konvertieren
SECSTODATE	zwischen Zeitformaten konvertieren
DAYSTODATE	Konvert. Tage seit d. 1.1.1900 in d. akt. Datum

## Grafik

<b>Zeichenbefehle</b>	
gAT, gMOVE	Position setzen
gLINEBY, gLINETO	Linie zeichnen
gPOLY	Folge von Linien Zeichnen
gBOX, gBORDER, gXBORDER	Rechteck zeichnen
gFILL	Rechteck füllen
gCIRCLE	Kreis gefüllt oder ungefüllt zeichnen
gELLIPSE	Ellipse gefüllt oder ungefüllt zeichnen
gINVERT	Rechteck invertieren
gCOLOR	Setzt d. Farbe f. d. Zeichenstift im akt. Fenster/Bitmap
gSETPENWIDTH	Setzt die Zeichenstiftstärke im akt. Fenster/Bitmap
gSCROLL	Rechteck verschieben
gX, gY	aktuelle Position herausfinden
gCLOCK	Laufende Uhr darstellen
gBUTTON	3D-Taste zeichnen

<b>Grafische Textdarstellung</b>	
gPRINT	Eine Liste von Ausdrücken anzeigen
gPRINTB	Text in einem leeren Feld anzeigen
gPRINTCLIP	Text sauber abgeschnitten anzeigen
gTWIDTH	Textbreite herausfinden
gXPRINT	Text unterstreichen / invertiert darstellen

<b>Schriftart setzen</b>	
gFONT	Font wählen
gLOADFONT, gUNLOADFONT	Benutzerdefinierten Font laden/entladen
gGMODE	Grafikmodus setzen
gTMODE	Textmodus setzen
gSTYLE	Schriftstil setzen

<b>Fenster und Bitmaps</b>	
gCREATE	Neues Fenster erstellen
gSETWIN	Position und Größe eines Fensters ändern
gORDER	Position setzen
gRANK	Position herausfinden
gVISIBLE	Fenster sichtbar/unsichtbar machen
gORIGINX, gORIGINY	Bildschirmposition herausfinden
gIDENTITY	Fragt die ID-Nummer eines Fensters/Bitmaps ab
gCREATEBIT	Ein Bitmap erstellen
gLOADBIT	Ein Bitmap aus einer Datei laden
gCLS	Ein Fenster/Bitmap löschen
gSAVEBIT	Ein Fenster/Bitmap in einer Datei sichern
gCLOSE	Ein Fenster/Bitmap schließen
-- Fortsetzung nächste Seite --	

<b>Fenster und Bitmaps, Fortsetzung</b>	
gUSE	Ein Fenster/Bitmap auswählen
gGREY, DEFAULTWIN	Graue Ebene in einem Fenster ein/ausschalten
gPATT	Bereich mit dem Inhalt eines anderen Fensters/Bitmaps füllen
gCOPY	Bereich aus einem Fenster/Bitmap in ein anderes kopieren
gPEEKLINE	Eine Pixelzeile aus einem Fenster auslesen
gWIDTH, gHEIGHT	Breite/Höhe eines Fensters/Bitmaps herausfinden
gINFO, gINFO32	Statusinformation über ein Bitmap/Fenster und den Cursor holen

## Menüs

mINIT	Menü initialisieren
mCARD	Menü definieren
MENU	Menü anzeigen
mCASC	Kaskadiertes Menü definieren
mPOPUP	Von MENU unabhängiges Popup-Menü aufrufen

## Dialoge

dINIT	Dialog initialisieren
dPOSITION	Dialog positionieren
DIALOG	Dialog anzeigen
dTEXT	Definiert Text für einen Dialog
dEDIT	Definiert String-Eingabefeld
dEDITMULTI	Definiert ein mehrzeiliges String-Eingabefeld
dXINPUT	Definiert Passwort-Eingabefeld
dFILE	Definiert Dateinamen-Eingabefeld
dCHOICE	Definiert Auswahlliste für einen Dialog
dFLOAT, dLONG	Definiert Eingabefeld für numerische Werte
dDate, dTIME	Definiert Eingabefeld für Datum / Uhrzeit
dBUTTONS	Definiert Tasten für einen Dialog
dCHECKBOX	Definiert eine Dialogzeile mit Checkbock
ALERT	Einfache Warnung anzeigen

## Statusmeldungen

GIPRINT	Infos anzeigen
BUSY	"Arbeite"-Meldung anzeigen

## Applikationen

APP .. ENDA	Schließt einen Definitionsblock für Applikationen ein
CAPTION	Name der Applikation, sprachabhängig
ICON	Benennt die Bitmapdateien für die Icon-Darstellung
FLAGS	Setz Flags für das Verhalten der Applikation
CMD\$, GETCMD\$	Kommandozeilenparameter auslesen
SETDOC	Macht eine Datei zum Dokument, vergibt Namen f. d. Taskliste
GETDOC\$	Liefert den Namen des aktuellen Dokumentes zurück
LOCK	Applikation für Systemmeldungen sperren

## Fortgeschrittene Programmierung

<b><i>I/O-Funktionen</i></b>	
IOOPEN	Datei beliebigen Typs öffnen
IOREAD	Aus Datei beliebigen Typs lesen
IOWRITE	In Datei beliebigen Typs schreiben
IOCLOSE	Datei beliebigen Typs schließen
IOSEEK	Innerhalb Datei beliebigen Typs positionieren
IOA, IOC	Asynchrone I/O- Funktion ausführen
IOCANCEL	Asynchrone I/O- Funktion abbrechen
IOWAIT, IOWAITSTAT32	Auf Ergebnis einer asynchronen Funktion warten
IOSIGNAL	Beendigung einer asynchronen I/O-Funktion signalisieren
IOYIELD	I/O-Handler Gelegenheit zum ablaufen geben
IOW	Synchrone I/O-Funktion ausführen
KEYA	Asynchron Tastatur auslesen
KEYC	KEYA abbrechen

<b><i>Eventhandling</i></b>	
TESTEVENT	Testet, ob ein Systemereignis (Event) stattgefunden
GETEVENT, GETEVENT32	Fragt ab, welches Systemereignis stattfand
GETEVENTA32, GETEVENTC	Systemereignisse asynchron abfragen
POINTERFILTER	Filtert Events ein oder aus

## OPL16-Befehle (nicht für EPOC32 verwenden)

<b>Datenbankdateien</b>	
COMPRESS	Eine Datei kopieren, optional an eine andere Datei anfügen und gelöschte Datensätze entfernen
ODBINFO	Betriebssysteminformation über Datenbankdatei abfragen

<b>Speicherverwaltung</b>	
CACHE	Einen Prozedurzwischenspeicher definieren
CACHETIDY	Inaktive Prozeduren aus dem Puffer entfernen
CACHEHDR	Puffer-Header auslesen
CACHEREC	Puffer-Indexeintrag lesen

<b>Betriebssystemaufrufe</b>	
CALL	Ruft eine Systeminterruptnummer auf, liefert Inhalt des AX-Registers zurück
OS	Ruft eine Systeminterruptnummer auf, liefert alle Register und Flags zurück

<b>Maschinenprogrammaufrufe</b>	
USR	Ruft eigenes Maschinenprogramm auf, liefert Zahl zurück
USR\$	Ruft eigenes Maschinenprogramm auf, liefert String zurück

<b>Handling dynamischer Bibliotheken (dynamic libraries, DYL)</b>	
LOADLIB, LINKLIB	DYL laden / linken
UNLOADLIB	DYL entfernen
FINDLIB, GETLIBH	Kategorie-Handle herausfinden

<b>Sprites</b>	
CREATESPRITE	Sprite erstellen
APPENDSPRITE CHANGESPRITE	Bitmap-Gruppen für ein Sprite definieren
DRAWSPRITE	Sprite zeichnen
POSSPRITE	Position für Sprite setzen
USESPRITE	Sprite anwählen
CLOSESPRITE	Sprite schließen

<b>Statusfenster</b>	
STATUSWIN	Statusfenster anzeigen/verstecken
STATUSWININFO	Information über das Statusfenster herausfinden
DIAMINIT	◆-Liste initialisieren
DIAMPOS	◆ in der ◆-Liste positionieren

<b>Objekt-Handling</b>	
SEND, ENTERSEND ENTERSEND0	Nachricht an Objekt senden
NEWOBJ, NEWOBJH	Erstellt neues Objekt über Kategorienummer / - handle

<b>Applikationen</b>	
TYPE	Definiert den Applikationstyp
PATH	Setzt den Pfad für Dateien, die durch die Applika- tion erstellt bzw. genutzt werden
SETNAME	Programmname setzen
EXT	Endung für Dateien, die durch die Applikation erstellt bzw. genutzt werden



# OPL-Befehle (alphabetische Liste)

---

## Vorbemerkungen

Wie schon mehrfach erwähnt, ist dieses Buch gedacht für Einsteiger, die unter und für EPOC32 arbeiten wollen. Trotzdem macht es aus verschiedenen Gründen einen Sinn, hier den gesamten Umfang des OPL-Sprachschatzes zu listen. So haben sowohl der erfahrene als auch der Programmierer, der erst am Beginn seines Weges steht, eine uneingeschränkte Nachschlagemöglichkeit. Für den Anfänger habe ich allerdings die Orientierung etwas erleichtert. Jeder Befehl hat in der gleichen Zeile rechts eine Kennzeichnung bekommen, die darauf hinweist, für welches Betriebssystem er funktioniert und welchem "Schwierigkeitsgrad" er zuzuordnen ist. Die einzelnen Kürzel stehen für:

**SIBO:** Befehl/Funktion funktioniert nur unter EPOC16/SIBO

**OPL32:** Befehl/Funktion funktioniert unter EPOC32

**OPL32!:** neu hinzugekommen ab EPOC32/v.5, funktioniert nur ab diesem Betriebssystem.

**(OPL32):** Diese Befehle/Funktionen funktionieren aus Kompatibilitätsgründen zwar noch auf den EPOC32-Geräten, ihre Verwendung wird jedoch von Symbian nicht mehr empfohlen. Zumeist gibt es dafür einen qualitativ besseren und/oder schnelleren Ersatz.

**F:** Bedeutet "nur für Fortgeschrittene" und markiert alle Befehle, die im Buch nicht ausführlich besprochen wurden. Sie sollten diese Befehle/Funktionen als Einsteiger vermeiden. Sowie Sie ein gewisses Verständnis für OPL erworben haben, können Sie die eine oder andere Anweisung mit wachen Verstand und einem gewissen Instinkt für die Folgen durchaus auch einmal testen ...

Gelegentlich unterscheiden sich die Anwendung und Wirkung von SIBO- und OPL32-Befehlen. Wo das der Fall ist, habe ich sie bei umfangreicher Beschreibung getrennt aufgeführt – die OPL32-Beschreibung erfolgt immer zuerst.

Achtung: Die Code-Beispiele sind in den meisten Fällen unvollständig und dienen nur dem besseren Verständnis der einzelnen Anweisung. Grundsätzlich müssen also alle Variablen wie üblich vorher deklariert werden und die Programmschnipsel in eine Prozedur eingebunden sein.

## Schreibweisen

Obwohl ich hier wegen der Übersichtlichkeit die Großschreibung von OPL-Schlüsselwörtern bevorzugt habe, können Sie die Befehle (fast) beliebig groß oder klein schreiben. Wenn der Platz besonders knapp war, habe ich von der Möglichkeit Gebrauch gemacht, mehrere Anweisungen in eine Zeile zu notieren. Die Anweisungen müssen lediglich durch die Folge Leerzeichen – Doppelpunkt – Leerzeichen voneinander getrennt werden. Es sind bis zu 255 Zeichen pro Zeile erlaubt. Vor einem REM muss kein Doppelpunkt stehen, sehr wohl aber wenigstens ein Leerzeichen.

## Befehle und Argumente

Auch Befehle können Argumente haben, liefern aber im Gegensatz zu Funktionen keine Werte zurück. Die Argumente sind durch ein Leerzeichen vom Befehl zu trennen. Beispiel:

```
AT x%,y%    oder
AT 15,2
```

## **Funktionen**

Funktionen liefern mit Hilfe einer Zuweisung ("=") Werte an eine Variable zurück:

```
string$ = HEX$(x&)
```

Das Argument (es können auch mehrere sein, die dann durch Kommas getrennt werden müssen) steht in Klammern unmittelbar hinter dem Schlüsselwort. Die Variable muss vom Typ her zu dem erwarteten Ergebnis passen.

Wenn Sie einen falschen Argument-Typ verwenden, wird das Argument nach Möglichkeit konvertiert. Sie können also z.B. eine Integer-Zahl anstelle einer Fließkomma-Zahl oder eine im Wert noch passende Longinteger-Zahl anstelle einer Integer-Zahl einsetzen. Wenn der übergebene Wert nicht konvertiert werden kann (z.B. zu große Longinteger-Zahl anstelle einer Integer-Zahl), wird ein Fehler ausgegeben und die Ausführung des Programms gestoppt.

Es gibt auch Funktionen, die kein Argument benötigen:

```
i% = GET
```

Rückgabewerte dürfen, wenn es angebracht ist, ignoriert werden:

```
GET
```

Funktionen können auch als Argumente anderer Funktionen oder Befehle dienen

```
PRINT LEFT$(A$,3) oder  
a = COS(ABS(x))...
```

## Alphabetische Liste der OPL-Befehle

### ABS

*OPL32 – SIBO*

`y=ABS (x)`

Gibt den Absolutwert von x zurück.

### ACOS

*OPL32 – SIBO*

`y=ACOS (x)`

Gibt den Arcus Kosinus (d.h. 1/Kosinus) von x als Winkel im Bogenmaß zurück – s.a. DEG. x darf die Werte der Kosinusfunktion nicht überschreiten ( -1 .. 1).

### ADDR

*OPL32 – SIBO*

`a&=ADDR(variable) REM in OPL32`  
`a%=ADDR(variable) REM in SIBO`

Liefert die Adresse, an der die Variable im Speicher abgelegt ist. Die Adresse ist die Anfangsadresse der Variablen. Je nach Typ werden mehrere Bytes ab dieser Adresse von der Variablen belegt.

Erl.: Die 64K-Speichereinschränkung der SIBO-Geräte ist in EPOC32-Geräten aufgehoben, entsprechend müssen anstelle der Integervariablen nun Long-Integers verwendet werden.

Siehe auch PEEK, UADD, USUB, SETFLAG

### ADJUSTALLOC

*F – OPL32 – SIBO*

`pzelln&=ADJUSTALLOC(pzell&,off&,me&) REM in OPL32`  
`pzelln%=ADJUSTALLOC(pzell%,off%,me%) REM in SIBO`

Öffnet oder schließt eine Lücke bei off& (off%) innerhalb des reservierten Speicherplatzes pcell& (pcell%). Die neue Adresse steht in pzelln& (pzelln%). Ist nicht genug Speicher vorhanden, wird 0 zurückgegeben. Eine Lücke wird geöffnet, wenn me& (me%) positiv oder geschlossen, wenn me& (me%) negativ ist.

Erl.: Die 64K-Speichereinschränkung der SIBO-Geräte ist in EPOC32-Geräten aufgehoben, entsprechend müssen anstelle der Integervariablen nun Long-Integers verwendet werden.

Der Befehl gehört zum Komplex "Dynamische Speicherverwaltung", Anfänger: Finger weg!

Siehe auch SETFLAGS

### ALERT

*OPL32 – SIBO*

`ret%=ALERT(text1$)`  
`ret%=ALERT(text1$,text2$)`  
`ret%=ALERT(text1$,text2$,but1$)`  
`ret%=ALERT(text1$,text2$,but1$,but2$)`  
`ret%=ALERT(text1$,text2$,but1$,but2$,but3$)`

Stellt eine ein- oder zweizeilige Meldung auf dem Bildschirm per "Dialog" dar und wartet auf einen Tasten- oder Stiftdruck. Dabei nehmen text1\$ und text2\$ den Text der ersten bzw. zweiten Zeile auf, ohne weitere Angaben wird ein "Weiter"-Button automatisch zur Verfügung gestellt, der auf <Esc> reagiert. Es besteht die Möglichkeit bis zu drei Buttons darzustellen, deren Beschriftung in but\$ geliefert wird. Die zu betätigende Abbruchtaste wird automatisch unter dem Button benannt, in ret% wird die Nummer der gedrückten Taste zurückgegeben:

```
but1$: <Esc>, ret%= 1
but2$: <Enter>, ret%= 2
but3$: <Leertaste>, ret%= 3
```

**ALLOC***F – OPL32 – SIBO*

```
pzell&=ALLOC(groesse&) REM in OPL32
pzell%=ALLOC(groesse%) REM in SIBO
```

Reserviert einen Speicherbereich der angegebenen Größe auf dem Heap und liefert den Zeiger auf die Zelle zurück (bzw. 0 wenn nicht genügend Speicher auf dem Heap frei ist).

Erl.: Die 64K-Speichereinschränkung der SIBO-Geräte ist in EPOC32-Geräten aufgehoben, entsprechend müssen anstelle der Integervariablen nun Long-Integers verwendet werden. Der Befehl gehört zum Komplex "Dynamische Speicherverwaltung", Anfänger: Finger weg!

Siehe auch SETFLAGS, ADJUSTALLOC, REALLOC, FREEALLOC

**APP ... ENDA***OPL32 – SIBO*

```
APP appname,uid& REM in OPL32
...
END
APP appname REM in SIBO
...
END
```

Dient der Definition einer Applikation und vergibt den Applikationsnamen. Im Kapitel "Applikationen" finden Sie weitere Informationen zu diesem Thema.

Siehe auch CAPTION, ICON, FLAGS

**APPEND***SIBO – (OPL32)*

APPEND

Hängt einen neuen Datensatz an das Ende der aktuellen Datei an. Der vorher aktuelle Datensatz wird nicht verändert. Nach Ausführung ist der letzte Datensatz der aktuelle. Es werden die aktuellen Feldinhalte verwendet. Felder, denen kein Wert zugewiesen wurde, enthalten 0 oder "", je nach Typ. Beispiel:

```
OPEN "adress",A,f1$,f2$,f3$
PRINT "Datensatz hinzufügen:"
PRINT "Enter name:",
INPUT A
```

Um den aktuellen Datensatz mit neuen Werten zu überschreiben, müssen Sie den Befehl UPDATE verwenden.

Unter OPL32 sollten anstelle APPEND und UPDATE die neueren Befehle INSERT, PUT und CANCEL verwendet werden, obwohl die "alten" noch funktionieren. APPEND erzeugt unter Umständen eine ganze Anzahl von zwischendurch benutzten und dann gelöschten Datensätzen, die mit COMPACT entfernt werden sollten!

Siehe auch INSERT, MODIFY, PUT, CANCEL, SETFLAGS

**APPENDSPRITE***F-SIBO*

```
APPENDSPRITE time%, bit$(),dx%,dy%
APPENDSPRITE time%,bit$()
```

Fügt eine Bitmap-Gruppe an das aktuelle Sprite an.

time% gibt an, wie lange die Gruppe während der Animation angezeigt wird (Einheit 1/10 s).

bit\$() enthält die Namen der Bitmap-Dateien in der Gruppe oder "", um anzuzeigen, dass kein Bitmap verwendet werden soll. Das Array muss mindestens 6 Elemente groß sein.

```
bit$(1) REM Setzen der schwarzen Pixel
bit$(2) REM Löschen der schwarzen Pixel
bit$(3) REM Invertieren der schw. Pixel
bit$(4) REM Setzen der grauen Pixel
bit$(5) REM Löschen der grauen Pixel
bit$(6) REM Invertieren der grauen Pixel
```

Alle Bitmaps einer Bitmap-Gruppe müssen die selbe Größe haben, andernfalls wird der Fehler "Argument-Fehler" (-2) ausgegeben, wenn die Gruppe an das Sprite angehängt wird.

dx% und dy% stehen für den horizontalen bzw. vertikalen Versatz der Sprite-Position zur oberen linken Ecke der Bitmap-Gruppe. Positive Werte stehen für einen Versatz nach rechts bzw. unten. Als Standard wird jeweils 0 angenommen.

Unter OPL32 werden Sprites über das eingebaute Sprite-OPX/DLL-Modul verwaltet.

Siehe auch `CREATESPRITE`, `CHANGESPRITE`, `DRAWSPRITE`, `POSSPRITE`, `CLOSESPRITE`

**ASC***OPL32 – SIBO*

```
code%=ASC(string$)
```

Gibt den Zeichencode des ersten Zeichens eines Strings string\$ zurück. Ist string\$ ein Leerstring, ist das Ergebnis Null. Das Ergebnis von ASC("Auto") ist z.B. 64. Will man lediglich den Wert eines Zeichens herausfinden, geht das unter Zuhilfenahme des %-Operators auch einfacher, z.B. code%= %A (code% ist 64)

**ASIN***OPL32 – SIBO*

```
y=ASIN(x)
```

Gibt den Arcus Sinus (d.h. 1/Sinus) von x als Winkel im Bogenmaß zurück – s.a. DEG. x darf die Werte der Sinusfunktion nicht überschreiten (-1 .. 1).

**AT***OPL32 – SIBO*

```
AT x%,y%
```

Setzt den Text-Cursor in Spalte x% und Zeile y% im Textfenster. Die linke obere Ecke des Bildschirms wird durch AT 1,1 gesetzt. Achtung: Die absolute Position in Pixel sowie die Anzahl der Spalten und Zeilen ändern sich, wenn der Font mit dem FONT-Befehl geändert wird.

```
AT 5,3 : PRINT "Eilmeldung!"
```

Siehe auch `SCREEN`, `SCREENINFO`, `CURSOR`

**ATAN**

OPL32 – SIBO

a=ATAN (x)

Gibt den Arcus Tangens (d.h. 1/Tangens) von x als Winkel im Bogenmaß zurück – s.a. DEG.

**BACK**

OPL32 – SIBO

BACK

Macht den vorherigen Datensatz in der aktuellen Ansicht der DB zum aktuellen Datensatz. Wenn der aktuelle Datensatz bereits der erste Datensatz ist, gibt es keine Veränderungen.

Siehe auch NEXT, FIRST, LAST

**BEEP**

OPL32 – SIBO

BEEP zeit%,frequ%

Gibt einen Signalton aus. Die Dauer wird über zeit% in 1/32 Sekunden angegeben (maximal 3840 = 2min). Die Tonhöhe kann über frequ% variiert werden (Frequenz=512(frequ%+1) KHz).

```
BEEP 5,300 REM gibt einen angenehmen Signalton aus
```

Wenn Sie zeit% negativ belegen, überprüft BEEP, ob das Tonsystem frei ist oder nicht. Hat gerade eine anderes Programm das System belegt, kehrt BEEP sofort zurück, ohne den Ton auszugeben. Bei positivem zeit%-Argument wartet BEEP, bis das Tonsystem frei ist.

Beispiel: (Tonleiter vom mittleren C aus)

```
PROC Tonfolge:
  LOCAL freq,n%
  REM n% relativ zum A-Ton
  n%=3 REM beginnt am C-Ton
  WHILE n%<16
    freq=440*2**(n%/12.0)
    REM A=freq 440Hz
    BEEP 8,512000/freq-1.0
    n%=n%+1
    IF n%=4 OR n%=6 OR n%=9 OR n%=11 OR n%=13
      n%=n%+1
    ENDIF
  ENDWH
ENDP
```

Sie können auch über PRINT CHR\$(7) einen einfachen, kurzen Signalton ausgeben.

**BEGINTRANS**

OPL32

BEGINTRANS

Mit BEGINTRANS wird eine "Transaktion" für die geöffnete aktuelle DB-Ansicht eingeleitet. Es können alle Datenbankbefehle normal verwendet werden, das Schreiben der Daten erfolgt aber erst nach Aufruf von COMMITTRANS

Siehe auch COMMITTRANS, ROLLBACK, INTRANS

**BOOKMARK**

OPL32

n% = BOOKMARK

Setzt eine Markierung für den aktuellen Datensatz in der aktuellen DB-Ansicht. n% ist nicht (!) die aktuelle Position im Datensatz, sondern die Durchnummerierung der Markierungen.

Siehe auch GOTOMARK, KILLMARK

**BREAK**

OPL32 – SIBO

BREAK

Verlässt eine DO...UNTIL oder WHILE...ENDWH-Schleife sofort und setzt das Programm hinter dem letzten Schleifenbefehl fort

```
WHILE 1
  ...
  IF taste%=27
    BREAK REM springt hinter ENDWH
  ENDIF
  ...
ENDWH
REM hier geht's weiter
```

Siehe auch CONTINUE

**BUSY**

OPL32 – SIBO

```
BUSY meldung$
BUSY meldung$,ort%
BUSY meldung$,ort%,verzoeigerung%
BUSY OFF
```

Zeigt den String meldung\$ in der linken unteren Ecke des Bildschirms an, bis die BUSY OFF Anweisung gegeben wird. Das wird oft benutzt, wenn das Programm mit zeitaufwendigen Schritten beschäftigt ist und der Nutzer darüber informiert werden soll, dass das Programm deshalb im Moment keine Eingaben entgegen nehmen kann.

Mit ort% lässt sich die Standardausgabeposition verlegen:

```
ort%= 0 REM obere linke Ecke
ort%= 1 REM untere linke Ecke (Standard)
ort%= 2 REM obere rechte Ecke
ort%= 3 REM untere rechte Ecke
```

Eine verzögertes Zuschalten der BUSY-Meldung durch verzoeigerung% (in 1/2-Sekunden-Schritten) wird dann benutzt, wenn die Meldung immer wieder nur kurz am Bildschirm erscheint (abhängig vom dahinter stattfindenden Programmablauf).

Neue BUSY-Meldungen überschreiben die aktuelle, es kann immer nur gerade eine angezeigt werden.

Die maximal verwendbare Zeichenzahl ist 80, möglicherweise werden aber gar nicht alle dargestellt (ausprobieren!).

Siehe auch GIPRINT

**BYREF***F - OPL32*

BYREF variable

Zur Übergabe von Variablen an OPX-Module.

**CACHE***F - SIBO*

```
CACHE init%,max%
CACHE ON
CACHE OFF
```

CACHE initialisiert einen Prozedur-Puffer mit Anfangsgröße init% und maximaler Größe max%. Der Befehl sollte mit TRAP verwendet werden.

CACHE OFF schaltet die Pufferung neu hinzukommender Prozeduren aus, nachdem ein Puffer initialisiert wurde. CACHE ON schaltet die Pufferung wieder an.

Gehört zum Komplex "Pufferung von Prozeduren" im Zusammenhang mit LOADM. Wird unter OPL32 nicht mehr benötigt.

Siehe auch CACHEHDR, CACHEREC, CACHETIDY

**CACHEHDR***F - SIBO*

```
CACHEHDR addr(hdr%)
```

Liest den aktuellen Pufferindex-Kopf in das Array hdr%(), das mindestens 11 Elemente lang sein muss.

Gehört zum Komplex "Pufferung von Prozeduren" im Zusammenhang mit LOADM. Wird unter OPL32 nicht mehr benötigt.

Siehe auch CACHE, CACHEREC, CACHETIDY

**CACHEREC***F - SIBO*

```
CACHEREC add(rec%),off%
```

Liest den Pufferindexeintrag mit Offset off% in rec%(). rec% muss mindestens 18 Elemente umfassen.

Gehört zum Komplex "Pufferung von Prozeduren" im Zusammenhang mit LOADM. Wird unter OPL32 nicht mehr benötigt.

Siehe auch CACHE, CACHEHDR, CACHETIDY

**CACHETIDY***F - SIBO*

CACHETIDY

Entfernt alle nicht aktiven Prozeduren aus den Puffer.

Gehört zum Komplex "Pufferung von Prozeduren" im Zusammenhang mit LOADM. Wird unter OPL32 nicht mehr benötigt.

Siehe auch CACHE, CACHEHDR, CACHEREC



**CALL***F - SIBO*

```
e%=CALL(s%,bx%,cx%,dx%,si%,di%)
```

Mit dieser Funktion können Sie Betriebssystemfunktionen aufrufen. Nur erfahrene Programmierer mit fundierten Betriebssystemkenntnissen sollten diese Funktion verwenden.

Die C-Nummer selbst ist das niederwertige Byte von s%. Der AH-Wert (die Unterfunktionsnummer) ist das höherwertige Byte von s%. Die übrigen Argumente werden an die entsprechenden Register des Prozessors weitergereicht. Der Wert des AX-Registers wird zurückgeliefert.

OPL32 unterstützt Betriebssystemaufrufe über OPX-Module, Anfänger – Finger weg!

**CANCEL***OPL32*

```
CANCEL
```

Beendet den Vorgang des Einfügens (INSERT) oder Veränderens (MODIFY) eines Datensatzes, ohne die zuletzt zugeordneten Daten in die Datenbank zu schreiben.

Siehe auch INSERT, MODIFY, PUT

**CAPTION***OPL32 – SIBO*

```
CAPTION appname$,sprachcode%
```

Benennt den Namen einer Applikation, der unter dem Applikations-Icon in der Extras-Leiste erscheint, bei Bedarf je nach Geräte-Sprache.

CAPTION kann nur innerhalb von APP...ENDA angewendet werden.

sprachcode% spezifiziert, welcher Name für welche Sprachvariante des Gerätes angezeigt wird. So muss das Programm nicht für jede Sprache extra übersetzt werden. Wenn CAPTION verwendet wird, wird der Standard-Name, der im Kopf der APP-Deklaration angegeben ist, überschrieben. Deshalb müssen CAPTION-Angaben zu jeder Sprach-Variante der benutzten Maschinen angegeben werden. Die einzelnen sprachcode%-Werte:

English 1	French 2	German 3	Spanish 4
Italian 5	Swedish 6	Danish 7	Norwegian 8
Finnish 9	American 10	Swiss-French 11	Swiss-German 12
Portuguese 13	Turkish 14	Icelandic 15	Russian 16
Hungarian 17	Dutch 18	Belg.Flemish 19	Australian 20
Belgian-French 21		Austrian 22	New Zealand 23
Int. French 24			

Maximallänge für appname\$ ist 255 Zeichen. Allerdings passen mehr als 8 Zeichen zumeist nicht unter das Icon in der Extras-Leiste ...

**CHANGESPRITE***F - SIBO*

```
CHANGESPRITE x%,time%,bit$(),dx%,dy%
CHANGESPRITE ix%,time%,bit$
```

Ändert die in ix% angegebene Bitmap-Gruppe im aktuellen Sprite (1=erste Gruppe). Die anderen Argumente entsprechen APPENDSPRITE. Unter OPL32 werden Sprites über das eingebaute Sprite-OPX/DLL-Modul verwaltet.

Siehe auch CREATESPRITE, APPENDSPRITE, DRAWSPRITE, POSSPRITE, CLOSESPRITE

**CHR\$***OPL32 – SIBO*

```
zeichen$=CHR$(x%)
```

Gibt als Ergebnis das Zeichen zurück, das zum Wert x% gehört, z.B.

```
zeichen$=CHR$(64) REM Ergebnis: zeichen$ ist "A"
```

Damit werden auch Zeichen zugänglich, die man mit der Tastatur nur schwer oder gar nicht erzeugen kann. Zeichen unterhalb 32 sind Steuerzeichen und zumeist nicht druckbar. Ein Wert von 7 erzeugt bei der Ausgabe mit PRINT einen kurzen Piepston.

**CLEARFLAGS***OPL32*

```
CLEARFLAGS flags&
```

Löscht die Flags, die mit SETFLAGS gesetzt wurden.

Siehe auch SETFLAGS

**CLOSE***OPL32 – SIBO*

```
CLOSE
```

OPL32: Schließt die aktuelle geöffnete Ansicht einer Datenbank-Tabelle. Wenn diese Ansicht die einzige ist, wird auch die Datenbank selbst geschlossen. Mit dem Schließen kann das automatische Komprimieren verbunden werden, wenn in Applikationen SETFLAG entsprechend gesetzt wurde (siehe dort).

SIBO: Schließt die aktuelle Datei. Wurden Datensätze in der Datei mit ERASE gelöscht, wird der belegte Speicher durch CLOSE auf RAM SSDs bzw. im Hauptspeicher wieder freigegeben.

**CLOSESPRITE***F - SIBO*

```
CLOSESPRITE id%
```

Schließt das Sprite mit der Nummer id%.

Siehe auch CREATESPRITE, APPENDSPRITE, CHANGESPRITE, DRAWSPRITE, POSSPRITE

**CLS***OPL32 – SIBO*

```
CLS
```

Löscht den Inhalt des Textfensters, der Text-Cursor wird auf den Ursprung gestellt (AT 1,1).

**CMD\$***OPL32 – SIBO*

```
c$=CMD$(x%)
```

Liefert die beim Programmstart übergebenen Kommandozeilenparameter. Es können auch leere Strings "" zurückgeliefert werden, falls der entsprechende Parameter nicht übergeben wurde.

```
CMD$(2) bis CMD$(5) sind Applikationen vorbehalten.
CMD$(1) liefert den vollen Dateinamen des laufenden
          Programms (mit Pfad).
CMD$(2) liefert den vollen Dateinamen der von der
          Applikation benutzten Datei.
```

CMD\$(3) liefert "C" für "Datei erstellen" bzw. "O" für "Datei öffnen", unter OPL32 auch "R", wenn das Programm vom OPL-Editor oder von der Extras-Leiste aus gestartet wird.  
 CMD\$(4) liefert die "Aliasinformation" soweit vorhanden. Diese hat für Applikationen jedoch keinen praktischen Nutzen.  
 CMD\$(5) liefert den Namen der Applikation (definiert mit APP).

Für OPL32 ist x% von 1 .. 3 sinnvoll, unter SIBO= 1 .. 5.

## **COMMITTRANS**

COMMITTRANS

OPL32

Aktiviert die Übertragung der veränderten Daten der aktuellen Ansicht in die Datenbank.

Siehe auch BEGINTRANS, INTRANS, ROLLBACK

## **COMPACT**

COMPACT Dbname\$

OPL32

Komprimiert die geschlossene Datenbank. Es handelt sich um einen rechenintensiven Prozess, der die Batterien belastet! Die Komprimierung kann automatisch beim Schließen einer DB vorgenommen werden, wenn SETFLAG entsprechend gesetzt ist (siehe dort).

## **COMPRESS**

COMPRESS quelle\$,ziel\$

SIBO

Kopiert eine Datenbankdatei quelle\$ nach ziel\$. Existiert ziel\$, so werden die Datensätze hinten angefügt. Gelöschte Datensätze von quelle\$ werden nicht mitkopiert (wichtig bei der Benutzung von Flash-SSDs).

Um ziel\$ zu überschreiben, müssen Sie vor COMPRESS den Befehl

```
TRAP DELETE ziel$
```

verwenden.

Sie können die bekannten Platzhalter für die Eingabe von Dateinamen verwenden. Wenn Sie in quelle\$ Platzhalter verwenden, darf ziel\$ jedoch nur noch einen Pfad, aber keinen Dateinamen mehr enthalten:

```
COMPRESS "A:*.ODB","B:\BACK\ REM siehe COPY
```

Dieser Befehl wird unter EPOC32 durch COMPACT ersetzt.

**CONTINUE**

OPL32 – SIBO

CONTINUE

Springt sofort zu der Zeile, in der die Abbruch-Bedingung einer DO...UNTIL- bzw. WHILE...ENDWH-Schleife geprüft wird, also zu UNTIL ... bzw. WHILE ...

```
WHILE n%<34
...
IF taste%=27
CONTINUE REM springt zur WHILE-Zeile
ENDIF
...
ENDWH
```

Siehe auch BREAK

**CONST**

OPL32 – SIBO

CONST KKonstantenname=konstantenwert

Deklariert eine Konstante, die direkt als Operand benutzt wird, nicht als Variable. Die Deklaration erfolgt außerhalb aller Prozeduren am Beginn eines OPL-Moduls. Kkonstantenname kann wie üblich mit einer Typspezifikation versehen werden (% , & , \$). CONST-Werte sind global zugänglich und werden nicht von lokalen oder globalen Variablen überschrieben - der Übersetzer verhindert sogar, dass gleichartige Namen verwendet werden. Allgemein sollte der Konstantenname immer mit einem "K" beginnen, um ihn von den Variablen unterscheiden zu können.

Wenn sich die Konstanten im Bereich zu ihren Grenzwerten nicht dezimal deklarieren lassen, verwende man den Hexadezimalwert, z.B. gilt das für \$8000 and &80000000

**COPY**

OPL32 – SIBO

COPY quelle\$,ziel\$

Kopiert die Datei quelle\$, die einen beliebigen Typ haben darf, nach ziel\$. ziel\$ wird ohne Überprüfung überschrieben. Sie müssen die entsprechenden Dateierweiterungen verwenden, um anzuzeigen, welchen Dateityp Sie kopieren wollen. Sie können auch Platzhalter verwenden, um mehrere Dateien auf einmal zu kopieren.

Ein Platzhalter in quelle\$ bedingt, dass in ziel\$ nur noch ein Pfad, jedoch kein Dateiname mehr angegeben werden kann.

Beispiel:

```
COPY "C:\OPL\*", "D:\ME\" REM Backslash nicht vergessen
```

(Siehe COMPRESS, wenn Sie DB-Dateien kopieren wollen)

**COS**

OPL32 – SIBO

y=COS (x)

Gibt den Kosinus-Wert von x zurück (x im Bogenmaß angeben!). Zum Wandeln von Grad in Bogenmaß benutze man die RAD-Funktion.

**COUNT**

OPL32 – SIBO

n%= COUNT

Gibt die Anzahl der Datensätze in der geöffneten aktuellen DB-Ansicht nach n% zurück. Der Befehl löste einen Fehler aus, wenn er zwischen INSERT/MODIFY und PUT (OPL32) angewendet wird.

**CREATE**

OPL32 – SIBO

Unter OPL32:

```
CREATE "meineDB FIELDS Feld1,Feld2,Feld3 ... TO TabName",logName,f1,f2,f3,...
```

bzw.

DBname\$= "meineDB"

```
CREATE DBname$ + " FIELDS Feld1,Feld2,Feld3 ...TO TabName",logName ,f1,f2,f3,...
```

Legt die DB mit dem Namen "meineDB" an und öffnet sie. DBname\$ enthält im Bedarfsfall die volle Dateispezifikation (Laufwerk+Ordner+Dateiname: "C:\Dokumente\mdb1"). In der DB wird die Tabelle mit dem Namen TabName erzeugt. Sie enthält die Felder mit den Bezeichnern Feld1, Feld2, Feld3. Zum Ansprechen der Tabelle wird der logische Name logName verwendet. Der darf aus jeweils einem der Buchstaben von "A" bis "Z" bestehen. Zu den Feldbezeichnern gehören die Variablen f1, f2, f3. Dabei ist zum Feld1 die Variable f1 zugeordnet, f2 zu Feld2, usw. Die Variablen müssen vorher nicht deklariert werden. Den Strings kann eine Längenbegrenzung im Feldbezeichner mitgegeben werden, die Standardlänge ohne Zusatzangabe ist 255

```
CREATE "C:\meineDB FIELDS Name(50),Adresse(100),Nummer TO Adressen",A,n$,a$,n%
```

Datenbanken können mehr als eine Tabelle enthalten. Neue Tabellen erzeugt man unter Nennung der gleichen Datenbankdatei unter Angabe eines anderen Tabellennamens, die Datenbank muss dabei geschlossen sein:

```
CREATE "C:\meineDB FIELDS Index, BLZ, Konto TO Konten",A,i%,blz%,konto%
```

Siehe auch OPEN, OPENR, CLOSE

**Unter SIBO** (funktioniert auch unter OPL32, wird aber von Symbian nicht empfohlen):

```
CREATE file$,log,f1,f2,...
```

Erstellt eine Datenbankdatei mit dem Namen file\$. Als Dateiname kann eine volle Dateispezifikation übergeben werden (128 Zeichen maximal), Feldnamen können bis zu 8 Zeichen lang sein. Die Datei kann bis zu 32 Felder enthalten. Die Felder werden über f1, f2, .... spezifiziert. (In der Standard-Applikation DATEN entspricht f1 der ersten Zeile, f2 der zweiten Zeile...). log ist der logische Dateiname, der als Kürzel für den vollen Dateinamen innerhalb des Programms benutzt wird, um die Datei mit anderen Befehlen, z.B. USE anzusprechen.

Eine Datei, die mit CREATE erstellt wurde, ist gleichzeitig damit auch geöffnet und die aktuelle Datei. Beispiel:

```
CREATE "KUNDEN",B,Nm$,Tel$
```

Erstellt eine Datei Kunden mit dem logischen Namen B und den Feldern Nm\$ und Tel\$.

**CREATESPRITE***F - SIBO*`id%=CREATESPRITE`

Erstellt ein Sprite und liefert die ID-Nummer zurück.

Unter OPL32 werden Sprites über das eingebaute Sprite-OPX/DLL-Modul verwaltet.

Siehe auch APPENDSPRITE, CHANGESPRITE, DRAWSPRITE, POSSPRITE, CLOSESPRITE

**CURSOR***OPL32 – SIBO*

```
CURSOR ON
CURSOR OFF
CURSOR id%,asc%,w%,h%
CURSOR id%,asc%,w%,h%,type%
CURSOR id%
```

`CURSOR ON` schaltet den Textcursor an der aktuellen Bildschirmposition ein. Normalerweise wird kein Cursor angezeigt.

`CURSOR OFF` schaltet jeden Cursor aus.

Sie können mit `CURSOR` auch einen Grafikcursor definieren, der einen vorhandenen Textcursor ersetzt. `id%` gibt die ID-Nummer des Grafikfensters an.

`asc%` gibt an, wieviele Pixel der Cursor über der Grundlinie des aktuellen Fonts liegen soll (-128 bis 127). `h%` und `w%` (0..255) geben die Höhe und Breite des Cursors an.

Werden die Werte nicht angegeben, gelten folgende Voreinstellungen:

```
asc% = entsprechend dem Font
h%   = Fonthöhe
w%   = 2
```

Über `type%` kann optional die Art des Cursors festgelegt werden.

```
1  abgerundet
2  nicht blinkend
4  grau
```

Die Werte können addiert werden. `id%` muss auf ein Fenster verweisen, wird eine Bitmap-ID angegeben, erfolgt eine Fehlermeldung.

**DATETOSECS***OPL32 – SIBO*`s%=DATETOSECS (yr%,mo%,dy%,hr%, mn%,sc%)`

Liefert die Anzahl der Sekunden seit dem 1. Januar 1970, 00:00 zum angegebenen Datum und zur angegebenen Uhrzeit.

Liegt das Datum vor dem 1. Januar 1970, wird ein Fehler ausgegeben. Der zurückgelieferte Wert ist ein long Integer ohne Vorzeichen. Für Daten bis zum 19.1.2038, 03:14:07 (d.h. +2.147.483.647) werden die Werte wie erwartet als positive Zahl ausgegeben. Daten darüber werden als negative Zahl von -2.147.483.648 aufwärts berechnet.

Siehe auch SECSTODATE, HOUR, MINUTE, SECOND

**DATIM\$**

OPL32 – SIBO

d\$=DATIM\$

Liefert Datum und Uhrzeit gemäß Systemuhr als String (z.B.: Fre 16 Okt 1992 16:25:30). Der String hat immer das gezeigte Format, d.h. 3 Zeichen für den Wochentag, Leerzeichen, 2 Ziffern für den Tag, Leerzeichen etc.

**DAY**

OPL32 – SIBO

d%=DAY

Gibt den aktuellen Tag im Monat (1..31) gemäß Systemuhr aus.

**DAYNAME\$**

OPL32 – SIBO

d\$=DAYNAME\$ (x%)

Konvertiert x% (1 bis 7) in den Wochentag, dargestellt als String mit drei Zeichen Länge. d\$=DAYNAME\$ (1) liefert also Mon.

```
PROC Gebtag:
  LOCAL t&,m&,j&,Tag%
  DO
    dINIT
    dTEXT "","Geburtstagsdatum",2
    dTEXT "","z.B. 23 12 1963",$202
    dLONG t&,"Tag",1,31
    dLONG m&,"Monat",1,12
    dLONG j&,"Jahr",1900,2155
    IF DIALOG=0 :BREAK :ENDIF
    Tag%=DOW(t&,m&,j&)
    CLS :PRINT DAYNAME$(Tag%),
    PRINT t&,m&,j&
    dINIT
    dTEXT "","Nochmal?",$202
    dBUTTONS "Nein",%n,"Ja",%j
    UNTIL DIALOG<>%j
  ENDP
```

**DAYS**

OPL32 – SIBO

d&amp;=DAYS(day%,month%,year%)

Liefert zum angegebenen Datum die Anzahl der Tage seit dem 1.1.1990. Mit Hilfe dieser Funktion können Sie einfach die Anzahl der Tage zwischen zwei Daten berechnen. Beispiel:

```
PROC Termin:
  LOCAL a%,b%,c%,Termin&
  LOCAL heute&,noch%
  PRINT "Welcher Tag? (1-31)"
  INPUT a%
  PRINT "Welche Monat? (1-12)"
  INPUT b%
  PRINT "Welches Jahr? (19??)"
  INPUT c%
  Termin&=DAYS(a%,b%,1900+c%)
  heute&=DAYS(DAY,MONTH,YEAR)
  noch%=Termin&-heute&
  PRINT "noch",noch%,"Tage bis zum Termin"
  GET
ENDP
```

Siehe auch dDATE, SECSTODATE

**DAYSTODATE**

OPL32 – SIBO

```
DAYSTODATE days&,year%,month%,day%
```

Konvertiert day&, die Anzahl der Tage seit dem 1.1.1900, in das zugehörige Datum. Die Werte von Jahr, Monat und Tag stehen anschließend in den Variablen year%, month% und day%. Gut geeignet, um Werte aus einer dDATE-Eingabe umzuwandeln, bei der das Dialog-Ergebnis als "Tage seit dem 1.1.1900) anfällt.

**dBUTTONS**

OPL32 – SIBO

```
dBUTTONS text1$,k1%
dBUTTONS text1$,k1%,text2$,k2%
dBUTTONS text1$,k1%,text2$,k2%,text3$,k3%
```

Je nach Angaben in dINIT werden unter einem Dialog oder seitlich rechts davon einer oder mehrere Buttons dargestellt, mit denen ein Dialog beendet werden kann. Je nach Gerät können weitere Buttons nach dem selben Schema angeordnet werden. Im Parameter text\$ steht jeweils der Text, der auf dem Button gezeigt wird, k% ist der Tastenkode, der von DIALOG zurückgeliefert wird. text\$ und k% sind immer paarig anzugeben.

Unter dem Button wird der Tastenname angezeigt, wenn es sich um bezeichnete Tasten handelt (Esc, Enter, Tab ..) oder der Name der Buchstabentaste, ergänzt um "Strg + ". Das ist die Kombination, wenn man den Dialog statt mit dem Stiftes mit der Tastatur bedienen will. Die Zahlen findet man in der Zeichentabelle, den Code der Buchstaben kann man aber auch mit Hilfe des %-Zeichens eingeben: %s steht für den Tastencode von "s".

```
dINIT
  dEDIT name$,"Name:",30
  REM hier weitere Dialogelemente definieren
  dBUTTONS "OK",13,"Suchen",%S,"Abbruch",27
d%= DIALOG
```

In d% wird der Tastaturcode abgelegt, bei Esc allerdings Null statt 27.

Durch Addition bestimmter Zahlen zum Tastaturcode (Flags) erreicht man weitere Effekte (nur OPL32):

256 (\$100) - die Shortcuts unter den Buttons werden nicht dargestellt (spart Platz)

512 (\$200) - für "Buchstabentasten" muss die Strg-Taste nicht mitgedrückt werden

Neben Esc können auch eine oder mehrere andere Tasten als Abbruchtasten definiert werden. Wäre im obigen Beispiel die "Suchen"-Taste eine Abbruchtaste, wird das durch ein Minuszeichen beim Tastaturcode bewirkt (... "Suchen",-%s ...). Wird dieser Button oder die zugehörige Taste gedrückt, wird in d% eine Null zurückgeliefert und alle im Dialog veränderten Eingabe-Werte werden nicht in die zugehörigen Variablen übernommen. Werden Flags verwendet, müssen diese ebenfalls negativ angegeben werden.

dBUTTONS kann innerhalb von dINIT und DIALOG beliebig positioniert werden.

SIBO erlaubt nur 3 Buttons.

Siehe auch dINIT, DIALOG



**dCHECKBOX**

OPL32

`dCHECKBOX stat%,betreff$`

Zeigt innerhalb eines Dialoges eine Checkbox, `betreff$` enthält zugehörigen Text, `stat%` den Status der Checkbox (WAHR (-1) für angehakt, FALSCH (0) für nicht angehakt). Wird der Status geändert, steht er nach DIALOG in `stat%`.

Siehe auch `dINIT`, `DIALOG`

**dCHOICE**

OPL32 – SIBO

```
dCHOICE wahl%,betreff$,elemente$ oder mehrzeilig:
dCHOICE wahl%,betreff$,elemente1$+ "... "
dCHOICE wahl%,"",elemente2$+ "... "
...
dCHOICE wahl%,"",elementeN$
```

Zeigt innerhalb eines Dialoges eine Auswahlliste

Der Inhalt von `betreff$` wird in der Zeile links wiedergegeben. In `elemente$` steht eine kommagetrennte Liste mit den zur Auswahl stehenden Elementen (`elemente$="eins,zwei,drei"`). Wird `wahl%` vorab ein Wert gegeben, wird das Element mit der Nummer `wahl%` angezeigt, bei `wahl%=0` ist es das erste der Liste. Die Auswahl findet über die horizontalen Pfeiltasten oder den Stift statt. Nach Abschluß der Dialoges mit <Enter> steht die Ordnungsnummer des ausgewählten Listenelementes in `wahl%`.

OPL32: Umfasst die Auswahlliste `elemente$` sehr viele Elemente, darf die Zeile nach obiger Vorschrift geteilt werden. Beispiel:

```
dCHOICE wahl%,"Einkaufsliste","Kartoffeln, Butter, ..."
dCHOICE wahl%,"","Zucker, Brot,Karotten, ..."
dCHOICE wahl%,"","Honig,Marmelade,Senf"
```

Siehe auch `dINIT`, `DIALOG`

**dDATE**

OPL32 – SIBO

`dDATE datum&,betreff$,min&,max&`

Es wird eine Eingabemaske für das Datum innerhalb eines Dialoges zur Verfügung gestellt. `betreff$` wird in der Zeile links daneben gezeigt. In `datum&` wird die Eingabe aufgenommen. Enthält die Variable bereits vorher einen Wert, wird dieser in der Eingabemaske angezeigt.

Die Variable wird in der Eingabemaske wie ein Datum angezeigt und auch in dieser Weise editiert, enthält das Datum aber in der Form "Tage seit dem 1.1.1900". Auch `min&` und `max&` rechnen in dieser Weise. Sie geben die erlaubten Eingabegrenzen an.

Beispiel folgt auf der nächsten Seite.

```

PROC DatumsDialog:
  LOCAL d%, min&, max&, datum&, tag%, monat%, jahr%
  max&= DAYS(1,1,2100)
  min&= DAYS(DAY,MONTH,YEAR) REM heute
  dINIT "Datums-Dialog"
  dDATE datum&,"Datum",min&,max&
  IF DIALOG
    REM Rueckgabewerte ausgeben:
    PRINT datum&
    REM lesbare Rueckwandlung der Eingabe
    DAYSTODATE datum&,jahr%,monat%,tag%
    PRINT tag%;".";monat%;".";jahr%
    GET
  ENDIF
ENDP

```

Siehe auch DAYS, DAY,MONTH,YEAR, DAYSTODATE, dINIT, DIALOG

## **DECLARE EXTERNAL**

*F - OPL32*

```
DECLARE EXTERNAL
```

Veranlasst, dass der Übersetzer einen Fehler meldet, wenn irgendeine Variable oder Prozedur benutzt wird, ohne deklariert zu sein. Die Anweisung sollte am Anfang eines Moduls noch vor der ersten Prozedur stehen. Sie erleichtert, die Fehler vom Typ 'Undefined externals' während der Übersetzung zu finden, anstatt erst beim Programmlauf.

Man probiere selbst – einmal mit und einmal ohne DECLARE EXTERNAL:

```

REM DECLARE EXTERNAL
PROC main:
  LOCAL i%
  i%=10
  PRINT i
  GET
ENDP

```

Wenn Sie diese Anweisung verwenden, müssen in der Folge alle anderen Variablen und Prozeduren vorab mit EXTERNAL deklariert werden.

Siehe auch EXTERNAL

## **DECLARE OPX**

*F - OPL32*

```

DECLARE OPX opxname,opxUid&,opxVersion&
...
END DECLARE

```

Deklariert ein OPX-Modul, dessen Name "opxname" ist. opxUid& ist seine UID und opxVersion& seine Versionsnummer.

## **dEDIT**

*OPL32 – SIBO*

```

dEDIT text$,betreff$
dEDIT text$,betreff$,laenge%

```

Definiert innerhalb eines Dialoges eine Texteingabemaske. `betreff$` wird in der Zeile links angezeigt. Steht in der Variablen `text$` bereits vor dem Dialog ein Text, wird er angezeigt. Im Dialog kann der Text verändert oder überhaupt erst eingegeben werden. Das Ergebnis wird nach Dialogende wieder in `text$` abgelegt. Die Länge des Textes ist auf die deklarierte Länge des Strings begrenzt. Reicht das dargestellte Fenster in der Breite nicht aus, wird der Text gescrollt.

Nutzt man den Parameter `laenge%`, kann man die Breite der Eingabemaske in Zeichen angeben.

```
PROC Text:
  LOCAL text$(40)
  dINIT
  dEDIT text$, "Name: ", 20
  DIALOG
ENDP
```

Siehe auch `dINIT`, `DIALOG`

### **dEDITMULTI**

*F - OPL32*

```
dEDITMULTI adr&,betreff&,breite%,zeilen%,anz%
```

In einem Dialog: Alternative zu `dEDIT`, wenn der einzugebende Text länger als die Maximallänge eines Strings ist (255 Zeichen). Dazu ist vorab ein zusammenhängender reservierter Pufferspeicher-Bereich festzulegen. Es gibt mehrere Methoden, bewährt hat sich die Festlegung eines Long-Integer-Arrays, das nicht anderweitig benutzt werden darf.

Die Anzahl der benötigten Felder ergibt sich aus der Formel:

Feldanzahl =  $\text{INT}(1 + (\text{maxZeichenanzahl} + 3) / 4)$ .

Für 256 Zeichen wäre ein Puffer dann so zu deklarieren:

```
LOCAL puffer&(65)
```

Der Pufferbereich beginnt ab der Adresse:

```
adr&=ADDR(puffer&(1))
```

`dEDITMULTI` benutzt den Bereich der ersten vier Bytes des geschaffenen Pufferbereiches zur Rückgabe der Anzahl der eingegebenen Zeichen - das entspricht also genau dem Inhalt von `puffer&(1)`. Ab dem ersten Byte von `puffer&(2)` (also ab `ADDR(puffer&(2))` oder auch identisch `4+ADDR(puffer&(1))`) liegen byteweise die eingegebenen Zeichen als Tastaturwerte.

Der reservierte Bereich muss zur Weiterverwendung der eingegebenen Daten mit der `PEEKB`-Funktion ausgelesen werden. Direkter Zugriff ist nicht möglich.

`betreff$` wird am linken oberen Rand des Editierfeldes angezeigt, dessen Breite in Zeichen und die angezeigte Zeilenanzahl durch `breite%` und `zeilen%` festgelegt. Reicht die Zeilenanzahl nicht, um den Text vollständig anzuzeigen, wird er gescrollt.

Die Enter-Taste ist wie in einem Textprogramm für den Textumbruch vorgesehen, der Dialog muss also anders beendet werden. Dazu wird eine Tastenkombination per `dBUTTONS` aktiviert. `<Esc>` bewirkt wie immer einen Abbruch dieses Dialoges ohne Übernahme der Werte in den Pufferbereich.

Beispiel (nächste Seite):

```

CONST anz%= 500      REM Puffergroesse/Anzahl der Zeichen
PROC Editmulti:
  LOCAL puffer&(126)
  LOCAL adr&         REM Adresse Pufferbeginn
  LOCAL n%           REM Zaehlvariable
  LOCAL z%           REM Zeichenwert
  LOCAL breite%      REM Editfensterbreite in Zeichen
  LOCAL zeilen%      REM Anzahl der sichtbaren Zeilen
  breite%= 20
  zeilen%= 4
  adr&=ADDR(puffer&(1))
  dINIT "EditMulti Demo"
  dEDITMULTI adr&,"Text:",breite%,zeilen%,anz%
  dBUTTONS "Fertig",%f
  IF DIALOG
    PRINT "Zeichenanzahl: ";puffer&(1)
    PRINT "Text:",
    n%=0
    WHILE n%<puffer&(1)
      z%=PEEKB(adr&+4+n%)
      IF z%>=32
        PRINT CHR$(z%);
      ELSE
        PRINT "."; REM nicht darstellb. Sonderzeichen
      ENDIF
      n%=n%+1
    ENDWH
  ENDIF
  GET
ENDP

```

In dem Editierfeld können Texte mit Hilfe von "Kopieren & Einfügen" bearbeitet werden.

Das Beispiel versagt, wenn mehr als 64KB Text verarbeitet werden sollen. Man muss dann die Funktion ALLOC zur Speicherreservierung verwenden. ALLOC ist eine von mehreren Funktionen zur dynamischen Speicherverwaltung.

Siehe auch dINIT, DIALOG, dBUTTONS, PEEKB, ADDR

## DEFAULTWIN

DEFAULTWIN modus%

OPL32 - SIBO

### OPL32:

Ändert den Farbanzeigemodus des Basis-Fensters. Die Grundeinstellung ist: 4-Farb-Modus für Geräte mit Grau-Bildschirm und 256 für Farb-Geräte. DEFAULTWIN ohne Parameter löscht lediglich den Bildschirm. Mit Parameter wird jeweils der Bildschirm gelöscht und ein anderer Modus gesetzt. Der kann sich aber nur auswirken, wenn es das Gerät zulässt.

```

REM für den neuen Farbmodus:
modus%= 0 setzt  2-Graustufen-Modus
modus%= 1 setzt  4-Graustufen-Modus
modus%= 2 setzt  16-Graustufen-Modus
modus%= 3 setzt  256-Graustufen-Modus
modus%= 4 setzt  16-Farb-Modus
modus%= 5 setzt  256-Farb-Modus

```

Der Modus mit den meisten Farben/Graustufen belastet zugleich die Batterie am meisten.

**SIBO:**

Verändert das Standardfenster (ID=1), um die Benutzung der grauen Ebene zu ermöglichen (ist ursprünglich nicht möglich).

mode%=1 schaltet die graue Ebene ein mode%=0 schaltet sie wieder aus.

Das Standardfenster wird bei jedem Aufruf von DEFAULTWIN gelöscht. Beachten Sie bitte, dass die graue Ebene zusätzlichen Speicher verbraucht.

Sie sollten DEFAULTWIN nur am Anfang des Programms verwenden, wenn Sie die graue Ebene benötigen, da das Programm dann im Falle von fehlendem Hauptspeicher sauber verlassen werden kann, ohne wichtige Daten zu verlieren.

Siehe auch: gGREY, gCOLOR, gCOLORINFO, gCREATE

**DEG**

OPL32 – SIBO

y=DEG(x)

Wandelt x (im Bogenmaß) in Grad um.

Siehe auch RAD

**DELETE**

OPL32

DELETE dbname\$,tabelle\$

Löscht eine Tabelle (tabelle\$) aus der Datenbank(dbname\$). Das funktioniert nur, wenn alle Ansichten und die Datenbank selber geschlossen sind.

**DELETE**

OPL32 – SIBO

DELETE datein\$

Löscht die Datei mit Namen datein\$ unabhängig vom Typ der Datei.

Sie können Platzhalter verwenden (z.B.: DELETE "D:\OPL\\*.OPL" um alle OPL-Dateien zu löschen).

Siehe auch RMDIR

**dFILE**

OPL32

dFILE dateiname\$,betreff\$,format%

dFILE dateiname\$,betreff\$,format%,uid1&,uid2&,uid3&

Definiert ein Eingabefeld zur Eingabe von Dateinamen für einen Dialog. Zur Wahl des Laufwerks und eines Verzeichnisses werden zusätzliche Auswahlfelder unter dem Eingabefeld eingefügt.

Standardmäßig werden unter OPL32 keine Bezeichnungen an den Eingabefeldern gezeigt. Mit betreff\$ jedoch werden solche Bezeichnungen am linken Dialogrand vor dem Eingabefeld angezeigt. betreff\$ sollte eine kommagetrennte Liste nach diesem Beispiel enthalten:

```
dFILE dateiname$,"Datei, Ordner, Laufwerk",format%
```

format% gibt den Typ des Eingabefelds und die erlaubten Eingaben an, Sie können die folgenden Optionen durch Addition kombinieren; enthält die Rückgabefeldvariable dateiname\$ (Deklaration auf 255 Zeichen!) bereits einen Wert, so wird er je nach format% ignoriert oder angezeigt.

Der Parameter format% steuert die Nutzungsart:

format%	Wirkung
0	bietet Liste zur Dateiauswahl an
1	Dateiname selbst eingegeben, keine Auswahl
2	Eingabe von Ordnernamen ist erlaubt
4	Nur Ordnernamen erlaubt
8	Nimmt existierende Dateinamen nicht an
16	ragt bei bereits existierendem Dateinamen um Überschreiberlaubnis nach
32	Gestattet leeres Datei-Eingabefeld
128	Benutzung von Wildcards erlaubt
256	Dateien im ROM können gewählt werden
512	Dateien im Systemordner können gewählt werden

Einige format%-Werte machen allein keinen Sinn oder werden gar mit einer Fehlermeldung quittiert.

Anmerkung zu format%=2: Zusammen mit format%=1 (Eingabe statt Auswahl des Dateinamens): Der Ordner kann im Dateinamen mit angegeben werden. In diesem Falle wird der Wert in der Ordner-Auswahlbox negiert. Gibt man keinen Ordner an, wird der Wert aus der Ordner-Auswahlbox übernommen.

Anmerkung zu format%=4: Nur sinnvoll zusammen mit format%=1 (Eingabe statt Auswahl des Dateinamens): Die Eingabe in das Dateifeld benennt einen Ordner, die Auswahl aus der Ordner-Box wird mit berücksichtigt. Der "neue Ordner" entspricht also einem Unterordner. Sinnvoll im Zusammenhang mit MKDIR.

Für Datei-Auswahlfelder (format%=0) wird eine automatische Begrenzung auf bestimmte Dateitypen mit Hilfe der UIDs unterstützt. Man kann z.B. so erreichen, dass nur Dateien der eigenen oder ganz spezieller Applikationen in der Liste angeboten werden. OPL-bezogene UIDs findet man in der Datei, die die Beschreibung der Konstanten enthält (CONST.OPH), zu verwenden als uid2&:

```
CONST KUidOplApp&=268435572    OPL-Applikation
CONST KUidOPO&=268435571      OPO-Datei
CONST KUidOplFile&=268435594  z.B. Datenbankdatei
```

Die Angabe von uid3& ist sinnvoll für Dokumente eigener Applikationen. Diese findet man in der Zeile, in der die Applikation deklariert wird.

Das folgende Beispiel lässt nur OPO-Dateien und Dateien einer eigenen Applikation zur Auswahl zu:

Demo-Nummer der eigenen Applikation = 47114711

```
dFILE dateiname$,betreff$,format%,0, 268435571, 47114711
```

Siehe auch dINIT, DIALOG

## **dFILE**

*SIBO*

```
dFILE str$,p$,f%
```

Definiert ein Eingabefeld zur Eingabe von Dateinamen für einen Dialog. Eine Auswahlfeld zur Wahl des Laufwerks wird automatisch in die Zeile unter dem Eingabefeld eingefügt.

p\$ wird am linken Dialogrand, vor dem Eingabefeld angezeigt.

f% gibt den Typ des Eingabefelds und die erlaubten Eingaben an, Sie können folgende Optionen durch Addition kombinieren.

Wert	Bedeutung
1	String-Eingabefeld verwenden
2	Verzeichnisnamen erlauben
4	nur Verzeichnisnamen
8	nur neue Dateinamen

(Fortsetzung)

Wert	Bedeutung
16	Sicherheitsabfrage bei bestehenden Dateinamen
32	leere Strings als Eingabe akzeptieren
64	Dateierweiterung nicht anzeigen
128	Platzhalter zulassen

Der erste mögliche Wert ist der wichtigste. Wenn Sie 1 zu f% addieren, können Sie neue Dateinamen in einem String-Eingabefeld editieren, andernfalls haben Sie nur die Möglichkeit, zwischen bestehenden, passenden Dateinamen zu wählen.

Um einen Dialog zum Kopieren von Dateien zu definieren, könnten Sie z.B. 1,2 und 16 addieren, um ein Eingabefeld zu ermöglichen, in das Verzeichnisse eingegeben werden können und das bei Eingabe eines bestehenden Dateinamens über eine Sicherheitsabfrage warnt.

Um ein Verzeichnis zu entfernen, sollten Sie den Wert 4 verwenden, um nur bestehende Verzeichnisse anzuzeigen. Die Option "Sicherheitsabfrage bei existierenden Dateinamen" wird ignoriert wenn zugleich "nur neue Dateinamen" angewählt wurde.

str\$ ist die zu editierende String-Variable. Die voreingestellten Werte definieren die ursprünglich benutzen Dateinamen bzw. das ursprüngliche Verzeichnis. Bei einem String-Eingabefeld wird jeder angegebene Teil eines Dateinamens angezeigt, bei einem Auswahlfeld für existierende Dateien werden die voreingestellten Werte benutzt um die anwählbaren Dateien zu spezifizieren. Sie könnten also z.B. \*.tmp angeben um alle Temporärdateien anzeigen zu lassen (Zur Verwendung des Platzhalters \* muss 128 zu f% addiert werden).

Enthält str\$ keine Laufwerks- oder Verzeichnisangaben, wird das mit SETPATH gesetzte Standardverzeichnis des Programms verwendet oder, wenn auch SETPATH nicht verwendet wurde, das Verzeichnis \OPD des Standardlaufwerks.

Wenn Sie ein Dateiauswahlfeld definiert haben (1 nicht addiert) bewirkt die Option 8, dass nur zu str\$ passende Dateien angewählt werden können.

Sie können immer mit TAB das große Dateiauswahlfenster für das Dateieingabefeld öffnen.

str\$ muss mindestens 128 Zeichen lang sein, sonst wird beim Aufruf ein Fehler ausgegeben.

Siehe auch dINIT, DIALOG

## **dFLOAT**

OPL32 – SIBO

dFLOAT fliessk,betreff\$,min,max

Zeigt innerhalb eines Dialoges eine Eingabemaske für Fließkommazahlen, deren Wertebereich durch min und max eingeschränkt wird. betreff\$ wird am linken Rand der Zeile angezeigt. Ist fliessk bereits mit einem Wert belegt, wird er ebenfalls angezeigt. Diese Variable nimmt nach dem Dialog die Eingabe auf.

Siehe auch dINIT, DIALOG

## **DIALOG**

OPL32 – SIBO

d%=DIALOG

Zeigt den mit den Befehlen dINIT, dEDIT, dDATE, usw. definierten Dialog an. Wird der Dialog mit ENTER verlassen, werden die Eingabewerte in den zugeordneten Werten gespeichert. Mit Hilfe von dBUTTONS können Sie dies verhindern.

Wenn Sie dBUTTONS zur Definition von speziellen Tasten zum Verlassen des Dialogs verwendet haben, wird der Code der Taste zurückgeliefert, über die der Dialog beendet wurde. Andernfalls wird die Nummer des aktuellen Felds beim Verlassen des Dialogs ausgegeben (1 steht für das erste Feld oder eine evtl. vorhandene Titelzeile).

Mit ESC wird der Dialog abgebrochen, ohne die Feldinhalte zu verändern. `DIALOG` liefert in diesem Fall 0 zurück.

Siehe auch `dINIT`

### **DIAMINIT**

*SIBO*

`DIAMINIT pos%,str1%,str2%...`

Initialisiert die ♦-Liste (eine bestehende Liste wird überschrieben). `str1$`, `str2$`... enthalten den Text, der im Statusfenster als Listeneintrag für das entsprechende Element angezeigt werden soll.

`pos%` zeigt auf das Element, das ursprünglich selektiert sein soll. (1 steht für `str1$`...). Für `pos%>=1` müssen Sie mindestens `pos%` Strings angeben. Wenn Sie `pos%` größer als die Anzahl der Strings angeben, wird der letzte String in der Liste angewählt.

Wenn `pos%` nicht angegeben wird oder `pos%=0` ist oder `DIAMINIT` alleine ohne Argumente aufgerufen, wird kein Balken angezeigt.

Bei `pos%=-1` wird die Liste durch das Programmsymbol im großen Statusfenster ersetzt.

OPL32 verwendet anstelle des Statusfensters den Toolbar!

### **DIAMPOS**

*SIBO*

`DIAMPOS pos%`

Positioniert den ♦-Zeiger in der ♦-Liste.

Bei Positionierung über die Anzahl der Elemente in der Liste hinaus wird auf die Position "`pos%` modulo Anzahl" positioniert. `pos%=0` entfernt den ♦-Zeiger.

### **dINIT**

*OPL32*

`dINIT`  
`dINIT titel$`  
`dINIT titel$,flag%`

Kennzeichnet den Anfang der Definition einer Dialogbox. Es folgen Dialog-Elemente wie `dTEXT` usw. Um den Dialog schließlich anzuzeigen, wird die Definition mit `DIALOG` abgeschlossen.

Wenn in `titel$` ein Wert übergeben wird, wird er im grauen Feld der Dialogbox ganz oben angezeigt.

Mit `flag%` kann die Ansicht beeinflusst werden:

```
flag%= 1 Buttons werden statt unten horizontal rechts
        und vertikal dargestellt
flag%= 2 der Titel wird nicht angezeigt, Box kann nicht
        verschoben werden
flag%= 4 der gesamte Bildschirm wird benutzt
flag%= 8 die Dialogbox kann nicht per Stift
        verschoben werden
flag%= 16 platzsparende Anzeige, es passt mehr auf den
        Bildschirm
```

Siehe auch `DIALOG`



## SIBO

**dINIT**

```
dINIT titel$
dINIT
```

Bereitet die Definition eines neuen Dialogs vor. Jede bestehende Dialogdefinition wird dadurch verworfen. Mit den Befehlen dEDIT... können Sie nach Aufruf von dINIT den Dialog definieren, den Sie dann mit DIALOG aufrufen.

titel\$ definiert eine optionale Titelzeile, die zentriert und durch eine horizontale Linie vom restlichen Dialog getrennt am oberen Rand des Dialogs angezeigt wird.

Siehe auch DIALOG

## OPL32 – SIBO

**DIR\$**

```
d$=DIR$(filespez$) REM und danach
d$=DIR$ (" ")
```

Gibt alle Dateinamen bzw. Verzeichnisse aus, die mit filespez\$ übereinstimmen. Sie können wie gewohnt Platzhalter zur Angabe der Dateispezifikation verwenden. Wenn Sie nur ein Verzeichnis an filespez\$ übergeben, müssen Sie unbedingt "\*" an das Ende der Spezifikation stellen ("D:\TEMP\"). DIR\$ wird wie folgt benutzt:

DIR\$(filespez\$) liefert den ersten passenden Dateinamen.

DIR\$ (" ") liefert dann bei jedem weiteren Aufruf den nächsten passenden Dateinamen. Wird keine passende Datei mehr gefunden liefert DIR\$ den Wert "" zurück.

Beispiel:

```
PROC dir:
  LOCAL d$(255)      REM bei SIBO reicht: LOCAL d$(128)
  d$=DIR$ ("D:\DAT\*.DBF")
  WHILE d$<>" "
    PRINT d$
    d$=DIR$ (" ")
  ENDWH
  GET
ENDP
```

Das Beispielprogramm zeigt alle .DBF-Dateien im Verzeichnis "D:\DAT\" an.

## OPL32 – SIBO

**dLONG**

```
dLONG longint&,betreff$,min&,max&
```

Eingabebox für eine (Long-)Integerzahl innerhalb eines Dialoges. Der eingegebene Wert landet in longint&, min& und max& sind die Begrenzungen des Eingabe-Wertebereiches. Der Parameter betreff\$ wird links der Eingabebox angezeigt. Benötigt man diese Anzeige nicht, ist ein Leerstring anzugeben. Hat longint& einen Wert, wird dieser beim Aufblenden des Dialoges angezeigt.

dLONG ist die einzige Art, Integers einzugeben, es gibt keine eigenständige Eingabemöglichkeit für normale (kurze) Integer-Zahlen. Wer nur den Wertebereich der normalen Integers ausschöpfen möchte, begrenze die Eingabe von vornherein mit den entsprechenden min&- und max&-Werten.

Siehe auch dINIT, DIALOG

**DO...UNTIL***OPL32 – SIBO*

```
DO
... Anweisungsfolge ...
UNTIL abbruchbedingung
```

DO startet eine Schleife, in der eine Anweisungsfolge immer wieder abgearbeitet wird, bis die Abbruchbedingung zutrifft. Die Abbruchbedingung wird bei UNTIL formuliert, UNTIL ist gleichzeitig das Schleifen-Ende. Wird die Abbruchbedingung nie erfüllt, läuft die Schleife unendlich. In OPO-Programmen kann sie dann nur mit der Tastenkombination <Strg><Esc> abgebrochen werden, in Applikationen oder wenn die Anweisung ESCAPE OFF gesetzt wurde, ist die Beendigung nur über "Schließen" in der Liste der geöffneten Programme oder einen Softreset möglich.

**DOW***OPL32 – SIBO*

```
d%=DOW(tag%,monat%,jahr%)
```

Gibt den Wochentag zum angegebenen Datum als Zahlenwert von 1 (=Montag) bis 7 (=Sonntag) aus. Tag% muss zwischen 1 und 31, monat% zwischen 1 und 12 und jahr% zwischen 1900 und 2155 liegen.

D%=DOW(4,7,1992) liefert z.B. 6 (also Samstag).

**dPOSITION***OPL32 – SIBO*

```
dPOSITION x%,y%
```

Positioniert eine Dialogbox auf dem Bildschirm (ohne diese Angabe wird die Box zentriert dargestellt). Der Befehl kann irgendwo zwischen dINIT und DIALOG angeordnet sein. Mit x%,y% werden horizontale und vertikale Position festgelegt. Die Positionierung erfolgt nur grob gegliedert:

```
x%= -1 links
x%=  0 Mitte
x%=  1 rechts
y%= -1 oben
y%=  0 Mitte
y%=  1 unten
```

Siehe auch dINIT

**DRAWSPRITE***SIBO*

```
DRAWSPRITE x%,y%
```

Zeichnet das aktuelle Sprite im aktuellen Fenster, wobei die linke obere Ecke des Sprites bei Position x%,y% (Pixel) liegt.

Unter OPL32 werden Sprites über das eingebaute Sprite-OPX/DLL-Modul verwaltet.

Siehe auch CREATESPRITE, APPENDSPRITE, CHANGESPRITE, POSSPRITE, CLOSESPRITE

**dTEXT***OPL32*

```
dTEXT betreff$,text$
dTEXT betreff$,text$,format%
```

Zeigt einen Text innerhalb eines Dialoges. Dabei wird betreff\$ links neben dem eigentlichen Text text\$ dargestellt. Wird betreff\$ nicht benötigt, ist ein Leerstring zu verwenden, andererseits darf text\$ kein Leerstring sein

(Ausnahme s.u.)! `text$` wird immer linksbündig dargestellt, mit `format%` kann das beeinflusst werden, wenn `betreff$` ein Leerstring ist:

<code>format%</code>	Wirkung
0	linksbündig
1	rechtsbündig
2	zentriert

Zusätzlich können jeweils diese Werte zu `format%` addiert werden, um weitere Effekte zu erreichen:

<code>format%</code>	Wirkung
\$200	Linie unter dieses Dialogelement ziehen
\$400	dieses Dialogelement darf ausgewählt werden ( <code>betreff\$</code> antippen)
\$800	Textseparator (nur wenn <code>betreff\$</code> und <code>text\$</code> Leerstrings sind!)

Der Separator zählt als Eintrag in der Liste der Dialogelemente. Darauf ist zu achten, wenn der Dialog mit Enter beendet wird.

Siehe auch `dINIT`, `DIALOG`

## ***dTEXT***

*SIBO*

```
dTEXT betreff$,text$
dTEXT betreff$,text$,format%
```

Definiert eine Textzeile, die in einem Dialog angezeigt wird.

`betreff$` wird am linken Dialogrand vor dem in `text$` definierten Text angezeigt. Um nur einen String anzuzeigen, initialisieren Sie `betreff$` einfach mit `""`. Die gesamte Zeile steht dann für `text$` zur Verfügung. Ist `text$=""` wird ein Fehler ausgegeben. `text$` wird normalerweise linksbündig neben `p$` angezeigt. Sie können diese Einstellung jedoch mit `format%` verändern:

<code>format%</code>	Wirkung
0	linksbündig
1	rechtsbündig
2	zentriert

Zusätzlich können Sie noch folgende Optionen durch Addition zu `format%` hinzufügen:

<code>format%</code>	Wirkung
\$100	Fettschrift verwenden
\$200	Linie unter diesem Element ziehen
\$400	Element selektierbar machen.

Je Dialog kann nur eine Linie gezeichnet werden. Jeweils die letzte in einem Dialog definierte Linie wird gezeichnet, unabhängig, ob von der Titeldefinition des Dialogs oder von `dTEXT` stammend.

Siehe auch `dEDIT`

## OPL32 – SIBO

**dTIME**

```
dTIME zeit&,betreff$,format%,min&,max&
```

Zeigt eine Eingabemaske zum Editieren einer Zeit innerhalb eines Dialogs.

Dabei wird `betreff$` in der Zeile links neben der Maske präsentiert. In der Maske wird der bereits in `zeit&` befindliche Wert als normale Uhrzeit angezeigt. Diese Variable muss einen Sekundenwert enthalten. Nach dem Dialog wird die eingegebene Uhrzeit wieder als Sekundenwert abgespeichert. Handelt es sich um eine Uhrzeit, nicht um einen Zeitwert, rechnen die Sekunden ab Mitternacht 00:00.

In `min&` und `max&` stehen Start- und möglicher Endwert der Eingabe.

Die Anzeige des Zeit-Dialogs lässt sich mit `format%` beeinflussen:

format%	Anzeige
0	abs. Zeit ohne Sekunden
1	abs. Zeit mit Sekunden
2	Zeitdauer ohne Sekunden
3	Zeitdauer mit Sekunden
4	abs. Zeit in Minuten innerhalb der Stunde (nur OPL32)
8	24h-Uhr, nur interessant bei nicht-deutschen Systemen (nur OPL32)

Siehe auch `dINIT`, `DIALOG`

## OPL32 – SIBO

**dXINPUT**

```
dXINPUT text$,betreff$
```

Ermöglicht innerhalb eines Dialoges die Eingabe von verdecktem Text, z.B. für Passwörter. Anstelle der Buchstaben wird ein "\*" wiedergegeben. `betreff$` wird in der Zeile links angezeigt.

Die Eingabe wird in `text$` zurückgeliefert. Die Stringlänge ist auf 16 Zeichen begrenzt (OPL32) bzw. muss mindestens 8 Zeichen betragen (SIBO); die Deklaration muss dementsprechend erfolgen. Das Dialogfeld enthält ursprünglich keine Zeichen. Bei diesem Dialogelement wird ein in `text$` bereits vorhandener Wert nicht angezeigt.

Siehe auch `dINIT`, `DIALOG`

## OPL32 – SIBO

**EDIT**

```
EDIT string$
```

Die Stringvariable `string$` wird im Textfenster direkt am Bildschirm editiert. Dabei sind alle Editier-Tasten verwendbar (Esc, Pfeiltasten, Entf, Tab ..). Der Editiervorgang wird durch <Enter> beendet und die Veränderungen erst dann nach `string$` übernommen. <Enter> ohne Veränderung des Strings erhält den Wert, der schon vorher in `string$` stand, das kann auch ein Leerstring sein.

Eine Besonderheit ergibt die Kombination mit `PRINT`:

```
PRINT "Eingabe Name:",
EDIT name$
```

Die Eingabe erfolgt dann auf der gleichen Zeile wie die `PRINT`-Ausgabe.

<Esc> löscht normalerweise nur die Eingabezeile. Durch eine besondere Konstruktion kann `EDIT` auch mit <Esc> verlassen werden. Der `EDIT`-Befehl muss dazu "getrappt" sein:

TRAP EDIT string\$

Ist die Eingabezeile leer und <Esc> wird gedrückt, wird EDIT beendet. Die Variable string\$ behält ihren alten Wert. In ERR steht die Fehlernummer -114.

Siehe auch INPUT, dEDIT

## **ELSE/ELSEIF/ENDIF**

OPL32 – SIBO

Siehe IF

## **EOF**

OPL32 – SIBO

ende%= EOF

Überprüft auf Dateiende.

Liefert -1 (WAHR), wenn das Dateiende erreicht wurde, oder 0 (FALSCH), wenn nicht.

Sie sollten EOF immer beim sequentiellen Dateizugriff verwenden, um Fehlermeldungen zu vermeiden, wenn über das Dateiende hinaus gelesen wird.

Beispiel:

```
PROC eoftest:
  OPEN "xydatei",A,a$,b%
  DO
    PRINT A.a$
    PRINT A.b%
  NEXT
  PAUSE -40
  UNTIL EOF
  PRINT "Dies ist der letzte Datensatz"
  GET
  RETURN
ENDP
```

## **ENDWH**

OPL32 – SIBO

Siehe WHILE

## **ENDV**

OPL32 – SIBO

Siehe VECTOR

## **ENTERSEND**

SIBO

ret%=ENTERSEND(zobj%,m%,p1,...)

Arbeitet wie SEND, mit dem Unterschied, dass bei Abbruch der Methode der Fehlercode an die aufrufende Prozedur zurückgeliefert wird. Sonst wird der von der Methode zurückgelieferte Wert zurückgegeben.

## **ENTERSEND0**

SIBO

ret%=ENTERSEND0(zobj%,m%,p1%)

Wie ENTERSEND, mit dem Unterschied, dass 0 zurückgeliefert wird, wenn die Methode nicht abbricht.

**ERASE***OPL32 – SIBO*

ERASE

Der Datensatz wird gelöscht und der nächst folgende zum aktuellen. Steht der zu löschende Datensatz in der Position `LAST`, wird der Datensatz nur geleert, `COUNT` aber um eins vermindert und `EOF` auf `WAHR` (-1) gesetzt, der leere Datensatz bleibt der aktuelle.

**ERR***OPL32 – SIBO*

fehler%=ERR

Liefert bei einem Fehler die Fehlernummer nach fehler% zurück oder, wenn es keinen Fehler gab, eine Null.

Die Funktion ist nur verwendbar, wenn der Programmierer eine Fehlerbehandlung vorgesehen hat (`TRAP`, `ONERR`, ...), ansonsten greift die System-Fehlerroutine und das Programm wird abgebrochen. Siehe Kapitel "Fehler".

ERR behält seinen Wert bis zur nächsten Fehlerbehandlung. Wenn der Wert zurückgesetzt werden soll, wende man:

```
TRAP RAISE 0
```

**ERR\$***OPL32 – SIBO*

fehler\$=ERR\$(fehler%)

ERR\$(fehler%) enthält die zur Fehlernummer fehler% gehörige Fehlerbeschreibung. Für den letzten Fehler ist das:

```
fehler$=ERR$(ERR)
```

Die Funktion ist nur verwendbar, wenn der Programmierer eine Fehlerbehandlung vorgesehen hat (`TRAP`, `ONERR`, ...), ansonsten greift die System-Fehlerroutine und das Programm wird abgebrochen. Siehe Kapitel "Fehler".

**ERRX\$***OPL32 – SIBO*

fehler\$=ERRX\$

ERRX\$ enthält die erweiterte Fehlermeldung, die Auskunft gibt, wo der Fehler aufgetreten ist (Modul, Prozedur, ...)

Die Funktion ist nur verwendbar, wenn der Programmierer eine Fehlerbehandlung vorgesehen hat (`TRAP`, `ONERR`, ...), ansonsten greift die System-Fehlerroutine und das Programm wird abgebrochen. Siehe Kapitel "Fehler".

**ESCAPE OFF, ESC ON***OPL32 – SIBO*

ESCAPE OFF

...

ESCAPE ON

ESCAPE OFF verhindert das Funktionieren von `<Strg><Esc>` (OPL32) bzw. `<PSION><Esc>` (SIBO) zum Abbruch eines laufenden Programmes. Mit `ESCAPE ON` wird die Funktion wieder aktiviert.

Bleibt das Programm unter diesen Umständen in einer Endlosschleife hängen, muss es über die Taskliste mit "Datei schließen" (OPL32) bzw. die Systemoberfläche (SIBO) beendet werden.

**EVAL***OPL32 – SIBO*

```
wert=EVAL(string$)
```

Wenn string\$ einen korrekten mathematischen Ausdruck enthält, errechnet EVAL daraus die Fließkommazahl. Der Ausdruck kann auch Prozeduraufrufe enthalten, die ein Rechenergebnis liefern.

OPL32: Nicht verwendet werden können lokale Variablen!

```
PROC Main:
  PRINT EVAL ("22*3/Rechne:")
  GET
ENDP

PROC Rechne:
  RETURN 5
ENDP
```

Siehe auch: VAL

**EXIST***OPL32 – SIBO*

```
e%=EXIST(datei$)
```

Überprüft ob die Datei datei\$ existiert. Liefert -1 (WAHR), wenn die Datei existiert, und 0 (FALSCH), wenn nicht. Die Funktion wird benutzt, um beim Erstellen oder Öffnen von Dateien auf Existenz der Datei zu prüfen.

Beispiel:

```
IF NOT EXIST("Kunden")
  CREATE "Kunden",A,name$
ELSE
  OPEN "Kunden",A,name$
ENDIF
...
```

**EXP***OPL32 – SIBO*

```
y=EXP(x)
```

Liefert  $e^x$ , die x-Potenz der Eulerschen Zahl (2.71828...)

**EXT***SIBO*

```
EXT name$
```

Bestimmt die Dateierweiterung von Dateien, die einer Applikation zugeordnet sind. Der Befehl kann nur zwischen APP und ENDA benutzt werden.

**EXTERNAL**

EXTERNAL variable oder  
EXTERNAL prototype

Erforderlich, wenn im Modul DECLARE EXTERNAL verwendet worden ist.

Beispiel 1 deklariert eine Variable zur externen, z.B.

```
EXTERNAL screenHeight%
```

Beispiel 2 deklariert den Prototyp einer Prozedur. Dadurch kann sich eine Prozedur auf den Prototyp beziehen, ohne dass dieser bereits existieren muss. Damit wird der Typ-Check von Parametern sowie die notwendige Anpassung von numerischen Argumenten während der Übersetzung möglich, anstatt dass ein Fehler erst zur Laufzeit bemerkt wird.

```
DECLARE EXTERNAL
EXTERNAL myProc%:(i%,l&)
REM oder INCLUDE "myproc.oph", wo alle Prozeduren
REM definiert sind

PROC test:
  LOCAL i%,j%,s$(10)
  REM j% wird intern in eine Long-Integer Zahl gewan-
  REM delt, wie das im Prototyp spezifiziert ist:
  myProc%:(i%,j%)
  REM Übersetzer-Fehler 'Type mismatch':
  REM ein String kann nicht in einen numerischen
  REM Typ gewandelt werden:
  myProc%:(i%,s$)
  REM falsche Argument-Anzahl ergibt Übersetzer-Fehler:
  myProc%:(i%)
ENDP

PROC myProc%:(i%,l&)
  REM Übersetzer prüft die Konsistenz mit d.Prototyp
  ...
ENDP
```

Siehe auch DECLARE EXTERNAL

**FIND**

```
n%= FIND(string$)
```

Durchsucht die aktuelle Datei (oder "Ansicht" in OPL32) nach Feldern, die string\$ entsprechen. Die Suche beginnt beim aktuellen Datensatz. Um weitere Datensätze zu finden, nachdem FIND bereits erfolgreich war, müssen Sie vorher NEXT verwenden. FIND macht den gefundenen Datensatz zum aktuellen Datensatz und liefert die Datensatznummer zurück. Groß-/Kleinschreibung wird ignoriert.

Sie können folgende Platzhalter verwenden:

```
?      Beliebiges einzelnes Zeichen.
*      Beliebige Gruppe von Zeichen
```

Um alle Felder mit "Dr." und "Schmidt" oder "Schmitt" zu finden, verwenden Sie also:

```
n%=FIND ("*DR*SCHMI?T")
```

Um alle Felder mit " Schmitt" zu finden, würden Sie

```
f%=FIND ("SCHMITT") verwenden.
```

Die Suche funktioniert nur bei String-Feldern!



**FINDFIELD**

OPL32 – SIBO

```
n%= FINDFIELD(string$,startfeld%,anzahl%,flag%)
```

FINDFIELD kann wie FIND ebenfalls nur in Stringfeldern suchen, ist aber flexibler.

Der Suchbegriff steht in string\$. In startfeld% steht die Nummer des ersten Stringfeldes, ab dem gesucht wird. Der Parameter anzahl% enthält die Anzahl der rechts von startfeld% stehenden Stringfelder, die in jedem Datensatz durchsucht werden sollen, einschließlich dem ersten Feld. Ist anzahl%= 1, durchsucht man nur ein einziges Feld.

Mit flag% bestimmt man die Suchrichtung:

```
flag%= 0 rückwärts ab aktuellem Datensatz
flag%= 1 vorwärts ab aktuellem Datensatz
flag%= 2 rückwärts vom Ende der Ansicht
flag%= 3 vorwärts vom Anfang der Ansicht
```

Addieren von 16 zu flag% erlaubt die Unterscheidung nach Groß- und Kleinschreibung.

Bei Erfolg wird die Datensatznummer in n% zurückgeliefert und der Datensatz wird zum aktuellen. Um weitere Datensätze zu finden, nachdem FINDFIELD bereits erfolgreich war, müssen Sie vorher NEXT verwenden.

```
OPEN DBnam$ + " SELECT Name,Augenfarbe FROM
    Liste",D,name$,farbe$ REM alles in eine Zeile!
WHILE NOT EOF
    FINDFIELD("*gr??",1,1,3)
    PRINT A.name$, A.farbe$
NEXT
ENDWH
CLOSE
```

Siehe auch FIND

**FINDLIB**

SIBO

```
ret%=FINDLIB(kat$,name$)
```

Sucht die DYL-Kategorie name\$ (inklusive Erweiterung .DYL) im ROM. Bei Erfolg wird 0 zurückgegeben und der Handle in kat% gespeichert.

**FIRST**

OPL32 – SIBO

```
FIRST
```

Macht den ersten Datensatz in der aktuellen Datenbank/Ansicht zum aktuellen Datensatz.

Siehe auch BACK, NEXT, LAST

**FIX\$**

OPL32 – SIBO

```
string$=FIX$(x,y%,z%)
```

Macht aus der Fließkommazahl x einen String, der y% Dezimalstellen anzeigt und bis zu z% Zeichen lang ist. Ist z% zu klein, werden Asterisks ("Sternchen") angezeigt. Bei negativen z%-Wert wird der String rechtsbündig dargestellt, den Rest des Platzes nehmen Leerzeichen ein.

Beispiele auf der nächsten Seite.

```

FIX$(12345,3,9)    --> "12345,000"
FIX$(12345,3,7)    --> "*****"
FIX$(12345,3,-16)  --> "          12345,000"

```

Siehe auch GEN\$, NUM\$, SCI\$

## FLAGS

*F - OPL32*

FLAGS flags%

FLAGS kann nur zwischen APP...ENDA verwendet werden.

flag%= 1 Es wird eine Applikation erzeugt, die Dateien erzeugen kann, die eine eigene "Kennung" besitzen, die sie dem Programm zuordnet. Klickt man eine solche Datei an, wird automatisch das zugehörige Programm aufgerufen (z.B. ist das bei WORD der Fall), oder man kann auch eine neue Datei am Systembildschirm erzeugen, dann findet sich das zugehörige Programm in der Auswahlbox.

flag%= 2 Verhindert, dass die Applikation in der Extras-Leiste angezeigt wird.

## FLT

*OPL32 – SIBO*

```

y=FLT(x&)
y=FLT(x%)

```

Wandelt den Integer-Ausdruck x& bzw. x% (Long- oder Normal-Integer-Zahl) in das Fließkommaformat um.

## FONT

*OPL32 – SIBO*

FONT font&, stil%

Setzt für das Textfenster Fontart und -stil. Jeder FONT-Befehl löscht zuerst das Textfenster!

font&	Name	Pixelhöhe
4	Courier	8
5	Times	8
6	Times	11
7	Times	13
8	Times	15
9	Arial	8
10	Arial	11
11	Arial	13
12	Arial	15
13	Tiny	4

Die Darstellung der Schrift wird zusätzlich durch den Stil beeinflusst:

stil%	Aussehen
0	normal
1	fett
2	unterstrichen
4	invers
8	doppelte Höhe
16	leichter Zeichenabstand
32	kursiv

Die Werte dürfen durch Addieren kombiniert werden.

**FREEALLOC***F - OPL32 – SIBO*

```
FREEALLOC pzell&    REM in OPL32
FREEALLOC pzell%    REM in SIBO
```

Gibt die reservierte Zelle mit Adresse pzell&/pzell% wieder frei.

Der Befehl gehört zum Komplex "Dynamische Speicherverwaltung", Anfänger: Finger weg!

Siehe auch ALLOC, ADJUSTALLOC, REALLOC

**gAT***OPL32 – SIBO*

```
gAT x%,y%
```

Setzt den Cursor auf absolute Pixel-Koordinaten im aktuellen Fenster. Die linke obere Ecke wird durch gAT 0,0 erreicht. Negative Koordinaten sind möglich.

Siehe auch gMOVE

**gBORDER***OPL32 – SIBO*

```
gBORDER flag%
gBORDER flag%,weite%,hoehe%
```

Zieht einen Rahmen um das aktuelle Fenster, bei Bedarf mit einem schmalen, nach rechts unten gerichteten schwarzen Schatten. Einfluss von flag%:

```
flag%= 0 kein Schatten
flag%= 1 Schatten mit einem Pixel Breite
flag%= 3 Schatten mit zwei Pixel Breite
flag%= $200 rundet die Ecken etwas mehr ab
```

Kombination der runderen Ecken mit den anderen Werten sind möglich, z.B. \$203

OPL32: "Echte Schatten" erzeugt man am besten gleich mit gCREATE/gXBORDER.

Siehe auch gXBORDER, gBOX

**gBOX***OPL32 – SIBO*

```
gBOX weite%,hoehe%
```

Zeichnet im aktuellen Fenster ab der aktuellen Position eine Box nach rechts (weite%) und unten (hoehe%). Die Startposition des Cursors wird nicht verschoben.

**gBUTTON***OPL32*

```
gBUTTON text$,type%,weite%,hoehe%,status%
gBUTTON text$,type%, weite%,hoehe%,status%,bitmapId&
gBUTTON text$,type%, weite%,hoehe%,
                        status%,bitmapId&,maskId&
gBUTTON text$,type%, weite%,hoehe%, status%,
                        bitmapId&,maskId&,layout%
```

gBUTTON zeichnet einen 3D-Button, dessen Aussehen sich noch beeinflussen lässt. In der einfachsten Form zeigt er einen Text (text\$). Durch Gestaltung von weite% und hoehe% muss man selber dafür sorgen, dass der Text auch in den Button passt. Er beginnt an der aktuellen Cursorposition und zeichnet dann nach rechts

und unten. Über `status%` regelt man den optischen Zustand: der Knopf kann erhaben oder halb oder ganz gedrückt sein.

```
status%= 0 Urzustand, ungedrückt
status%= 1 halb gedrückt
status%= 2 vollständig gedrückt
```

Bei Verwendung von `status%=2` verschwindet unschönerweise der Text auf dem Button.

Für EPOC32-Geräte nimmt `type%` den Wert 2 an (für SIBO ist er 1).

Zusätzlich zum Text lassen sich Bitmaps auf dem Button positionieren. Diese werden über Identifikationsvariablen `bitmapId&` und `maskId&` angesprochen. Das bedeutet gleichzeitig, dass die Bitmaps bereits existieren und geladen sein müssen. Wird keine Maske verwendet, darf `maskId&` Null sein. Das kommt zum Tragen, wenn man `layout%` angeben will. Benutzung einer Maske wird empfohlen, um unschöne weiße Bereiche auf dem grauen Hintergrund des Button zu vermeiden. Als Maske ist auch das eigentliche Bitmap verwendbar.

Mit `layout%` wird die relative Position von Text und Bitmap bestimmt:

```
layout%= 0 Textposition rechts
layout%= 1 Textposition unten
layout%= 2 Textposition oben
layout%= 3 Textposition links
```

Addiert man die folgenden Werte, legt man fest, wie verbleibender Freiraum zwischen Text und Bitmap verteilt wird, was jedoch unterschiedliche Auswirkung bei vertikaler und horizontaler Anordnung hat:

```
layout%= 0 Text und Bitmap teilen sich den Freiraum
layout%= $10 Text bekommt den Freiraum zugeschlagen
layout%= $20 Bitmap bekommt den Freiraum zugeschlagen
```

## **gBUTTON**

*SIBO*

```
gBUTTON text$,type%,b%,h%,modus%
```

Zeichnet eine 3D-Taste in der schwarzen und grauen Ebene an der aktuellen Bildschirmposition in einem `b%` breiten und `h%` hohen Rechteck. Das Rechteck umfaßt die Taste in allen möglichen Modi. `text$` enthält den bis zu 64 Zeichen langen Text, der innerhalb der Taste mit aktuellen Font und Schriftstil geschrieben wird. Es liegt in Ihrer Verantwortung, dass der Text in die Taste paßt.

`type%=0` Taste wie im Serie 3

`type%=1` Taste wie im Serie 3a

Die Bedeutung von `modus%` hängt von `type%` ab:

`type%=0`: `state%=0` Taste nicht gedrückt, `state%=1` Taste gedrückt.

`type%=1`: `state%=0` Taste nicht gedrückt, `state%=1` Taste halb gedrückt, `state%=2` Taste ganz gedrückt. Wenn das aktuelle Fenster keine graue Ebene besitzt, wird ein Fehler ausgegeben.

## **gCIRCLE**

*OPL32*

```
gCIRCLE radius%
gCIRCLE radius%,fuellung%
```

Zeichnet einen Kreis, dessen Mittelpunkt sich an der aktuellen Cursorposition befindet, mit dem Radius `radius%`. Wenn das Argument `fuellung%` ungleich Null ist, wird der Kreis mit der aktuellen Vordergrundfarbe gefüllt.

Siehe auch gELLIPSE, gCOLOR

## gCLOCK

OPL32

```
gCLOCK ON/OFF
gCLOCK ON,modus%
gCLOCK ON,modus%,offset&
gCLOCK ON,modus%,offset&,format$
gCLOCK ON,modus%,offset&,format$,font&
gCLOCK ON,modus%,offset&,format$,font&,style%
```

Es wird eine Uhr mit der Systemzeit dargestellt (ON), deren Aussehen sich beeinflussen über modus% beeinflussen lässt. Mit OFF wird die Uhr wieder ausgeblendet.

offset& benennt einen Offset von der Systemzeit in Minuten, dadurch lässt sich eine Uhr mit einer anderen als der Systemzeit darstellen.

```
modus% Uhrtyp
6 schwarz und grau, mittel, nach Systemeinstellung
7 schwarz und grau, mittel, analog
8 zweiter Typ, mittel, digital
9 schwarz und grau, sehr groß
11 digital, formatiert
```

Sie können obige Modi über OR mit folgendem Werte kombinieren:

\$100 --> offset& darf jetzt in Sekunden angegeben werden

Werden modus% und offset& nicht explizit angegeben, sind die Standardwerte modus%=1 und offset&=0.

Angaben zu format\$, font& und style% werden nur für formatierte Uhren benötigt. Die Werte für font& und style% sind dieselben wie sie für gFONT und gSTYLE verwendet werden. Stil-Standardwert ist 0.

format\$ gibt an, wie die Uhr angezeigt wird. Der String kann bis zu 255 Zeichen lang sein und enthält verschiedene Format-Spezifizierer, die immer aus einem %-Zeichen und einem Buchstaben bestehen. Groß-/Kleinschreibung wird dabei nicht beachtet.

Format	Anzeige (Abk.)
%%	%-Zeichen
%%*	Kürzt den folgenden Eintrag ab. der Asterisk soll zwischen %-Zeichen und der folgendne Zahl oder dem folgenden Buchstaben stehen, z.B. %*1. Das verhindert die Anzeige führender Nullen, z.B. am ersten eines Monats zeigt "%F %*M" eine "1" anstelle einer "01"
%D,%W,%M	Tag, Kalenderwoche, Monat als Zahl, 01...31, 01...53, 01...12
%E,%N	Tag/Monat als Text (bei Abkürzung: 3 Zeichen lange Kürzel)
%H,%I	Stunden im 24- bzw. 12-Stunden Format.
%S,%T	Sekunden/Minuten als Zahl, 01...59.
%X	Suffix für Tag im Datum (Punkt für 1.Dezember).
%Y	Jahr als 4-stellige Zahl (Abk.: ohne Jahrhundert)
%:n	Stunden-Minuten-Sekunden-Trennzeichen, Ganzzahl n=0,1,2 oder 3, für Europa kommen nur 1 und 2 in Frage.
%/n	Tag-Monat-Jahr-Trennzeichen, Ganzzahl n=0,1,2 oder 3, für Europa kommen nur 1 und 2 in Frage.
%1,%2,%3	Tag,Monat,Jahr in der Reihenfolge, die in der Applikation "Uhr" eingestellt wurde. Im europäischen Zeitformat ist also %1=%T, %2=%M, %3=%Y. Um immer das voreingestellte Format zu verwenden geben Sie also einfach "%1,%2,%3" an. (Sehen Sie bei %G,%P,%U nach)
%4,%5	Tag und Monat, wie in der Applikation "Uhr" eingestellt.

(Fortsetzung nächste Seite)

(Fortsetzung)

Format	Anzeige (Abk.)
%	Verwendet "am" oder "pm" in Abhängigkeit von der aktuellen Sprache und der Uhrzeit, auch wenn eine 24-Uhr verwendet wird. Der Text kann vor oder hinter der Zeit mit einem Abstand dargestellt werden. In der Kurzversion mit "%*A" wird der Abstand weggelassen. Verwendung von Plus- bzw. Minuszeichen: %- "am/pm"-Text wird nur verwendet, wenn die System-Einstellungen der "am/pm"-Anzeige die Darstellung vor der Zeit verlangen, %+ wirkt nur, wenn die Einstellungen auf "am/pm nach der Zeit" stehen. Um beide Fälle einzuschließen, verwendet man z.B.: "%-A%H%:1%T%+A"
%B	Wie %A, nur dass "am/pm"-Text ausschließlich verwendet wird, wenn die Systemuhr auf das 12-Stunden-Format eingestellt ist
%D	Tag als Zahl 01...31
%E	Tag als Text (auf drei Zeichen gekürzt)
%F	Die Datum-Zeit-Formatierung wird unabhängig von den Systemeinstellungen (Abkürzungen nicht möglich)
%J	Stunden im 12- oder 24-Stunden-Format in Abhängigkeit von den Systemeinstellungen. Im 12-Stunden-Format wird die führende Null immer unterdrückt, auch wenn ein "*" zwischen % und J angegeben wurde.
%H	Stunden im 24-Stunden-Format.
%%I	Stunden im 12-Stunden-Format.
%M	Monat als Zahl 01...12
%N	Monat als Text (auf drei Zeichen gekürzt)
%S	Sekunden/Minuten als Zahl, 01...59.
%T	Sekunden/Minuten als Zahl, 01...59.
%W	Kalenderwoche als Zahl 01...53
%X	Suffix für Tag im Datum (Punkt für 1.Dezember).
%Y	Jahr als 4-stellige Zahl (Abk.: ohne Jahrhundert)
%Z	Tag innerhalb des Jahres als dreistellige Zahl

Beispiele:

Das Beispiel bezieht sich auf Mittwoch, den 1. Januar 1997, 13:30:05 (=1:30:05 pm). Die zugehörigen Voreinstellungen: Europäisches Datum, mit am/pm nach der Zeit:

1. "%-A%I:%T:%S%+A"

Darstellung: Zeit als 12-Stunden-Uhr mit Sekunden, am/pm vor oder nach der Zeit (abhängig von der Systemeinstellung), z.B.: 1:30:05 pm.

2. "%F%E %\*D%X %N %Y"

Darstellung: Wochentag, gefolgt vom Datum mit Suffix, dem Monat als Wort und der Jahreszahl, z.B. Mittwoch 1. Januar 1997.

3. "%E %D%X%N%Y %1 %2 %3"

Darstellung: Benutzt die lokalen Einstellungen für die Anordnung der Datumselemente, besitzt aber ein Suffix aus Tag und Monat (als Text), z.B.: Mittwoch 01. Januar 1997.

4. "%\*E %\*D%X%\*N%\*Y %1 %2 %3"

Darstellung: Ähnlich wie Beispiel 3, kürzt aber den Wochentag, den Tag sowie Monat und Jahr ab, z.B.: Mit 1. Jan 97.

5. "%M%Y%D%1%/0%2%/0%3"

Darstellung: 01/01/1997. Die Anordnung von %D, %M and %Y ist hier also irrelevant, wenn lokal-abhängige Formatierung benutzt wird. Anstatt dessen ist die Anordnung der Datums-Teile von den Formatierungsbefehlen %1, %2, and %3 abhängig.

style% kann jeden Wert, wie er auch für gSTYLE verwendet wird, annehmen, jedoch nicht 2 (unterstrichen).

## OPL-Befehle (alphabetische Liste)

Der Fehler "Allgemeiner Fehler" tritt auf, wenn versucht wird, eine ungültige Formatierung zu verwenden. Das ist auch der Fall, wenn "falsche", nicht zur Lokalisierung des Gerätes passende Zeit- bzw. Datums-Trenner verwendet werden oder "%+" und "%-" nicht mit "A" bzw. "B" zusammen verwendet werden.

Verwenden Sie nie `gSCROLL` um den Bereich des Fensters zu verschieben, der die Uhr enthält. Die neue Uhrzeit würde bei einer Änderung der Uhrzeit wieder an der alten Stelle angezeigt. Sie können nur das gesamte Fenster verschieben.

Die Darstellung der Uhr mit Grautönen (oder Farbe beim netBook) hängt vom gesetzten Farbmodus im Fenster ab (`gCREATE`)!

## **gCLOCK**

*SIBO*

```
gCLOCK ON/OFF
gCLOCK ON mode%
gCLOCK ON mode%,offset%
gCLOCK ON mode%,offset%,format$
gCLOCK ON mode%,offset%,format$,font%
gCLOCK ON mode%,offset%,format$,font%,style%
```

Zeigt oder entfernt eine Uhr, die die Systemzeit anzeigt. Es kann nur eine Uhr je Fenster angezeigt werden. Die Uhr wird an der aktuellen Bildschirmposition ausgegeben.

`mode%` gibt an welcher Typ verwendet werden soll. Die Typen 1-5 stehen für Serie 3-kompatible Uhren.

`mode%` Uhrtyp

1	klein, digital	
2	mittel, entsprechend	Systemeinstellung
3	mittel, analog	
4	mittel, digital	
5g	groß, analog	

Für den Serie 3a/c/mx stehen folgende zusätzliche Typen zur Verfügung:

<code>mode%</code>	Uhrtyp
6	schwarz und grau, mittel, nach Systemeinstellung
7	schwarz und grau, mittel, analog
8	zweiter Typ, mittel, digital
9	schwarz und grau, sehr groß
10	digital, formatiert

Sie können obige Modi über OR mit folgenden Werten kombinieren:

\$10	zeigt das Datum in allen Uhren außer der sehr großen und der formatierten.
\$20	zeigt die Sekunden in der kleinen digitalen, der großen analogen, der schwarz/grauen mittleren analogen sowie der extra großen Uhr.
\$40	zeigt am/pm in kleinen digitalen, und schwarzen mittleren Uhren.
\$80	zeigt die Uhr in der grauen Ebene (nur bei Uhren die nicht beide Ebenen enthalten).

Verwenden Sie nie `gSCROLL` um den Bereich des Fensters zu verschieben, der die Uhr enthält. Die neue Uhrzeit würde bei einer Änderung der Uhrzeit wieder an der alten Stelle angezeigt. Sie können nur das gesamte Fenster verschieben.

Sie können schwarz/grau Uhren auch in Fenstern ohne graue Ebene zeichnen.

Digitale Uhren verwenden den 12-/24-Stunden Modus je nach Einstellung auf Systemebene. Bei digitalen Uhren und 12-Stunden Modus kann mit Hilfe der Option \$40 am/pm angezeigt werden (s. o.).

offset% gibt an um wieviele Minuten die angezeigte Uhrzeit von der Systemzeit abweichen soll. Sie können also mit offset% eine andere Zeit als die Systemzeit anzeigen.

Wird mode% oder offset% nicht angegeben, ist mode%=1 und offset%=0.

Formatierte digitale Uhren:

Sie können mit font% und style% die verwendete Schrift einstellen. Verwenden Sie die Werte entsprechend den Befehlen gFONT und gSTYLE. Der Standardfont ist \$9 (Systemfont), der Standard-Schriftstil ist 0 (normal).

format\$ gibt an wie die Uhr angezeigt wird. Der String kann bis zu 255 Zeichen lang sein und enthält verschiedene Format-Spezifizierer, die immer aus einem %-Zeichen und einem Buchstaben bestehen. Groß-/Kleinschreibung wird dabei nicht beachtet. Um die Anzeige so kurz wie möglich zu gestalten, können Sie mit \* nach dem % alle angezeigten Werte abkürzen. Bei Zahlenwerten würden in diesem Fall führende Nullen weggelassen, bei anderen Ausgaben werden die in der folgenden Liste der Format-Optionen angegebenen Abkürzungen benutzt.

Format	Anzeige (Abk.)
%%	%-Zeichen
%, %/, %:	im Setup der Applikation "Uhr" eingestellte Uhrzeit- und Datums trennzeichen.
%	"am" oder "pm" (Abk.: "a" oder "p")
%D, %W, %M	Tag, Kalenderwoche, Monat als Zahl, 01...31, 01...53, 01...12
%E, %N	Tag/Monat als Text (Abk.: 3 Zeichen lange Kürzel)
%H, %I	Stunden im 24- bzw. 12-Stunden Format.
%S, %T	Sekunden/Minuten als Zahl, 01...59.
%X	Suffix für Tag im Datum (Punkt für 1.Dezember).
%Y	Jahr als 4-stellige Zahl (Abk.: ohne Jahrhundert)
%1, %2, %3	Tag, Monat, Jahr in der Reihenfolge, die in der Applikation "Uhr" eingestellt wurde. Im europäischen Zeitformat ist also %1=%T, %2=%M, %3=%Y. Um immer das voreingestellte Format zu verwenden geben Sie also einfach "%1,%2,%3" an. (Sehen Sie bei %G, %P, %U nach)
%4, %5	Tag und Monat, wie in der Applikation "Uhr" eingestellt.
%F, %O	schaltet Tag und Monat von %1,%2,%3 zwischen numerischer und String-Darstellung um. Zum Beispiel "%1,%F%1%F%1" ergibt "09Dienstag09" (am 9.3.93)
%G, %P, %U	schaltet für %1,%2,%3 zwischen langer Darstellung und Abkürzung um.
%L	schaltet die Darstellung des Tags zwischen "mit" und "ohne" Suffix um. Am 16. September 1993 folgt auf das Format "%L%1%L%2%3" die Ausgabe "16.091993"
%6, %7	stellt die Stunden und den "am"/"pm"-Text, gemäß "Uhr"-Setup dar. Im 24-Stunden Format wird kein "am"/"pm"-Text dargestellt.

Der Format-String "%1%/%2%/ %3" zeigt also das Datum wie in der Applikation "Uhr" an. "%4%/ %5" zeigt nur Tag und Monat im eingestellten Format an. "%6%:%T%:%S%7" zeigt eine Uhr mit Stunden, Minuten und Sekunden an, die immer konform mit den Systemeinstellungen ist.

Die Umschalter, wie z.B. %F, wirken immer nur innerhalb eines Format-Strings. Wurde also mit %F umgeschaltet und eine neue Uhr wird angezeigt, so wird für diese Uhr wieder die Grundeinstellung verwendet.

Als letztes Beispiel hier noch der Format-String "%7%G%L%P%O%\*E, %1 %2 %3 %6%:%T%:%S". Bei den Systemeinstellungen "Tag/Monat/Jahr" für das Datum, "am-pm" für die Uhrzeit und "." als Trennzeichen für die Uhrzeit wird um 11 Uhr 30:05 pm 9. März 1993 ausgegeben: "Di, 9. Mar 1993 11:30:05pm". Derselbe Format-String gibt für die Einstellungen "Monat/Tag/Jahr" und 24-Stunden Modus aus: "Di, Mar 9. 1993 23:30:05".

Systemfehler-Workaround: Schalten Sie vor dem Schließen des Fensters, in dem sich die Uhr befindet, die Uhr-Anzeige mit gCLOCK OFF ab.



**gCLOSE**

OPL32 – SIBO

`gCLOSE id%`

Schließt das Fenster mit der Kenn-Nummer `id%`. Betrifft alle Fenster, die vorher mit `gCREATE`, `gCREATEBIT` oder `gLOADBIT` geöffnet wurden. Das Basis-Fenster lässt sich also nicht schließen.

SIBO: Vor dem Schließen des Fensters wird empfohlen, vorhandene Uhren zu schließen (`gCLOCK OFF`) und "`gGREY 0`" anzuwenden, wenn das Fenster eine Grauebene hat.

**gCLS**

OPL32 – SIBO

`gCLS`

Löscht den Inhalt des aktuellen Fensters, der Cursor kehrt in die Ruhestellung 0,0 zurück. Achtung, eventuelle Rahmen (`gBORDER`) gehen verloren (`gBORDER` neu anwenden), Schatten aber bleiben erhalten!

**gCOLOR**

OPL32

`gCOLOR rot%,gruen%,blau% (mögl. Wertebereich 0 .. 255)`

Setzt die Pen-Farbe des aktuellen Fensters. Sind die Werte von rot, grün und blau gleich groß, entstehen reine Grauwerte (0= schwarz, 255= weiß). Bei Bildschirmen ohne Farbdarstellung wird immer auf grau umgerechnet.

**gCOLORBACKGROUND**

OPL32!

`gCOLORBACKGROUND rot%,gruen%,blau%`

Setzt die Hintergrundfarbe eines Fensters. Nachfolgende Grafikbefehle benutzen dann im aktuellen Fenster diese Einstellung.

Der Befehl lässt sich richtig wirkungsvoll nur auf Farbgeräten verwenden.

Die Werte für `rot%`, `gruen%` und `blau%` dürfen von 0 (dunkel) bis 255 (hell) reichen. Beispiel:

```
gUSE 1
gCOLOR 255,0,0
gCOLORBACKGROUND 0,0,255
gAT 50,50
gPRINTB "Roter Text auf blauem Hintergrund",250
```

**gCOLORINFO**

OPL32!

`gCOLORINFO cinfo&()`

Stellt Farbinformationen für das verwendete System zusammen und liefert sie in `cinfo&()` zurück. `cinfo&()` muss wenigsten für 7 Elemente deklariert sein.

Die einzelnen Elemente enthalten jeweils die maximal möglichen, nicht die gerade benutzten Werte! Für aktuelle Werte des Anzeigemodus verwende man `gINFO32`.

Die mit `gCOLORINFO` verfügbaren Informationen sind:

```
cinfo&(1) = Anzeigemodus des Standardfensters
cinfo&(2) = Anzahl der unterstützten Farben
cinfo&(3) = Anzahl der unterstützten Graustufen
cinfo&(4) .. (7) = für spätere Nutzung
```

Dem Anzeigemodus in cinfo&(1) sind die folgenden Werten zugeordnet:

```

1 =          2 Graustufen (= schwarz/weiss, Monochrom)
2 =          4 Graustufen
3 =         16 Graustufen
4 =        256 Graustufen
5 =         16 Farben
6 =        256 Farben
7 =       65.536 Farben (16-Bit Farbe)
8 = 16.777.216 (24-Bit Farbe)
9 =          RGB-Displaymodus
10 =        4.096 Farben (12-Bit Farbe)

```

Warnung! Der Befehl ist erst ab EPOC32/Version 5 verfügbar und kann nur hier verwendet werden. Wird er auf Geräten mit früheren Versionen eingesetzt, kommt es zu einem Fehler! Lässt man übersetzte EPOC32/v.5-Programme auf älteren Geräten laufen, ergibt sich ebenfalls ein Fehler. Hier gibt es jedoch ein von Symbian empfohlenen Workaround, der hier im Original wiedergegeben wird:

```

REM testColorSupport.opl
REM (C) 1998-1999 Symbian Ltd. All rights reserved.
INCLUDE "Const.oph"
DECLARE EXTERNAL
EXTERNAL Main:
EXTERNAL ColorAvailable%:

PROC Main:
    REM This will run on ER1 to ER5 devices (but will only
    REM translate on ER5). Attempt to find
    REM the color capabilities of the device.
    IF ColorAvailable%:
        PRINT "This device has color support."
    ELSE
        PRINT "No color support on this device."
    ENDIF
    GET
ENDP

PROC ColorAvailable%:
    LOCAL cinfo&(7)
    ONERR ErrorHandler::
    gCOLORINFO cInfo&()
    IF cInfo&(gColorInfoANumColors%)
        RETURN KTrue%
    ENDIF
    RETURN KFalse%
ErrorHandler::
    ONERR OFF
    IF ERR=-96 REM Illegal Opcode
        REM gCOLORINFO not supported
        RETURN KFalse% REM No color support.
    ENDIF
    RAISE ERR          REM Not expecting this error.
ENDP

```

## **gCOPY**

*OPL32 – SIBO*

`gCOPY id%,x%,y%,weite%,hoehe%,modus%`

Kopiert eine rechteckige Fläche mit den Abmessungen weite% \* hoehe% von der Stelle x%,y% aus dem Fenster oder von der Bitmap mit der Kenn-Nummer id% in das aktuelle Fenster an die aktuelle Position.

Der Parameter modus% bestimmt, wie die kopierte Fläche in das Ziel einkopiert wird:

```

modus%= 0 kopiert gesetzte Pixel der Quelle und setzt
           sie im Ziel
modus%= 1 kopiert gesetzte Pixel der Quelle und löscht
           gesetzte Pixel im Ziel
modus%= 2 kopiert gesetzte Pixel der Quelle und
           invertiert die Pixel im Ziel
modus%= 3 kopiert alle Pixel der Quelle (einschl.
           der weißen) und ersetzt alle Pixel im Ziel

```

Die aktuelle Position wird dabei weder im Quell- noch im Zielfenster verändert.

gCOPY wird von gGREY wie folgt beeinflusst:

```

gGREY 0   schwarze Ebene wird auf schwarze Ebene kopiert.
gGREY 1   graue Ebene wird auf graue Ebene kopiert, bzw.,
           schwarz nach grau, wenn Quellfenster nur
           schwarz.
gGREY 2   schwarz nach schwarz und grau nach grau bzw.
           schwarz nach schwarz und grau wenn Quellfenster
           nur schwarz.

```

OPL32: Aus Geschwindigkeitsgründen wird empfohlen, mit diesem Befehl nur Daten aus Bitmaps - nicht aus Fenstern - in andere Bitmaps oder Fenster zu kopieren.

## **gCREATE**

*OPL32 – SIBO*

```

id%=gCREATE(xpos%,ypos%,weite%,hoehe%,sichtbarkeit%)
id%=gCREATE(xpos%,ypos%,weite%,hoehe%,sichtbarkeit%,flag%)

```

Erzeugt ein Fenster an der Stelle xpos%,ypos% mit den Abmessungen weite% und hoehe%. Das neue Fenster ist automatisch das aktuelle und steht zugleich im Vordergrund. Die anfängliche Cursorposition im Fenster ist 0,0.

Ist der Wert von sichtbarkeit% gleich 1, wird das Fenster auch sofort sichtbar, bei 0 ist es unsichtbar.

Der Parameter flag% sorgt für die Einstellung des Farbmodus und die Schattierung (nicht die Rahmung!) des Fensters. Ohne Angaben wird der 2-Farb-Modus gesetzt und das Fenster hat keinen Schatten.

Die niedrigsten 4 Bits von flag% enthalten den Farbmodus (Zahlenangaben hexadezimal):

```

$0=   2-Graustufen-Modus
$1=   4-Graustufen-Modus
$2=  16-Graustufen-Modus
$3= 256-Graustufen-Modus
$4=   16-Farben-Modus
$5= 256-Farben-Modus

```

Die folgenden vier Bits sind für die Schattenform zuständig:

```

$00= kein Schatten
$10= Schatten

```

Die nächsten vier Bits stehen für die Schattenhöhe und den Versatz (ein Wert von n ergibt einen Schatten von 2n Pixel), z.B. bei n=2:

```

$200= Schatten mit 2 Pixel Versatz und 4 Pixel Breite

```

Die einzelnen Werte werden miteinander kombiniert und ergeben dann z.B.:

```

$112 = 16-Graustufen-Modus, mit 2 Pixel breitem Schatten

```

Es dürfen bis zu 63 Fenster zusätzlich zum Basisfenster angelegt werden.

Das Fenster bekommt automatisch eine Kenn-Nummer, die man in id% wiederfindet. Über die Kenn-Nummer kann das Fenster mit anderen Befehlen angesprochen werden.

Siehe auch gCLOSE, gGREY, DEFAULTWIN

### **gCREATE**

*SIBO*

```
id%=gCREATE(x%,y%,b%,h%,s%)
id%=gCREATE(x%,y%,b%,h%,s%,grau%)
```

Erstellt ein Fenster an der angegebenen Bildschirmposition(x%,y%) mit der angegebenen Größe (b%=Breite,h%=Höhe). Die aktuelle Position im neuen Fenster ist 0,0 (links oben). Wenn s%=1 ist wird das Fenster sofort sichtbar, bei s%=0 ist es unsichtbar.

Zurückgeliefert wird die ID-Nummer des Fensters (2-8) über die das Fenster in anderen Befehlen adressiert wird.

Siehe auch gCLOSE, gGREY, DEFAULTWIN

### **gCREATEBIT**

*OPL32 – SIBO*

```
id%=gCREATEBIT(weite%,hoehe%)
id%=gCREATEBIT(weite%,hoehe%,flag%) REM nur OPL32
```

Erzeugt ein nicht sichtbares, aber über die Kenn-Nummer id% durch die verschiedensten Befehle ansprechbares Bitmap-Fenster. Der darin befindliche Cursor steht auf Position 0,0. Das Bitmap ist automatisch das aktuelle.

OPL32: Wird flag% genutzt, lassen sich Einstellungen der Farb-Eigenschaften wie bei gCREATE einstellen. Der voreingestellte Wert lässt nur 2-Farb-Bitmaps zu.

```
$0= 2-Graustufen-Modus
$1= 4-Graustufen-Modus
$2= 16-Graustufen-Modus
$3= 256-Graustufen-Modus
$4= 16-Farben-Modus
$5= 256-Farben-Modus
```

Siehe auch gCLOSE, gCREATE

### **gDRAWOBJECT**

*SIBO*

```
gDRAWOBJECT type%,flags%,b%,h%
```

Zeichnet das skalierbare Grafikobjekt, das in type% angegeben ist, passend in das Rechteck der Breite b% und Höhe h% an der aktuellen Bildschirmposition. Der Serie 3a stellt für diese Funktion nur einen Objekttyp (type%=0) zur Verfügung. Es ist eine 3D-Box mit runden Ecken und einem Schatten rechts unten. Die Box selbst ist grau. Ein Beispiel finden Sie beim Text "Stadt" in der Weltzeituhr.

Beim Typ 0 definiert flags% die Rundung der Ecken:

```
0 normale Rundung
1 stärker gerundet
2 ein einzelnes Pixel in jeder Ecke ausgeschnitten.
```

Wenn das aktuelle Fenster keine graue Ebene hat, wird ein Fehler ausgegeben.

**gELLIPSE**

OPL32

```
gELLIPSE h-radius%,v-radius%
gELLIPSE h-radius%,v-radius%,fuellung%
```

Zeichnet eine Ellipse, deren Zentrum die aktuelle Cursorposition in der aktuellen Zeichenfläche darstellt und deren Radien durch h-radius% und v-radius% gegeben sind. Ist der Parameter fuellung% ungleich Null, wird die Ellipse mit der aktuellen Vordergrundfarbe ausgefüllt.

Siehe auch gCIRCLE, gCOLOR

**GEN\$**

OPL32 – SIBO

```
string$=gen$(x,z%)
```

Macht aus der Fließkommazahl x einen String, der bis zu z% Zeichen lang ist. Ist z% zu klein, werden Asterisks ("Sternchen") angezeigt. Bei negativen z%-Wert wird der String rechtsbündig dargestellt, den Rest des Platzes nehmen Leerzeichen ein. Wenn z% und x nicht zueinander passen - der Dezimaltrenner genau mit hineinfällt ohne folgende weitere Dezimalstelle - werden Asterisks ("Sternchen") zurückgeliefert.

Beispiele:

```
GEN$(123.45,6) à "123,45"
GEN$(123.45,3) à "123"
GEN$(123.45,4) à "****"
GEN$(123,6) à "123"
GEN$(123,-6) à "    123"
```

Siehe auch FIX\$, NUM\$, SCI\$

**GET**

OPL32 – SIBO

```
taste%=GET
```

Das Programm stoppt und wartet auf den Druck einer beliebigen Taste. Ausgenommen sind Tasten, die lediglich modifizierenden Charakter haben, also <Strg> und <Shift>. Der Tastendruck liefert in GET den Tastaturcode zurück. Das Ergebnis wird einer passend deklarierten Variablen übergeben. Die Tastaturcodes sind im Anhang zu finden.

OPL32: Wird zusammen mit den Buchstabentasten die Taste <Strg> gedrückt, bekommt man nicht den Code, sondern die Stellung des Buchstabens im Alphabet zurück, egal ob es sich um den klein- oder großgeschriebenen Buchstaben handelt:

```
"A" --> 1 oder
"d" --> 4
```

Siehe auch KEY

**GET\$**

OPL32 – SIBO

```
taste$=GET$
```

Das Programm stoppt und wartet auf den Druck einer (fast) beliebigen Taste. Ausgenommen sind Tasten, die lediglich modifizierenden Charakter haben, also <Strg> und <Shift>. Der Tastendruck liefert in GET\$ das Zeichen der gedrückten Taste zurück. Das Ergebnis wird einer passenden deklarierten Variablen übergeben. Einige Zeichen können nicht am Bildschirm dargestellt werden. Um herauszufinden, ob dazu eine Modifiziertaste (Strg, Shift, Fn) gedrückt wurde, muss KMOD benutzt werden.

Siehe auch KEY\$

**GETCMD\$***F - OPL32 – SIBO*

w\$=GETCMD\$

Liefert die neuen Kommandozeilenparameter einer Applikation, nachdem eines der Ereignisse "Datei wechseln" oder "Beenden" aufgetreten ist. Das erste Zeichen des Strings ist "C", "O" oder "X". Bei "C" bzw. "O" ist der restliche String ein Dateiname. Das erste Zeichen hat folgende Bedeutung:

"C" schließe die aktuelle und erstelle die angegebene neue Datei.  
 "O" schließe die aktuelle und öffne die bestehende neue Datei.  
 "X" schließe die aktuelle Datei (wenn geöffnet) und verlasse die Applikation.

GETCMD\$ kann für jede Systemnachricht nur einmal aufgerufen werden.

Siehe auch CMD\$

**GETDOC\$***F - OPL32*

docname\$=GETDOC\$

Gibt den Namen des aktuellen Dokumentes zurück.

Siehe auch SETDOC

**GETEVENT***F - OPL32 – SIBO*

GETEVENT a%()

Wartet auf ein Ereignis. Legt das aufgetretene Ereignis in a%() ab. Die in a%() zurückgelieferten Daten hängen vom aufgetretenen Ereignis ab. Wenn das Ereignis ein Tastendruck ist, liefert a%(1) AND \$400 den Wert 0 für andere Ereignisse einen von 0 verschiedenen Wert.

Bei Tastaturereignissen liefert:

a%(1) den Tastencode  
 a%(2) AND \$00ff die gedrückte Modifizierertaste (SHIFT,STRG...)  
 a%(2)/256 den Wiederholungszähler inkl. erstem Tastendruck! (von GET ignoriert)

Folgende weitere Ereignisse können auftreten:

a%(1)=\$401 Programm wurde zum Vordergrundtask  
 a%(1)=\$402 Programm wurde zum Hintergrundtask  
 a%(1)=\$403 Serie 3a wurde eingeschaltet  
 a%(1)=\$404 Dateien sollen gewechselt werden oder die Applikation soll verlassen werden.  
 a%(1)=\$405 Datum hat sich verändert

Ereignisse werden ignoriert, wenn Sie Befehle verwenden, die auf einen Tastendruck warten (GET, GET\$, EDIT, INPUT, MENU und DIALOG). Wenn Sie einen dieser Befehle in einer Applikation verwenden müssen, sollten Sie LOCK ON/LOCK OFF benutzen.

Wenn Sie GETEVENT einsetzen, sollten Sie mögliche Erweiterungen hinsichtlich neuer Ereignisse berücksichtigen.

Bei einem Tastaturereignis steht der Code der Modifiziertaste in `a%(2)` und kann nicht mit `KMOD` gelesen werden.

Wenn andere als Tastaturereignisse überwacht werden sollen, müssen Sie `GETEVENT` anstelle von `GET...` einsetzen, da dieser Befehl alle anderen Ereignisse ignoriert. Sehen Sie unter `LOCK` nach., um bei `GET` und ähnlichen Befehlen zumindest das Ereignis `$404` berücksichtigen zu können.

`a$()` muss mindestens 6 Integers aufnehmen können.

Siehe auch `TESTEVENT`, `GETCMD$`

## GETEVENT32

*F - OPL32*

`GETEVENT32 a&()`

Behandelt alle Events wie `GETEVENT`, berücksichtigt aber außerdem Pointer- Events, also Eingaben über den Stift (oder eine Maus oder Pad, wenn eingebunden). Das Array zur Aufnahme der Werte muss mindestens 16 Elemente besitzen.

Alle Events geben einen 32-Bit-Zeitstempel zurück. Die unten angesprochene Fenster-Id ist die Fensternummer von `gCREATE`.

Der Inhalt der einzelnen Arrayvariablen:

Wenn eine Taste gedrückt wurde, also wenn `(a&(1) AND &400)=0` , gilt:

```
ev&(1)  Tastaturwert
ev&(2)  Zeitstempel
ev&(3)  Scancode (örtliche Zuordnung einer Taste
         auf der Tastatur)
ev&(4)  Modifiziertaste
ev&(5)  Anzahl der Wiederholungen des Tastendrucks
```

Für alle anderen Fälle, bei denen `a&(1) > &400` ist:

```
a&(1)=$401  Progr. wurde z.Vordergrundtask,
             a&(2) = Zeitstempel
a&(1)=$402  Progr. wurde z.Hintergrundtask,
             a&(2) = Zeitstempel
a&(1)=$403  Computer wurde eingeschaltet,
             a&(2) = Zeitstempel, dieser Event ist nur
             aktiv, wenn das durch SETFLAG
             vorher veranlasst wird! s. dort
a&(1)=$404  Dateien sollen gewechselt werden oder die
             Applikation soll verlassen werden. Nach
             diesem Event sollte GETCMD$ aufgerufen
             werden, um festzustellen, welche Reaktion
             stattfinden soll, s. GETCMD$
a&(1)=$406  Taste wurde gedrückt:
             a&(2)  Zeitstempel
             a&(3)  Scancode
             a&(4)  Modifiziertaste
a&(1)=$406  Taste wurde losgelassen:
             a&(2)  Zeitstempel
             a&(3)  Scancode
             a&(4)  Modifiziertaste
a&(1)=$408  Stift-(Pen-)-Event:
             a&(2)  Zeitstempel
             a&(3)  Fenster-ID wie von gCREATE zugeordnet
             a&(4)  Pointer-Typ:
                     0= Pen hat aufgesetzt
                     1= Pen hat abgehoben
                     6= Pen wurde gezogen
             a&(5)  Modifiziertaste
```

```

a&(6)  x-Koordinate im Fenster
a&(7)  y-Koordinate im Fenster
a&(8)  x-Koordinate, bezogen auf
        Basisfenster
a&(9)  y-Koordinate, bezogen auf
        Basisfenster
a&(1)=$409 Pen hat auf dem Bildschirm aufgesetzt:
a&(2)  Zeitstempel
a&(3)  Fenster-ID
a&(1)=$40  Pen hat vom Bildschirm abgehoben:
a&(2)  Zeitstempel
a&(3)  Fenster-ID

```

Einige Pen-Events können ausgefiltert werden, um das Programm nicht mit unerwünschten Events zu überfluten, s. `POINTERFILTER`.

Bei unbekannten Events enthält `a&(1)` zusätzlich zum zurückgegebenen Code den Wert `&1400`.

Ereignisse werden ignoriert, wenn Sie Befehle verwenden, die auf einen Tastendruck warten (`GET`, `GET$`, `EDIT`, `INPUT`, `MENU` und `DIALOG`). Wenn auch noch andere als Tastaturereignisse überwacht werden sollen, müssen Sie immer `GETEVENT32` einsetzen und geeignet auswerten. Müssen Sie doch aus irgendeinem Grunde die genannten Schlüsselwörter verwenden, teilen Sie das dem System mit `LOCK ON` mit. Anforderungen zum Schließen oder zum Dokumentenwechsel gelangen so gar nicht erst zum Programm. Der Systembildschirm meldet in diesem Falle dem Benutzer, z.B. bei dem Versuch, das Programm von "außen" zu beenden, dass das Programm beschäftigt ist. Wenn der Tastendruck erfolgt ist, geben Sie so bald als möglich den aufhebenden Befehl `LOCK OFF`.

Siehe auch `GETEVENT`, `GETEVENTA32`

### **GETEVENTA32**

*F - OPL32*

```
GETEVENTA32 status%,a&()
```

Asynchrone Version von `GETEVENT32`.

Siehe auch `GETEVENTC`, `GETEVENT32`, `GETEVENT`

### **GETEVENTC**

*F - OPL32*

```
GETEVENTC(stat%)
```

Beendet `GETEVENTA32`, der Status kommt nach `stat%` zu stehen.

### **GETLIBH**

*SIBO*

```
cat%=GETLIBH(num%)
```

Konvertiert eine Kategorienummer `num%` in ein Handle. Bei `num%=0` wird der Handle für `OPL.DYL` geliefert.

### **gFILL**

*OPL32 – SIBO*

```
gFILL weite%,hoehe%,gModus%
```

Füllt ab der aktuellen Position eine rechteckige Fläche mit den Abmaßen `weite%*hoehe%`. Wie das geschieht, hängt vom eingestellten Grafik-Modus ab. Der Cursor bleibt unverändert auf seiner Position stehen.

Siehe auch `gMODE`



**gFONT**

OPL32

gFONT fontId&amp;

Der mit der Kennung fontId& benannte Font wird benutzt. Zumeist wird es sich um die ins ROM eingebaute Fonts handeln, es lassen sich aber auch andere Fonts verwenden. Die Nummern der ROM Fonts findet man in der Datei CONST.OPH, besser noch, man benutzt gleich die zugehörige Konstante. Gleichzeitig kann man aber auch noch die von SIBO bekannten Werte benutzen. Es bedeuten:

fontId&	Beschreibung	Pixelgröße
4	Courier	8
5	Times	8
6	Times	11
7	Times	13
8	Times	16
9	Arial	8
10	Arial	11
11	Arial	13
12	Arial	16
13	Tiny (mono)	3x4

Beispiele aus CONST.OPH , Konstantenname und zugehöriger Wert:

```
KFontArialNormal8&= 268435954
KFontArialNormal11&= 268435955
KFontArialNormal13&= 268435956
KFontTimesNormal8&= 268435965
KFontTimesNormal11&= 268435966
KFontTimesNormal13&= 268435967
KFontCourierNormal8&= 268436065
KFontCourierNormal11&= 268436066
KFontCourierNormal13&= 268436067
```

Siehe auch Anhang "Fonts"

Will man den Konstantennamen für fontId& verwenden, muss CONST.OPH gleich zu Beginn noch vor jeder Prozedur in den Quelltext eingebunden sein:

```
INCLUDE "CONST.OPH"
```

Ansonsten verwende man den angegebenen Wert. Fonts aus dem ROM müssen nicht extra mit gLOADFONT geladen werden!

Siehe auch gLOADFONT, FONT

**gFONT**

SIBO

gFONT fontId%

Setzt den Font für das aktuelle Fenster/Bitmap gemäß fontId%. Der Font kann sowohl einer der vordefinierten Fonts als auch ein benutzerdefinierter Font sein.

fontId%	Beschreibung	Pixelgröße
1	Serie 3	normal8
2	Serie 3	fett8
3	Serie 3 Nummern	6x6
4	Mono	8x8
5	Roman	8
6	Roman	11
7	Roman	13
8	Roman	16
9	Swiss	8
10	Swiss	11
11	Swiss	13
12	Swiss	16
13	Mono	6x6

Benutzerdefinierte Fonts müssen zuerst mit gLOADFONT geladen worden sein. Verwenden Sie dann die von gLOADFONT zurückgelieferte Font-ID-Nummer.

Siehe auch gLOADFONT, FONT

**gGMODE**

OPL32 – SIBO

gGMODE modus%

Stellt die Art ein, wie Grafikbefehle in der aktuellen Zeichenfläche ausgeführt werden sollen. Wird gGMODE nicht verwendet, werden die Pixel standardmäßig gesetzt.

```
modus%= 0 Pixel werden gesetzt
modus%= 1 (unterliegende) Pixel werden gelöscht
modus%= 2 (unterliegende) Pixel werden invertiert
```

**gGREY**

OPL32

gGREY modus%

Ändert die Pen-Farbe.

```
modus%= 0 setzt die Vordergrund-Farbe auf schwarz
modus%= 1 setzt die Vordergrund-Farbe auf grau
modus%= 2 und alle anderen Werte setzen d.
      Vordergrund.farbe auf schwarz
```

gGREY konkurriert mit gCOLOR und kann bei einfachen Farbgebungen (schwarz und grau) herangezogen werden. Siehe auch DEFAULTWIN, gCREATE und gGREY/SIBO

**gGREY**

SIBO

gGREY modus%

Wählt die Ebene, auf die die nachfolgenden Befehle zeichnen oder schreiben.

modus%	Ebene
0	schwarz
1	grau
2	beide

Stellen Sie sich die schwarze Ebene als über der grauen Ebene liegend vor, da graue Pixel von schwarzen Pixeln verdeckt werden. Die graue Ebene muss am Anfang des Programms mit `DEFAULTWIN 1` für das Standardfenster eingeschaltet werden, wenn Sie sie benutzen wollen. Andere Fenster werden beim Erstellen mit einer grauen Ebene versehen, wenn sie die entsprechende Option setzen.

`gGREY` kann nicht auf Bitmaps angewandt werden, da diese nur eine Ebene haben.

Siehe auch `DEFAULTWIN` and `gCREATE`

### ***gHEIGHT***

*OPL32 – SIBO*

`hoehe%=gHEIGHT`

Die Variable enthält die Höhe der aktuellen Zeichenfläche.

### ***gIDENTITY***

*OPL32 – SIBO*

`id%=gIDENTITY`

Die Variable enthält die Kenn-Nummer der aktuellen Zeichenfläche (Fenster oder Bitmap)

### ***gINFO***

*SIBO*

`gINFO i%()`

Liefert verschiedene Informationen über das aktuelle Fenster/Bitmap sowie den Grafikcursor. Die Informationen werden im Array `i%()` abgelegt, das mindestens 32 Felder enthalten muss. Die Informationen beziehen sich auf den aktuellen Stand des aktuellen Fensters/Bitmaps.

Folgende Informationen werden zurückgeliefert:

<code>i%(1)</code>	niedrigster Zeichencode
<code>i%(2)</code>	höchster Zeichencode
<code>i%(3)</code>	Fontheöhe
<code>i%(4)</code>	Pixel unter der Basislinie des Fonts
<code>i%(5)</code>	Pixel über der Basislinie des Fonts
<code>i%(6)</code>	Breite des Zeichens "0"
<code>i%(7)</code>	Maximale Zeichenbreite
<code>i%(8)</code>	Flags des Fonts (s.u.)
<code>i%(9-17)</code>	Fontname
<code>i%(18)</code>	Grafikmodus
<code>i%(19)</code>	Textmodus
<code>i%(20)</code>	Schriftstil
<code>i%(21)</code>	Cursor (EIN=1/AUS)
<code>i%(22)</code>	ID des Fensters mit aktuellem Cursor
<code>i%(23)</code>	Cursorbreite
<code>i%(24)</code>	Cursorhöhe
<code>i%(25)</code>	Pixel des Cursor unter Font-Basislinie
<code>i%(26)</code>	X-Position des Cursor (im Fenster)
<code>i%(27)</code>	Y-Position des Cursor(im Fenster)
<code>i%(28)</code>	1 wenn Bitmap
<code>i%(29)</code>	Cursor-Effekte
<code>i%(30)</code>	<code>gGREY</code> -Modus
<code>i%(31)</code>	reserviert (interne ID im Windowserver)
<code>i%(32)</code>	reserviert

i%(8) enthält eine Kombination folgender Flags:

Wert:	Bedeutung:
1	Standard-ASCII-Font (32-126)
2	Font gemäß Codeseite 850 (128-255)
4	Font ist fett
8	Font ist kursiv
16	Font hat Serifen
32	Font ist nicht proportional
\$8000	Font ist erweitert gespeichert für schnelle Ausgabe

Verwenden Sie `PEEK$(ADDR(i%(9)))`, um den Fontnamen als String zu lesen.

Ist der Cursor eingeschaltet (i%(21)=1), ist er im Fenster mit der ID i%(22) sichtbar.

In i%(29) ist bit 0 (i%(29) AND 1) gesetzt, wenn der Cursor abgerundet ist, bit 1 (i%(29) AND 2) ist gesetzt, wenn der Cursor nicht blinkt und bit 3 (i%(29) AND 4) ist gesetzt, wenn der Cursor grau ist.

Ist der Cursor ausgeschaltet oder ein Textcursor (i%(21)=-1), sollten i%(23) bis i%(27) sowie i%(29) ignoriert werden.

### **gINFO32**

OPL32

`gINFO32 info&()`

Allgemeine Informationen über die aktuelle Zeichenfläche und den Cursor. Die Variable `info&()` muss 48 Felder aufnehmen, auch wenn nicht alle genutzt werden. Die Felder enthalten zumeist erst dann sinnvolle Werte, wenn entsprechende Konfigurationsbefehle vorangegangen sind, z.B. `gFONT`, `CURSOR` usw.

Wert und Bedeutung	
info&(1)	reserviert
info&(2)	reserviert
info&(3)	Höhe des Font
info&(4)	Font: Pixel unter der Basislinie
info&(5)	Font: Pixel über der Basislinie
info&(6)	Breite der Null
info&(7)	Maximale Zeichenweite
info&(8)	Flags für den verwendeten Font: Werte (auch kombiniert möglich) und Bedeutung, nur wenn vom Font-Designer in den Font impliziert! 1: Font benutzt Stand.ASCII Zeichen (32-126) 2: Code Page 1252 wird benutzt 4: Font ist fett 8: Font ist kursiv 16: Font mit Serifen 32: Monospace-Font
info&(9)	Font Uid von gFONT
info&(10-17)	unbenutzt
info&(18)	aktueller Grafik-Modus (gGMODE)
info&(19)	aktueller Text-Modus (gTMODE)
info&(20)	Wert von gSTYLE
info&(21)	Cursor Status (ON=1, OFF=0)
info&(22)	Kenn-Nummer des Fensters, in dem der Cursor sichtbar ist (-1 für Text-Cursor)
info&(23)	Cursor-Breite
info&(24)	Cursor-Höhe
info&(25)	Cursor Neigung
info&(26)	Cursor x-Position im Fenster
info&(27)	Cursor y-Position im Fenster
info&(28)	1 wenn die Zeichenfläche ein Bitmap ist
info&(29)	Cursor Effekte
info&(30)	Grafik-Modus des aktuellen Fensters

```

info&(31)  gCOLOR rot% Vordergrund
info&(32)  gCOLOR gruen% Vordergrund
info&(33)  gCOLOR blau% Vordergrund
info&(34)  gCOLOR rot% Hintergrund
info&(35)  gCOLOR gruen% Hintergrund
info&(36)  gCOLOR blau% Hintergrund

```

Siehe auch gINFO, gFONT, gCOLOR, gCREATE

### **gINVERT**

OPL32 – SIBO

```
gINVERT weite%,hoehe%
```

Invertiert ein sich mit den Abmaßen weite% und hoehe% nach rechts und unten ausdehnendes Rechteck, Startposition ist die aktuelle Cursorposition. Die vier Eckpunkte werden nicht mit bearbeitet!

### **GIPRINT**

OPL32 – SIBO

```

GIPRINT text$
GIPRINT text$,a%

```

Zeigt eine Nachricht text\$ für etwa zwei Sekunden am Bildschirm rechts unten. Die Position lässt sich auch mit a% grob gezielt festlegen:

```

a%= 0 oben links
a%= 1 unten links
a%= 2 oben rechts
a%= 3 unten rechts

```

Ist text\$ zu lang, wird ohne Fehlermeldung einfach gekappt.

Es kann immer nur eine Nachricht zugleich angezeigt werden. Die Nachricht kann vorzeitig entfernt werden, indem Sie GIPRINT "" aufrufen.

### **gLINEBY**

OPL32 – SIBO

```
gLINEBY dx%,dy%
```

Zeichnet eine Linie ausgehend von der aktuellen Position mit den relativen Werten dx% und dy%. Sind die Werte negativ, wird nach links bzw. nach oben gezeichnet, statt nach rechts bzw. unten. Achtung, der Endpunkt wird nicht gezeichnet! Ausnahme: ein einzelner Punkt wird gesetzt, wenn dx% und dy% Null sind. Die Cursorposition verschiebt sich auf den Endpunkt.

Siehe auch gLINETO, gPOLY

### **gLINETO**

OPL32 – SIBO

```
gLINETO x%,y%
```

Zeichnet eine Linie ausgehend von der aktuellen Position zu den absoluten Koordinaten x% und y%. Die Cursorposition verschiebt sich dabei auf den Endpunkt. Achtung, der Endpunkt wird nicht gezeichnet! Ausnahme: ein einzelner Punkt wird gesetzt, wenn x% und y% identisch mit den Werten der Cursorposition sind.

Siehe auch gLINEBY, gPOLY

## OPL32 – SIBO

**gLOADBIT**

```
id%=gLOADBIT(name$)
id%=gLOADBIT(name$,schreiben%)
id%=gLOADBIT(name$,schreiben%,index%)
```

Lädt ein Bitmap mit der Bezeichnung name\$. Dieses wird zur aktuellen Zeichenfläche, ist allerdings nicht sichtbar, jedoch unter Verwendung von id% mit anderen Befehlen manipulierbar. Der Cursor steht auf der Position 0,0. Bei dem Bitmap muss es sich um eine Grafik im EPOC-Format handeln (Multi Bitmap, MBM). Dieses Format wird durch gSAVEBIT erzeugt oder aus der Applikation SKIZZE exportiert. In beiden Fällen kann nur ein einzelnes Bitmap abgespeichert werden.

Soll die Grafik unverändert bleiben, setzt man den Parameter schreiben% auf Null, der Standardwert ist 1 (verändern und speichern des Bitmap sind erlaubt).

Extern erzeugte MBM-Grafikdateien können mehrere Einzelgrafiken enthalten. Sie werden mit Hilfe von index% angesprochen. Die erste Grafik hat den Index 0, das ist gleichzeitig der Standardwert.

Siehe auch gCLOSE

## OPL32

**gLOADFONT**

```
id%=gLOADFONT(fontname$)
```

Lädt einen benutzerdefinierten Font, der mit fontname\$ spezifiziert wird, und gibt die ID der Datei zurück (id%), mit deren Hilfe der Font wieder entladen werden kann (gUNLOADFONT id%).

Es können maximal 16 Fontdateien gleichzeitig geladen sein.

Die Aktivierung eines geladenen Fonts erfolgt wie üblich per gFONT font\_uid&. Die font\_uid& bzw. deren Konstante muss man kennen, sie ist interner Bestandteil der Fontdatei.

Beispiel:

```
id%=gLOADFONT("Musika")      REM Font laden
gFONT KMusikaFont&           REM Font benutzen
...
gUNLOADFONT id%              REM Font entladen
```

Beachten Sie, dass die eingebauten Fonts nicht geladen werden müssen.

Siehe auch gUNLOADFONT

## SIBO

**gLOADFONT**

```
fontID%=gLOADFONT(name$)
```

Lädt den benutzerdefinierten Font name\$. Die zurückgelieferte Font-Id-Nummer wird benutzt, um den Font mit gFONT anzuwählen. Wenn name\$ keine Dateierweiterung enthält wird .FON verwendet.

Weil gLOADFONT einen Dateizugriff benötigt und gFONT sehr schnell und effizient arbeitet, sollten Sie alle benötigten Fonts am Anfang des Programms laden um so eine möglichst hohe Ausführungsgeschwindigkeit zu erzielen.

Beachten Sie bitte, dass die eingebauten Fonts nicht geladen werden müssen.

Siehe auch gUNLOADFONT

**GLOBAL**

OPL32 – SIBO

GLOBAL variablenliste

Deklariert Variablen, die in der aktuellen Prozedur verwendet werden sollen (genau wie LOCAL). Die Variablen stehen auch in allen gerufenen Prozeduren und deren Unterprozeduren zur Verfügung (im Gegensatz zu LOCAL).

Es stehen 4 Variablentypen zur Verfügung:

Variablennamen ohne Suffix sind Fließkomma-Variablen (Floating point) (z.B.: preis, x,y)

Variablennamen mit dem Suffix % sind Ganzzahl-Variablen (Integer) (z.B.: preis%,x%,y%)

Variablennamen mit dem Suffix & sind große Ganzzahl-Variablen (Long Integer) (z.B.: preis&,x&,y&)

Variablennamen mit dem Suffix "\$" sind Text-Variablen (String) (z.B.: artikel\$(20),x\$(10),y\$(5)). Die in Klammern angegebene Zahl definiert die Länge des Strings.

Arrays werden durch in Klammern direkt nach dem Variablennamen angegebene Zahlen definiert (z.B.: preis(5), preis%(5), preis&(5)). Die Zahl benennt die Anzahl der Array-Elemente. Bei der Definition von String-Arrays müssen 2 durch Komma getrennte Zahlen angegeben werden, die erste definiert die Anzahl der Elemente, die zweite die Anzahl der Zeichen pro String, z.B.: artikel\$(5,10) = 5 Elemente mit je 10 Zeichen.

**OPL32:** Variablennamen können aus beliebigen Kombinationen von bis zu 32 Buchstaben, Zahlen und dem Unterstrich bestehen. Sie müssen mit einem Buchstaben beginnen.

**SIBO:** Variablennamen können aus beliebigen Kombinationen von bis zu 8 Buchstaben und Zahlen bestehen. Sie müssen mit einem Buchstaben beginnen.

Die angegebene Länge schließt das Suffix (%, \$, &) ein, jedoch nicht die Array-Größe oder Stringlänge in Klammern.

Es können mehrere GLOBAL bzw. LOCAL Befehle benutzt werden. Die Befehle müssen jedoch in getrennten Zeilen, direkt nach dem Prozedurnamen stehen.

Siehe auch LOCAL

**gMOVE**

OPL32 – SIBO

gMOVE dx%,dy%

Verschiebt die aktuelle Cursor-Position um dx% nach rechts bzw. dy% nach unten. Negative Werte verschieben in die entgegengesetzte Richtung.

Siehe auch gAT

**gORDER**

OPL32 – SIBO

gORDER id%,position%

Geöffnete Fenster liegen quasi übereinander, ihre Position kann mit gORDER geändert werden. Der Befehl verschiebt das geöffnete Fenster Nummer id% auf die genannte Position position%.

position%= 1 ist das Fenster in vorderster Position, alle anderen liegen dahinter. Wählt man bei position% einen Wert, der größer als die Anzahl der geöffneten Fenster ist, wird das Fenster auf die letztmögliche Position (in den Hintergrund) gestellt. Bei Befehlsausführung wird der Bildschirm neu gezeichnet. Das geht nicht mit Bitmaps!

Siehe auch gRANK

**gORIGINX***OPL32 – SIBO*`x%=gORIGINX`

x% ist die Entfernung zwischen linkem Bildschirmrand und der linken Kante des aktuellen Fensters. Geht nicht mit Bitmaps!

**gORIGINY***OPL32 – SIBO*`y%=gORIGINY`

y% ist die Entfernung zwischen oberem Bildschirmrand und der oberen Kante des aktuellen Fensters. Geht nicht mit Bitmaps!

**GOTO***OPL32 – SIBO*

```
GOTO marke oder
GOTO marke::
...
...
marke::
```

Springt zur Zeile nach der Markierung marke:: und setzt die Ausführung des Programms dort fort.

Die Marke ...

- muss in der aktuellen Prozedur liegen.
- muss mit einem Buchstaben beginnen und mit einem doppelten Doppelpunkt enden (kann beim GOTO-Befehl entfallen).
- darf 32 Zeichen (OPL32) bzw. 8 Zeichen (SIBO) lang sein (Doppelpunkte werden dabei nicht mitgezählt).

Der Sprung kann auch rückwärts gerichtet sein.

```
...
marke::
n%=n%+1
IF n%<100
    GOTO marke::
ENDIF
...
```

Die Verwendung von GOTO wird nicht empfohlen, da leicht unübersichtliche Programme entstehen können ("Spaghetticode").

**GOTOMARK***OPL32*`GOTOMARK n%`

Macht den Datensatz zum aktuellen, der vorher mit dem Befehl `BOOKMARK n%` markiert wurde.

Siehe auch `BOOKMARK`, `KILLMARK`



**gPATT**

```
gPATT id%,weite%,hoehe%,modus%
```

Füllt ein Rechteck mit den Abmaßen `weite%*hoehe%` mit einem Muster, das in einer Zeichenfläche mit der Kenn-Nummer `id%` bereitgehalten wird. Ist das Muster kleiner als die zu füllende Fläche, wird es solange wiederholt eingefügt, bis die Fläche gefüllt ist. Ist `id% -1`, wird ein vordefiniertes Muster eingefügt.

Mit `modus%` wird festgelegt, wie das Muster eingefügt wird:

```
modus%= 0 kopiert im Muster gesetzte Pixel und setzt sie
           im Ziel
modus%= 1 kopiert im Muster gesetzte Pixel und löscht
           gesetzte Pixel im Ziel
modus%= 2 kopiert im Muster gesetzte Pixel und
           invertiert die Pixel im Ziel
modus%= 3 kopiert alle Pixel des Musters (einschl. der
           weißen) und ersetzt alle Pixel im Ziel
```

**gPEEKLINE**

```
gPEEKLINE id%,x%,y%,d%(),pixel%
gPEEKLINE id%,x%,y%,d%(),pixel%,modus%
```

Tastet eine horizontale Linie der Zeichenfläche `id%` pixelweise ab und legt das Resultat im Array `d%()` ab. Wenn Sie `id%=0` setzen lesen Sie vom gesamten Bildschirm, nicht von einer bestimmten Zeichenebene.

Die Linie beginnt bei `x%,y%` und hat eine Länge von `pixel%`. Das Ergebnis für die ersten 16 Pixel liegen dann in `d%(1)`, wobei sich das erste Pixel im niederwertigsten Bit befindet.

Üblicherweise bekommen die Bits in `d%()` einen Eintrag, wenn ein Pixel schwarz ist. Durch den Parameter `modus%` wird beeinflusst, ob eventuell andere Farben zu einem Eintrag führen; der Standard-Modus ist -1:

```
modus%= -1 (schwarz/weiß-Modus), schwarz setzt Pixel
modus%= 0 (schwarz/weiß-Modus), weiß setzt Pixel
modus%= 1 (4-Farb-Modus), weiß setzt Pixel
modus%= 2 (16-Farb-Modus), weiß setzt Pixel
```

Achtung! Im 4-Farb-Modus bzw. 16-Farb-Modus werden 2 bzw. 4 Bit je Pixel erfasst, die zugehörige Feldanzahl muss also 2mal bzw. 4mal so groß sein als bei der Erfassung von schwarz/weiß Flächen. Die Anzahl der erforderlichen Felder für den schwarz/weiß-Modus errechnet sich aus:  $\text{INT}((\text{pixel\%}+15)/16)$ . Sie ist mit zwei bzw. vier zu multiplizieren, wenn 4- bzw. 16-Farbmodus verwendet werden.

Beispiel 1:

Im 4-Farb-Modus mit gesetzten Farben von

```
gCOLOR 16,16,16
```

würde ein Pixel als Ergebnis 0001 (binär) liefern; bei

```
gCOLOR 80,80,80
```

wäre das 0101 binär.

Beispiel 2:

Die abgetastete Pixelzeile sei schwarz, der Farb-Modus schwarz/weiß. Der Befehl `gPEEKLINE id%,x%,y%,d%(),pixel%,-1` liefert dann in `d%(1)` den Wert -1. Das bedeutet, dass alle 16 Bits der Inte-

gerzahl gesetzt sind, binär ist das: 1111 1111 1111 1111. Die Auswertung des Ergebnisses muss also immer auch das 16. Bit berücksichtigen!

### **gPEEKLINE**

*SIBO*

`gPEEKLINE id%,x%,y%,d%(),l%`

Liest eine horizontale Linie aus der schwarzen Ebene der Zeichenebene `id%` mit Länge `l%` von Position `x%,y%` aus. Die ersten 16 Pixel werden in `d%(1)` gelesen, das erste Pixel steht im niederwertigsten Bit. Das Array `d%()` muss lang genug sein um die Daten aufzunehmen. Die Anzahl der benötigten Elemente lässt sich mit  $((l\%+15)/16)$  berechnen (ganzzahlige Division).

Wenn Sie `id%=0` setzen lesen Sie vom gesamten Bildschirm, nicht von einer bestimmten Zeichenebene. Wenn Sie `$8000` zu `id%` addieren lesen Sie die graue Ebene aus.

### **gPOLY**

*OPL32 – SIBO*

`gPOLY a%()`

Zeichnet eine Folge von Linien, wie mit `gLINEBY` und `gMOVE`. Das Array `a%()` enthält folgende Daten:

```
a%(1) Startposition x.
a%(2) Startposition y.
a%(3) Anzahl der folgenden Versatzpaare.
a%(4) dx1%
a%(5) dy1%
a%(6) dx2%
usw.
```

Jedes Zahlenpaar `dx%` und `dy%` definiert eine zu zeichnende Linie oder ein Verschieben der aktuellen Position. Um eine Linie zu zeichnen geben Sie in `dy%` die Anzahl der Pixel nach unten an und in `dx%` die Anzahl der Pixel nach rechts multipliziert mit 2. Um nur die aktuelle Position zu ändern, addieren Sie zu `dx%` noch 1. Bei ungerade `dx%` wird also nur verschoben und nicht gezeichnet. Negative Angaben in `dx%` oder `dy%` bewegen oder zeichnen nach links bzw. oben.

`gPOLY` arbeitet weitaus schneller als Sequenzen von `gLINETO` oder `gMOVE` etc.

Das folgende Beispiel zeichnet drei 50 Pixel lange horizontale Linien beginnend an den Positionen 20,10;20,30 und 20,50.

```
a%(1)=20:a%(2)=10 REM 20,10
a%(3)=5 REM 5 Operationen
a%(4)=50*2:a%(5)=0
REM 20 Pixel nach unten
a%(6)=0*2+1:a%(7)=20
a%(8)=-50*2:a%(9)=0
a%(10)=0*2+1:a%(11)=20
a%(12)=50*2:a%(13)=0
gPOLY a%()
```

### **gPRINT**

*OPL32 – SIBO*

`gPRINT ausdrucksliste`

"Druckt" die Liste der Ausdrücke an der aktuellen Cursorposition der aktuellen Zeichenfläche. Der Cursor markiert dabei die Position der Schrift-Grundlinie. Die Darstellung der Ausdrücke erfolgt wie bei `PRINT`, jedoch wird der Cursor nicht weiterbewegt. Daher hat das Semikolon keine praktische Auswirkung, erzeugt aber auch keinen Fehler, wenn man es doch verwendet.

Siehe auch `gPRINTB`, `gPRINTCLIP`, `gTWIDTH`, `gXPRINT`, `gTMODE`

## OPL32 – SIBO

**gPRINTB**

```
gPRINTB text$,breite%
gPRINTB text$,breite %,a%
gPRINTB text$,breite %,a%,do%
gPRINTB text$,breite %,a%,do%,du%
gPRINTB text$,breite %,a%,do%,du%,r%
```

Zeigt den Text text\$ in einer vorher gelöschten Box mit der Breite breite%. Der Cursor markiert dabei horizontal den linken Rand der Box und vertikal die Textgrundlinie. Der Parameter a% steht für die Ausrichtung des Textes:

```
a%= 1 rechtsbündig
a%= 2 linksbündig (Standardwert)
a%= 3 zentriert
```

Die Parameter do% und du% stehen für den Abstand der Schrift in Pixel vom oberen bzw. unteren Rand der Box, der Standardwert ist Null.

Mit dem Parameter r% lässt sich ein zusätzlicher Rand in Pixel definieren. Der Standardwert ist Null. Die Auswirkung hängt von der Ausrichtung a% ab. Der Rand wird links bzw. rechts eingefügt, wenn die Ausrichtung links- bzw. rechtsbündig ist. Bei Zentrierung des Textes ergibt ein positiver r%-Wert einen zusätzlichen Offset auf der linken Seite des Textes, ein Offset auf der rechten Seite wird mit negativem r%-Wert erreicht.

Siehe auch gPRINT, gPRINTCLIP, gTWIDTH, gXPRINT

## OPL32 – SIBO

**gPRINTCLIP**

```
n%=gPRINTCLIP(text$,breite%)
```

Zeigt einen Text text\$ an der aktuellen Position in der aktuellen Zeichenfläche, der zeichengenau in die angegebene Breite breite% (in Pixel) passt. In n% wird die Anzahl der angezeigten Zeichen zurückgegeben.

Siehe auch gPRINT, gPRINTB, gTWIDTH, gXPRINT, gTMODE

## OPL32 – SIBO

**gRANK**

```
position%=gRANK
```

Liefert in position% die gegenwärtige Position des aktuellen Fensters zurück. Geht nicht für Bitmaps! position% liefert die Werte 1 – 8 unter SIBO und 1 – 64 unter OPL32.

Siehe auch gORDER

## OPL32 – SIBO

**gSAVEBIT**

```
gSAVEBIT name$
gSAVEBIT name$,weite%,hoehe%
```

Speichert die aktuelle Zeichenfläche oder einen Teil von ihr als Bitmap-Datei unter dem Namen name\$. Wird in weite% und hoehe% eine Breite und Höhe angegeben, wird nur das Rechteck mit der angegebenen Breite und Höhe, ausgehend von der aktuellen Position, gespeichert.

Wenn keine Dateiendung angegeben wird, wird sie unter OPL32 nicht automatisch ergänzt, während unter SIBO ".PIC" angefügt wird.

SIBO: Wird ein Fenster mit schwarzer und grauer Ebene gespeichert, werden nacheinander zwei Bitmaps in der Datei abgelegt. Zuerst die schwarze, dann die graue Ebene.

## OPL32 – SIBO

**gSCROLL**

```
gSCROLL dx%,dy%
gSCROLL dx%,dy%,x%,y%,breite%,hoehe%
```

Verschiebt Pixel innerhalb der aktuellen Zeichenfläche um den Betrag dx% nach rechts und um dy% nach unten, negative Werte kehren die jeweilige Richtung um. Soll nur ein Teilbereich verschoben werden, gibt man dessen Ursprung mit x%,y% und dessen Ausbreitung nach rechts und unten mit breite%, hoehe% an.

Die ursprünglich belegte Fläche wird nach dem Scrollen gelöscht, die aktuelle Cursorposition nicht verändert.

## OPL32

**gSETPENWIDTH**

```
gSETPENWIDTH breite%
```

Setzt die Breite des Zeichenstiftes in Pixel in der aktuellen Zeichenfläche auf breite%.

## OPL32 – SIBO

**gSETWIN**

```
gSETWIN x%,y%
gSETWIN x%,y%,breite%,hoehe%
```

Verändert die Position des aktuellen Fensters (linke obere Ecke) auf die Koordinaten x%,y%. Bei Bedarf lässt sich mit breite% und hoehe% die Ausdehnung rekonfigurieren. Geht nicht mit Bitmaps!

Wird gSETWIN auf das Basisfenster angewandt, ist auch der SCREEN-Befehl anzuwenden, siehe dort!

## OPL32 – SIBO

**gSTYLE**

```
gSTYLE stil%
```

Setzt die Text-Darstellung der aktuellen Zeichenfläche.

Wirkt sich aus bei gPRINT, gPRINTB und gPRINTCLIP.

```
stil%= 0 normal
stil%= 1 fett
stil%= 2 unterstrichen
stil%= 4 invers
stil%= 8 doppelte Höhe
stil%= 16 Monospace-Schrift
stil%= 32 kursiv
```

Die Werte lassen sich durch Addition kombinieren: z.B.: 1+16 für fett/kursiv. Befehle die im Textmodus arbeiten, wie z.B.: PRINT werden nicht beeinflusst.

## OPL32 – SIBO

**gTMODE**

```
gTMODE modus%
```

Bestimmt durch den Wert von modus%, wie die Zeichen durch gPRINT und gPRINTCLIP usw. in der aktuellen Zeichenfläche dargestellt werden.

```
modus%= 0 Pixel setzen
modus%= 1 Pixel löschen
modus%= 2 Pixel invertieren
modus%= 3 Pixel ersetzen
Standardwert ist Null.
```

OPL32 – SIBO

**gTWIDTH**

breite%=gTWIDTH(text\$)

Gibt die Pixelbreite des Textes text\$ in breite% zurück.

Siehe auch gPRINT, gPRINTB, gPRINTCLIP, gXPRINT

OPL32 – SIBO

**gUNLOADFONT**

gUNLOADFONT id%

Entlädt den vorher mit id%= gLOADFONT(fontname\$) geladenen Extra-Font. Es ergibt sich eine Fehlermeldung, wenn vorher kein Font geladen wurde. Interne Fonts müssen weder geladen noch entladen werden.

Siehe auch gLOADFONT

OPL32 – SIBO

**gUPDATE**

gUPDATE  
gUPDATE ON  
gUPDATE OFF

Die Abarbeitung einer Reihe von aufeinanderfolgenden Grafikbefehlen können zu einem verzögerten Bildschirmaufbau und zu Programmverzögerungen beitragen, da nach jedem dieser Befehle die Bildschirmanzeige erneuert wird. Diese sofortige Erneuerung kann man durch gUPDATE OFF stark reduzieren.

Die in Sachen Geschwindigkeit optimale Vorgehensweise bei Abarbeitung eines Grafikblocks wäre eine Reduzierung des Updates mit gUPDATE OFF, das Ablaufen lassen des Grafikbefehlsblockes und ein nachfolgendes Erzwingen des Updates mit gUPDATE. Ist der massive Anteil an Grafikbefehlen abgearbeitet, kehrt man schließlich mit gUPDATE ON wieder in den Standard-Modus zurück

Beachten Sie bitte, dass bei Benutzung von gUPDATE OFF auftretende Fehler an einer falschen Stelle gemeldet werden können. Der Befehl sollte daher erst am Ende der Programmentwicklung zur Optimierung eingesetzt werden.

OPL32 – SIBO

**gUSE**

gUSE id%

Benutzt nach dem Befehl die Zeichenfläche id% als aktive Zeichenfläche. Diese kommt dadurch aber nicht automatisch in den Vordergrund! Dazu ist gORDER zu bemühen!

OPL32 – SIBO

**gVISIBLE**

gVISIBLE ON  
gVISIBLE OFF

Durch gVISIBLE OFF verschwindet das aktuelle Fenster vom Bildschirm, gVISIBLE ON lässt es wieder erscheinen. Wird der Befehl auf ein Bitmap angewendet wird ein Fehler ausgegeben!

OPL32 – SIBO

**gWIDTH**

weite%=gWIDTH

Gibt die Breite des aktuellen Zeichenfensters an die Variable weite% zurück.

**gX**

x%=gX

Gibt in x% die x-Position des Cursors in der aktuellen Zeichenfläche zurück.

**gXBORDER**

gXBORDER typ%,flag%  
gXBORDER typ%,flag%,breite%,hoehe%

Zeichnet einen rechteckigen Rahmen innerhalb der aktuellen Zeichenfläche. Ohne Angabe von breite% und hoehe% wird es sozusagen am Innenrand entlang gezogen. Bei typ% ist für Geräte mit EPOC32 die 2 verbindlich, die 1 steht für die 3er Serie des Psion. Der Parameter flag% bietet Gestaltungsmöglichkeiten, der den Bereich Button-ähnlich aussehen lässt:

```
flag%= $00 kein Rahmen
flag%= $01 einfacher schwarzer Rahmen
flag%= $42 leicht eingesunkener Rahmen
flag%= $44 tief eingesunkener Rahmen
flag%= $54 tief eingesunkener Rahmen mit Outline
flag%= $82 leicht erhabener Rahmen
flag%= $84 stark erhabener Rahmen
flag%= $94 stark erhabener Rahmen mit Outline
flag%= $22 linker vertikaler Rahmenteil gelöscht
flag%= $2A oberer horizontaler Rahmenteil gelöscht
```

Das englischsprachige Originalhandbuch sagt, dass durch Addieren der folgenden flag%-Werte der Einfluss noch weiter geht, das konnte im Experiment auf verschiedenen Geräten allerdings nicht nachvollzogen werden:

```
$100 lässt eine ein Pixel breite Lücke um den Rahmen
$200 ergibt etwas rundere Ecken
$400 nimmt einen Pixel aus den Ecken ("Mini-Rundung")
$200 und $400 lassen sich nicht kombinieren, $200 hat Vorrang.
```

Siehe auch gBORDER

**gXPRINT**

gXPRINT string\$,flag%

Zeigt string\$ an der aktuellen Bildschirmposition optional invertiert oder unterstrichen an. Es werden der aktuelle Font und der aktuelle Schriftstil benutzt. gXPRINT überschreibt immer alle im Bereich des Textes liegenden Pixel (Textmodus 3). Einige Einstellungen überschneiden sich mit gTMODE und ergeben wenig gebräuchliche Ansichten.

```
flag%= 0 normal wie gPRINT
flag%= 1 invertiert
flag%= 2 invertiert. unter Auslassung der Eckpunkte
flag%= 3 invertiert, dünn unterstrichen, ohne Abstand
flag%= 4 invertiert, dünn unterstrichen, ohne Abstand,
        unter Auslassung der Eckpunkte
flag%= 5 unterstrichen noch unter der Zeichen-Unterlänge
flag%= 6 dünn unterstrichen direkt am Ende der Zeichen-
        Unterlänge
```

Die Ausgabe einer Liste mit Ausdrücken verschiedener Variablentypen wird von gXPRINT nicht unterstützt, es kann nur ein String angezeigt werden. Es werden auch endständige oder alleinstehende Leerzeichen unterstrichen.

Für Textzeilen, die nur durch einen Pixel getrennt sind, erhält die Option "dünn" die Zeilentrennung aufrecht.

**gY**

OPL32 – SIBO

`y%=gY`

Gibt in y% die y-Position des Cursors in der aktuellen Zeichenfläche zurück.

**HEX\$**

OPL32 – SIBO

`string$=HEX$(x&)`

Liefert einen String, der die hexadezimale Darstellung der in x& übergebenen Zahl beinhaltet. Beispiel: HEX\$(255) ergibt "FF".

Anmerkungen: Um hexadezimale Konstanten einzugeben, setzen Sie einfach das Zeichen \$ vor die Konstante, z.B.: "\$FF" für 255. Für Long-Integer Werte in hexadezimaler Darstellung verwenden Sie das Zeichen &, also "&FFFFFF" für 1048575.

Hexadezimal Zahlen werden wie folgt gezählt: 0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15 16 17 18 19 1A 1B etc.. A steht dabei für 10, B für 11 etc. Wie bei Zehnerübergang im dezimalen Zahlenformat hat die nächst höhere Stelle eine höhere Wertigkeit. 10 dezimal bedeutet  $1 \cdot 10 + 0 \cdot 1 = 10$ ; 1A hexadezimal bedeutet dezimal gerechnet:  $1 \cdot 16 + 10 \cdot 1 = 26$ . Jede weitere Stelle hat also eine um Faktor 16 höhere Wertigkeit.

**HOURL**

OPL32 – SIBO

`h%=HOURL`

Liefert die Stunden der aktuellen Uhrzeit als Zahl zwischen 0 und 23.

**IABS**

OPL32 – SIBO

`y&=IABS(x&)`  
`y%=IABS(x%)`

Liefert den Absolutwert der Integerzahl x& bzw. x% (Long- oder Normal-Integer-Zahl) wieder als Integer zurück.

`i&=IABS(x&)`

Liefert den Absolutwert der Zahl x&, d.h. den Wert ohne Vorzeichen (IABS(-10)=10).

Siehe auch ABS

**ICON**

OPL32

`ICON mbm$`

Der Befehl darf nur zwischen APP und ENDA benutzt werden.

Hinter ICON wird der Name der Bitmap-Datei benannt, die als Icon für eine OPL-Applikation verwendet werden soll. Die Bitmap-Datei muss im EPOC-Format (\*.mbm, MBM, Multi-Bitmap-Datei) vorliegen.

Wird kein Dateiname geliefert, werden Standard-Icons verwendet (das "Fragezeichen").

Es ist sinnvoll, für jede Zoom-Einstellung eine Bitmap-Paar, das aus dem eigentlichen Bitmap und der Maske besteht, zu liefern. Die dafür erforderlichen drei quadratischen Abbildungen sollen die Kantenlängen 24, 32 und 48 besitzen. Diese Abbildungen können auch in einer einzigen MBM-Datei untergebracht werden. Benutzt man für jedes Bitmap eine einzelne Datei, wird jede von ihnen mit dem ICON-Befehl eingebunden. Das Betriebssystem sucht sich später das zur Zoom-Einstellung passende Bitmap automatisch aus.

Wird nur ein Bitmap geliefert, vergrößert oder verkleinert das Betriebssystem die Darstellung passend, die optische Darstellung ist damit aber nicht optimal.

Beispiel:

```
APP ...
  ICON "icon24.mbm"
  ICON "mask24.mbm"
  ICON "icon32.mbm"
  ICON "mask32.mbm"
  ICON "icon48.mbm"
  ICON "mask48.mbm"
ENDA
```

Siehe auch: Kapitel "Applikationen"

## ICON

SIBO

ICON name\$

Definiert den Namen der Bitmap-Datei, in der das Symbol für die Applikation gespeichert ist. Der Befehl darf nur zwischen APP und ENDA benutzt werden.

## IF...ENDIF

OPL32 – SIBO

```
IF bedingung_1
...
ELSEIF bedingung_2
...
ELSEIF bedingung_n
...
ELSE
...
ENDIF
```

IF leitet eine Verzweigung ein. Die Verzweigungen enden mit dem abschließenden ENDIF. IF und ENDIF sind Pflichtbestandteile, ELSEIF (mehrfach verwendbar) und ELSE sind optional.

Bei einer einfachen IF...ENDIF-Konstruktion wird der Programmteil zwischen IF und ENDIF ausgeführt, wenn die hinter IF genannte Bedingung erfüllt (WAHR) ist.

Für eine IF...ELSE...ENDIF-Konstruktion gilt: Wenn bedingung\_1 WAHR ist, wird der Programmteil zwischen IF und ELSE ausgeführt, ansonsten der zwischen ELSE und ENDIF.

Mit einem zusätzlichen ELSEIF besteht eine zusätzliche Auswahlmöglichkeit. In diesem Falle wird der Programmabschnitt zwischen ELSEIF und dem nächsten ELSEIF oder ELSE abgearbeitet. Gibt es kein ELSE, kann das letzte Programmstückchen auch zwischen ELSEIF und ENDIF liegen.

IF...ELSEIF...ELSE...ENDIF-Verzweigungen dürfen bis zu acht Mal ineinander verschachtelt sein.

Beispiel verschachtelte Verzweigung:

```
IF bedingung_1
  IF bedingung_2
    ...
  ELSE
    IF bedingung_3
      ...
    ENDIF
  ENDIF
ELSE
  ...
ENDIF
```



**INCLUDE***F - OPL32*

```
INCLUDE datei$
```

Bezieht die Datei datei\$ in ein Programm mit ein, "inkludiert" sie. Diese Datei kann enthalten:

- CONST-Definitionen oder
- Prototypen von OPX-Prozeduren oder
- Prototypen von Modulen/Prozeduren

Die inkludierte Datei darf nicht die Prozeduren selber enthalten. Prozedur- und OPX-Prozedur-Prototypen gestatten es dem Übersetzer, Parameter gleich zu prüfen (Vermeidung von Laufzeit-Fehlern) und numerische Parameter (die nicht per Referenz übergeben werden) typgerecht anzupassen.

Eine Datei zu inkludieren ist logisch gleichwertig damit, den Inhalt dieser Datei direkt an die Stelle zu schreiben, an der der INCLUDE-Befehl steht.

Der Dateiname darf einen Pfad enthalten. Wenn ein Pfad angegeben wird, sucht OPL nur in dem benannten Ordner nach dem Dateinamen. Der Standardpfad für INCLUDE ist \System\Op\l. Wird also kein Pfad angegeben, sucht OPL zuerst im aktuellen Verzeichnis und dann im Standardpfad \System\Op\l aller Laufwerke, beginnend bei Y: und in Richtung A:, danach in Z:

Siehe auch CONST, EXTERNAL

**INPUT***OPL32 – SIBO*

```
INPUT var          (var= beliebige numerische Variable)
INPUT log.feld     (log= logischer Bezeichner f. offene DB)
```

Wartet darauf, dass ein Wert von der Tastatur eingegeben wird und übergibt den Wert an die angegebene Variable oder das angegebene Feld des Datensatzes eine Datei.

Der eingegebene Wert kann mit den üblichen Editierfunktionen bearbeitet werden. Mit ENTER wird die Eingabe abgeschlossen.

Wurde ein Wert eingegeben, der nicht zum Typ der Variablen passt wird ein "?" angezeigt, der Wert kann neu eingegeben werden. INPUT wird normalerweise in Verbindung mit PRINT eingesetzt:

```
PROC exch:
LOCAL pds,rate
DO
  PRINT "Englische Pfund ?",
  INPUT pds
  PRINT "Kurs (Euro) ?",
  INPUT rate
  PRINT "=",pds*rate,"Euro"
  GET
UNTIL 0
ENDP
```

Beachten Sie bitte die Kommas am Ende der PRINT-Anweisung, damit die Eingabe in der selben Zeile wie die entsprechende Aufforderung erfolgt.

INPUT in Kombination mit TRAP:

```
TRAP INPUT
```

Wenn Sie TRAP verwenden, liefert INPUT nur einen Fehler zurück, gibt die Kontrolle aber zur nächsten Zeile des Programms weiter. Sie können also den Fehler selbst behandeln. Fehler treten auf, wenn eine falsche Eingabe (z.B. String statt Zahl) gemacht wurde oder im leeren Eingabefeld ESC gedrückt wurde (Fehler -114, "ESC gedrückt"). Die Fehler werden von ERR zurückgeliefert.

Siehe auch EDIT

**INSERT**

OPL32

INSERT

Leitet das Einfügen eines neuen Datensatzes in die geöffnete aktuelle Ansicht ein. Das Abschlusskommando ist PUT. Zwischen INSERT und PUT werden die Daten dem Datensatz zugeordnet. Sollen die Daten nicht mit PUT in die DB geschrieben werden, wird der Vorgang mit CANCEL abgebrochen. Werden keine Daten zugeordnet, entsteht ein leerer Datensatz.

```
INSERT
  A.name$= "Anette"
PUT
```

Siehe auch CANCEL, MODIFY, PUT

**INT**

OPL32 – SIBO

y&amp;=INT(x)

Liefert den Integeranteil einer Fließkomma-Zahl als Long-Integer-Zahl zurück. Die Zahlen werden nicht mathematisch, sondern immer in Richtung der Null abgerundet.

```
INT(-2.3) --> -2
INT(-2.7) --> -2
INT(2.3)  --> 2
INT(2.7)  --> 2
```

**INTF**

OPL32 – SIBO

y=INTF(x)

Funktioniert wie INT, jedoch wird als Ergebnis eine Fließkommazahl zurückgeliefert. Diese Berechnung kann verwendet werden, wenn das Ergebnis den Geltungsbereich der Integerzahlen überschreitet.

**INTRANS**

OPL32

i&amp;= INTRANS

INTRANS prüft, ob gerade eine Transaktion mit der aktuellen Ansicht läuft. Ist i& = -1, findet eine Transaktion statt, bei i&= 0 nicht.

Siehe auch BEGINTRANS, COMMITTRANS, ROLLBACK

**I/O Befehle**

F - OPL32 – SIBO

r%=IOA(h%,f%,status%,a1,a2)

Der Gerätetreiber mit Handle h% führt asynchron die Funktion f% mit den beiden Argumenten a1 und a2 durch. Das Argument status% wird vom Gerätetreiber gesetzt. Für jeden IOA-Aufruf muss ein IOWAIT-Befehl ausgeführt werden.

r%=IOC(h%,f%,status%,a1,a2)

Startet eine I/O-Anforderung, deren Beendigung garantiert ist. Wie IOA, liefert jedoch immer 0 zurück.

r%=IOCANCEL(h%)

Bricht eine offene I/O-Anforderung ab (IOC/IOA).

`r%=IOCLOSE(h%)`

Schließt die Datei mit Handle `h%`.

`r%=IOOPEN(h%,name$,mode%)`

Erstellt oder öffnet eine Datei mit Namen `name$`. Der Handle `h%` wird für den Zugriff auf die Datei mit anderen I/O-Befehlen zurückgeliefert. `mode%` gibt den Modus für das Öffnen/Erstellen an. Um eine garantiert noch nicht existierende Datei zu erstellen verwenden Sie `IOOPEN(h%,ADDR(name$),mode%)`. In `name$` muss dazu nur ein Pfad stehen. Der selbst generierte Dateiname wird anschließend nach `name$` zurückgeliefert.

`r%=IOREAD(h%,addr&,maxLen%)` OPL32

`r%=IOREAD(h%,addr%,maxLen%)` SIBO

Liest aus einer Datei mit Handle `h%`. `addr&` (bzw.`addr%`) ist die Adresse eines Puffers, der lang genug sein muss, um `maxLen%` Bytes aufzunehmen. Die Funktion gibt in `r%` die tatsächlich gelesenen Bytes zurück. Ist `r%` negativ, ist ein Fehler der Nummer `r%` aufgetreten.

`r%=IOSEEK(h%,mode%,off&)`

Positioniert innerhalb einer Datei über die Adresse `off&`. `mode%` gibt an wie `off&` benutzt werden soll.

IOSIGNAL

Signalisiert die Beendigung einer I/O-Funktion.

`r%=IOW(h%,func%,a1,a2)`

Der Gerätetreiber mit handle `h%` führt synchron die Funktion `func%` mit den Argumenten `a1` und `a2` aus.

IOWAIT

Wartet auf das Signal für die Beendigung einer asynchronen I/O-Funktion.

IOWAITSTAT `stat%`

Wartet auf die Beendigung eine bestimmten I/O-Funktion (mit Statusvariable `stat%`).

IOWAITSTAT32 `stat&` (OPL32)

Benutzt eine 32-Bit-Statusvariable. `IOWAITSTAT32` sollte nur aufgerufen werden, wenn auf die Beendigung einer Prozedur gewartet wird, die den Status als 32-Bit-Wort zurückliefert, z.B. bei einer entsprechenden asynchronen OPX-Prozedur.

`r%=IOWRITE(h%,addr&,length%)` OPL32

`r%=IOWRITE(h%,addr%,length%)` SIBO

Schreibt `length%` Bytes eines Puffer mit Adresse `addr%` in die Datei mit handle `h%`.

IOYIELD

Stellt sicher, dass asynchrone Funktionen die Gelegenheit bekommen abzulaufen.

## KEY

`taste%=KEY`

OPL32 – SIBO

Arbeitet in etwa wie `GET`, wartet aber nicht auf einen Tastendruck, sondern bezieht den Wert aus dem Taster-Speicher von der zuletzt gedrückten Taste. Wurde keine gedrückt, ist das Ergebnis Null.

Siehe auch `GET`

**KEY\$***OPL32 – SIBO*`taste$=KEY$`

Arbeitet in etwa wie `GET$`, wartet aber nicht auf einen Tastendruck, sondern bezieht den Wert aus dem Tastaturspeicher von der zuletzt gedrückten Taste. Wurde keine gedrückt, ist das Ergebnis ein Leerstring.

Siehe auch `GET$`

**KEYA***F - OPL32 – SIBO*`err%=KEYA(stat%,key%(1))`

Diese Funktion liest asynchron die Tastatur aus. Die Funktion wird über `KEYC` abgebrochen

**KEYC***F - OPL32 – SIBO*`err%=KEYC(stat%)`

Bricht die mit `KEYA` gestartete Anforderung wieder ab.

**KILLMARK***OPL32*`KILLMARK n%`

Entfernt die Markierung eines Datensatzes, die mit `BOOKMARK n%` gesetzt wurde.

Siehe auch `BOOKMARK`, `GOTOMARK`

**KMOD***OPL32 – SIBO*`mod%=KMOD`

Erfasst den Status der Modifiziertasten, wenn Tasten gedrückt werden (`GET ..`)

```
mod%= 2   keine Modifizierer-Taste gedrückt
mod%= 2   Shift gedrückt
mod%= 4   Strg gedrückt
mod%= 16  Caps Lock an
mod%= 128 Fn gedrückt
```

Mehrere gedrückte Tasten kombinieren sich: `mod%= 6` entspricht: Shift und Strg gedrückt. Welche Taste gedrückt wurde, erfährt man durch "Siebung" mit dem logischen AND-Operator nach dem Prinzip:

```
IF mod% AND 2
  PRINT "Shift gedrückt"
ENDIF
```

`KMOD` ist immer unmittelbar nach den `GET-/GET$-/KEY-/KEY$`-Anweisungen zu verwenden!

**LAST***OPL32 – SIBO*`LAST`

Macht den letzten Datensatz einer DB (`OPL32:` oder in der aktuellen Ansicht) zum aktuellen Datensatz.

Siehe auch `BACK`, `FIRST`, `NEXT`

**LCLOSE**  
LCLOSE

OPL32 – SIBO

Schließt eine mit LOPEN geöffnetes Gerät. Bei Programmende werden alle Geräte automatisch geschlossen.

**LEFT\$**

OPL32 – SIBO

```
teilstring$=LEFT$(string$,x%)
```

Gibt x% Zeichen der Zeichenkette string\$ von ganz links beginnend an teilstring\$ zurück. Beispiel:

```
string$="Mein Text" : x%=4
teilstring$=LEFT$(string$,x%)
REM Ergebnis: teilstring$ enthält "Mein"
```

Siehe auch: MID\$, RIGHT\$

**LEN**

OPL32 – SIBO

```
stringlaenge%=LEN(string$)
```

Übergibt die Länge der Zeichenkette string\$ an stringlaenge%. Beispiel:

```
PRINT LEN("Hallo")
REM Ergebnis: 5.
```

**LENALLOC**

F - OPL32 – SIBO

```
len&=LENALLOC(pcell&) REM OPL32
len%=LENALLOC(pcell%) REM SIBO
```

Liefert die Länge der reservierten Zelle an Adresse pcell& bzw. pcell%.

**LINKLIB**

F - SIBO

```
LINKLIB(cat%)
```

Verknüpft mit LOADLIB geladene Bibliotheken.

**LN**

OPL32 – SIBO

```
y=LN(x)
```

Liefert den natürlichen Logarithmus von x zurück.

**LOADLIB**

F - SIBO

```
ret%=LOADLIB(cat%,name$,link%)
```

Lädt eine Bibliothek und verknüpft sie optional, wenn sie nicht im ROM steht. Bei Erfolg schreibt LOADLIB den Kategorie-Handle in cat% und liefert 0 zurück. DYLS werden nur einmal in den Speicher geladen und dort von mehreren Prozessen geteilt.

**LOADM***OPL32 – SIBO*`LOADM modul$`

Lädt ein übersetztes OPL-Modul, um die Prozeduren des Moduls aufrufen zu können.

`modul$` enthält dabei den Dateinamen des übersetzten Moduls, wenn nötig mit Verzeichnis. Beispiel:

```
LOADM "RECHNEN.OPO"
```

Es können bis zu 4 Module (SIBO) bzw. 8 Module (OPL32) zugleich geladen werden. Bevor Sie ein weiteres Modul hinzuladen, müssen Sie eines der anderen wieder mit `UNLOADM` aus dem Speicher entfernen.

`LOADM` benutzt standardmäßig:

OPL32: das Verzeichnis des ersten Moduls,

SIBO: das Verzeichnis, in dem die Programmdatei liegt oder das in einer Applikation angegebene.

In beiden Fällen wird der Pfad nicht von `SETPATH` beeinflusst.

Um allen Problemen aus dem Weg zu gehen, sollte man von vornherein den Pfad mit benennen.

**LOC***OPL32 – SIBO*

```
position%=LOC(string$,suchstring$)
```

Übergibt die Position von `suchstring$` innerhalb von `string$` an `position%`. Ist `suchstring$` nicht in der durchsuchten Zeichenkette, wird `position%` Null. Wenn `suchstring$` aus mehr als einem Zeichen besteht, gibt `position%` die Position des ersten übereinstimmenden Zeichens wieder. Die Suche nimmt keine Rücksicht auf Groß- und Kleinschreibung. Beispiel:

```
position%= LOC("Buchhandel", "Handel")
REM Ergebnis: position% ist 5.
```

**LOCAL***OPL32 – SIBO*`LOCAL variablen`

Wird benutzt, um Variablen zu deklarieren. Die Variablen sind nur innerhalb der aktuellen Prozedur gültig. In von der Prozedur aufgerufenen Prozeduren werden die Variablen nicht erkannt. Um Variablen zu deklarieren, die auch von aufgerufenen Prozeduren erkannt werden, müssen Sie `GLOBAL` verwenden.

Die möglichen Werte für die Variablen sind bei der Beschreibung von `GLOBAL` aufgeführt.

Siehe auch `GLOBAL`

**LOCK***F - OPL32 – SIBO*

```
LOCK ON oder
LOCK OFF
```

Teilt dem Betriebssystem mit, dass die Applikation derzeit keine Nachrichten empfangen kann. Wenn Sie im Systembildschirm versuchen, die Applikation zu beenden oder die verwendete Datei zu wechseln, wird eine entsprechende Meldung ausgegeben.

Sie sollten `LOCK ON` benutzen, wenn Ihre Applikation die Tastatur über Befehle wie `EDIT` oder `DIALOG` synchron abfragt oder eine längere Berechnung durchführt. Andere Systemnachrichten, wie z.B. "Rechner aus" oder "Vordergrund/Hintergrund" können noch geschickt werden, werden aber ignoriert.

Die Standardeinstellung ist `LOCK OFF`.

**LOG**

OPL32 – SIBO

```
y=LOG(x)
```

Liefert den dekadischen Logarithmus von x zurück.

**LOPEN**

OPL32 – SIBO

```
LOPEN einheit$
```

Öffnet die Einheit, an die der LPRINT-Befehl seine Daten schickt. LPRINT kann nicht benutzt werden, bis eine Einheit mit LOPEN geöffnet wurde. "Einheiten" sind z.B. Geräte wie Drucker, aber auch Dateien.

Der Parallel-Port wird mit LOPEN "PAR:A", der serielle Port mit LOPEN"TTY:A", eine Datei auf dem Psion mit LOPEN name\$ geöffnet. Es kann immer nur eine Einheit zugleich geöffnet sein. Mit LCLOSE wird eine Einheit wieder geschlossen.

OPL32: Es können zusätzlich auch Dateien auf angeschlossenen und verbundenen Computern angesprochen werden:

```
PC:          LOPEN "REM::C:\BAK\MEMO.TXT"
Apple Macintosh: LOPEN "REM::HD40:ME:MEMO5"
```

**LOWER\$**

OPL32 – SIBO

```
kleinezeichen$=LOWER$(string$)
```

Wandelt Großbuchstaben in Kleinbuchstaben.

Siehe auch: UPPER\$

**LPRINT**

OPL32 – SIBO

```
LPRINT Liste von Ausdrücken
```

Gibt wie PRINT die übergebene Liste von Ausdrücken aus. Ausgabeziel ist jedoch die letzte mit LOPEN geöffnete Einheit. Die Ausdrücke können Stringkonstanten, Variablen oder mathematische Ausdrücke sein. Kommas bzw. Strichpunkte haben dieselbe Wirkung wie bei PRINT. Wurde keine Einheit geöffnet, wird ein Fehler ausgegeben.

Siehe auch PRINT, LOPEN, LPRINT

**MAX**

OPL32 – SIBO

```
y=MAX(liste)
y=MAX(array(),element)
```

Liefert den Maximalwert einer Werteliste zurück. Die Werteliste ist entweder eine kommasetrennte Liste:

```
y=MAX(22, 2*34, 180/PI,17,4)
```

oder ein Array von Fließkommazahlen:

```
LOCAL wert(20), n% REM Werte in wert() nicht vergessen zu setzen!
n%=10
y=MAX(wert(),n%)
```

Das erste Argument ist der Fließkomma-Array-Name, danach folgt mit n% die Angabe der Elemente, die ab dem ersten Element mit in die Bewertung einbezogen werden sollen.

**mCARD**

```
mCARD titel$,auswahl1$,shortcut1$,auswahl2$,shortcut2$, ...
```

Definiert zwischen mINIT und MENU ein Menü mit den einzelnen Menüpunkten. Der Name des Menüs wird durch titel\$ festgelegt (Zeichenanzahl ist auf 40 begrenzt!). Es folgen Paare aus Namen des Menüpunktes und zugehörigem Shortcut.

Über den Shortcut (eine Kombination aus Strg- und Buchstaben-Taste) kann ein Menüpunkt direkt aufgerufen werden, ohne das Menü "aufklappen" zu müssen. Die Verwendung von Groß- und Kleinbuchstaben ist möglich. Es sind bis zu acht Name/Shortcut-Paare erlaubt.

Beispiel:

```
mCARD "Datei", "Öffnen",%o,"Neu",%n,"Beenden",%e
```

Durch Modifizieren der Shortcuts sind Darstellung und Funktionsweise von Menüs beeinflussbar. So zieht ein Minuszeichen vor dem Shortcut (-n%) eine Linie unter diesen Punkt. Verwendet man für den Shortcutwert die Zahlen von 1 .. 32, werden in der Menüdarstellung keine Shortcuts angezeigt, der Wert bei Auswahl des Punktes aber immer noch an MENU übergeben.

Weitere Modifikationen - die Werte sind zu den Shortcutwerten zu addieren oder zu ODERieren:

Wert	Auswirkung
\$1000	Menüpunkt grau dargestellt, nicht auswählbar
\$0800	Menüpunkt hat eine Checkbox
\$0900	Kennzeichnet den Beginn einer Optionsbuttonliste
\$0A00	Kennzeichnet einen mittleren Optionsbutton, mehrere Buttons sind möglich
\$0B00	Kennzeichnet das Ende einer Optionsbuttonliste
\$2000	zeigt das Symbol an einer Checkbox oder einem Optionsbutton
\$4000	Keine Symbolanzeige

Optionsbuttons bilden eine Gruppe, die zugehörigen Eigenschaften oder Reaktionen schließen sich einander aus, so dass immer nur ein Button gewählt sein kann.

Durch die Modifikationen wird lediglich das optische Aussehen der Menüs beeinflusst, für die Umsetzung der dahinter stehenden Funktionen ist der Programmierer zuständig. Das gilt auch für Checkboxes.

Beispiel:

```
PROC Checkbox:
...
mINIT
mCARD "Einstellen","Toolbar",%j OR $0800 OR (k%*$2000)
m%= MENU
IF m%= %j
  Wechselschalter:
ENDIF
...
ENDP

PROC Wechselschalter:
IF k%
  k%= 0
  REM und Toolbar ausschalten ...
ELSE
  k%= 1
  REM und Toolbar einschalten ...
ENDIF
ENDP
```



Es dürfen mehrere Checkboxes und Optionsgruppen in einem Menü erscheinen. Die Optionsgruppen müssen aber zusammengehalten werden. Achtung! OPL überprüft nicht, ob der Programmierer hier korrekt gearbeitet hat.

Siehe auch `mINIT`, `MENU`

### **mCARD**

*SIBO*

```
mCARD titel$,n1$,k1%
mCARD titel$,n1$,k1%,n2$,k2%
...
```

`mCARD` definiert ein Menü. Sind alle Menüs definiert können Sie diese mit `MENU` anzeigen.

`titel$` ist der Name des Menüs. Die Argumentpaare `n1$-n8$` und `k1%-k8%` definieren einen der bis zu 8 Menüeinträge. `nx$` gibt jeweils den Namen und `kx%` den Tastaturcodes des zu verwendenden Tastenkürzels an. Das Tastenkürzel (z.B. "A") kann dann zusammen mit der Taste PSION (also z.B. PSION-SHIFT-A) benutzt werden, um die Menüoption direkt aufzurufen. Bei Tastenkürzeln wird Groß-/Kleinschreibung unterschieden.

### **mCASC**

*OPL32*

```
mCASC titel$,ausw1$,shortcut1$,ausw2$,shortcut2%, ...
```

Erzeugt ein kaskadiertes Menü ("Untermenü"). Die Markierung ">" in einem der Menüpunkte in `mCARD` verweist auf die Benutzung eines Untermenüs. Dieses muss definiert sein, bevor in `mCARD` sein Aufruf präpariert wird:

```
mINIT
mCASC "Ausrichtung", "links",%1,"rechts",%r
mCARD "Text", "Ausrichtung>",16,"Zeilenabstand",%N
REM auch: mCARD "Text", "Ausrichtung>",%A,"Zeilenabstand",%N
m%= MENU
```

Im Menü selber wird als Hinweis auf das Untermenü das ">"-Zeichen zu einem ausgefüllten Pfeil. Der Titel des kaskadierten Menüs muss identisch mit der Menüpunktbezeichnung sein, wird aber nicht angezeigt. Zumeist besitzen Menüpunkte mit Untermenü kein eigenen Shortcut, der darf aber durchaus vergeben werden.

Modifizierte Shortcuts können ebenso wie bei `mCARD` das Aussehen und die Funktion des Untermenüs beeinflussen (Checkboxes, Optionbuttons, Linien, ...) - siehe dort.

Siehe auch `mCARD`, `MENU`, `mINIT`

### **MEAN**

*OPL32 – SIBO*

```
y=MEAN(liste)
y=MEAN(array(),element)
```

Liefert den arithmetischen Mittelwert einer Werte-Liste. Die Werteliste ist entweder eine kommagetrennte Liste:

```
y=MEAN(22, 2*34, 180/PI,17,4)
```

oder ein Array von Fließkommazahlen:

```
LOCAL wert(20), n%
REM Werte in wert() setzen nicht vergessen
n%=10
y=MEAN(wert(),n%)
```

**MENU**

```
m%=MENU
m%=MENU(position%)
```

Zeigt vorher definierte Menüs (mINIT .. mCASC .. mCARD .. MENU) an und wartet auf die Auswahl durch den Nutzer. Die Auswahl wird über MENU nach m% übergeben. m% enthält den im Shortcut benannten Zeichen- bzw. Zahlenwert (1 .. 32, wenn ohne Shortcut gearbeitet wird). <Esc> liefert eine Null zurück.

Wenn ein position%-Wert als Parameter übergeben wird, zeigt das Menü einen vorbestimmten Menüpunkt aktiviert an. position% setzt sich additiv aus einer Zahl für den Menütitel und den Menüpunkt zusammen:

```
position%= 256*titelposition% + menüpunktposition%
```

Dabei steht Null für jeweils den ersten Titel und den ersten Menüpunkt, eine 1 für jeweils die zweite Position. Z.B.:

```
position%= 256*1 + 2 REM 2. Menütitel, 3. Punkt
```

Nach der Auswahl wird die letzte markierte Menüposition nach position% zurückgeschrieben, so dass mit dem nächsten Menüaufruf die zuletzt gewählte Menüposition gleich wieder markiert wird.

OPL32: Es ist nicht korrekt, Fehler von mCARD und mCASC zu ignorieren, indem man ONERR verwendet. In diesem Falle wird das Menü nicht aufgerufen und bei Verwendung von mCARD, mCASC oder MENU eine "Strukturfehler"-Meldung ausgegeben, wenn man nicht erst noch einmal mINIT verwendet.

Beispiel, in dem das Menü nicht aufgerufen wird:

```
mINIT
ONERR errIgnore1
mCARD "Xxx","ItemA",0 REM falscher Shortcut
errIgnore1::
ONERR errIgnore2
mCARD "Yyy","" REM 'Strukturfehler' (mINIT verworfen)
errIgnore2::
ONERR OFF
MENU REM noch einmal 'Strukturfehler'
```

Siehe auch mINIT

**MID\$**

```
teilstring$= MID$(string$,start%,anzahl%)
```

Gibt anzahl% Zeichen der Zeichenkette string\$ an teilstring\$ zurück, beginnend ab dem Zeichen, das an der Position start% steht. Beispiel:

```
string$="Romantextbeispiel" : start%= 6 : anzahl%= 4
teilstring$= MID$(string$,start%,anzahl%)
REM Ergebnis: teilstring$ enthält "text"
```

Siehe auch: RIGHT\$, LEFT\$

**MIME**

```
MIME pri%, dtype$
```

Assoziiert eine OPL-Applikation mit dem Internet-MIME-Content-Typ dtype\$ mit der Priorität pri%. Dem System wird dadurch mitgeteilt, dass die Applikation dem Nutzer gestattet, Dateien der benannten Art anzusehen oder zu bearbeiten.

Der Befehl kann nur zwischen APP .. ENDA eingesetzt werden.

```
APP myEditor, KUidMyEditorApp&
  FLAGS KFlagsAppFileBased%
  CAPTION "myEditor", KLangEnglish%
  MIME KDataTypePriorityNormal%, "text/plain"
ENDA
```

## **MIN**

*OPL32 – SIBO*

```
y=MIN(list)
y=MIN(array(),element)
```

Liefert den Minimal-Wert einer Werte-Liste. Die Werteliste ist entweder eine kommasetrennte Liste:

```
y=MIN(22, 2*34, 180/PI,17,4)
```

oder ein Array von Fließkommazahlen:

```
LOCAL wert(20), n%
REM Werte in wert() setzen nicht vergessen!
n%=10
y=MIN(wert(),n%)
```

Das erste Argument ist der Fließkomma-Array-Name, danach folgt mit n% die Angabe der Elemente, die ab dem ersten Element mit in die Bewertung einbezogen werden sollen.

## **mINIT**

*OPL32 – SIBO*

```
mINIT
```

Beginn einer Menüdefinition (mINIT .. mCASC .. mCARD .. MENU), beendet andere begonnene Menüdefinitionen.

Siehe auch mCASC, mCARD, MENU

## **MINUTE**

*OPL32 – SIBO*

```
m%=MINUTE
```

Liefert die Minuten der aktuellen Systemzeit (0-59).

## **MKDIR**

*OPL32 – SIBO*

```
MKDIR name$
```

Erstellt ein neues Verzeichnis.

Beispiel: MKDIR "C:\NEU\VERZ" erstellt das Verzeichnis C:\NEU\VERZ und ebenso das Verzeichnis C:\NEU, falls es noch nicht existiert.

**MODIFY**

OPL32

MODIFY

Leitet eine Änderung am aktuellen Datensatz ein, wird mit `PUT` oder `CANCEL` abgeschlossen. Die Position des Datensatzes in der Ansicht bleibt erhalten.

```
MODIFY
  A.name$= "Anette"
PUT
```

Siehe auch `CANCEL`, `INSERT`, `PUT`

**MONTH**

OPL32 – SIBO

m%=MONTH

Liefert den Monat des aktuellen Systemdatums (1-12).

**MONTH\$**

OPL32 – SIBO

m\$=MONTH\$

Konvertiert die Zahlen 1-12 in eine 3 Zeichen lange Abkürzung des Monatsnamens.

**mPOPUP**

OPL32

```
m%=mPOPUP (x%,y%,bezugspos%,
           ausw1$,shortcut1$,ausw2$,shortcut2$,...)
REM alles in eine Zeile!
```

Zeigt ein von der `mINIT` ... `MENU`-Struktur unabhängiges, platzierbares Menü. Die Variablen `x%` und `y%` liefern die pixelgenauen Koordinaten für die Anordnung, `bezugspos%` nennt den Eckpunkt, auf den sich `x%` und `y%` beziehen. Es gilt:

bezugspos%	Ort
0	linke obere Menüecke
1	rechte obere Menüecke
2	linke untere Menüecke
3	rechte untere Menüecke

`m%` enthält nach der Auswahl den Wert der Shortcut-Taste (siehe `mCARD`) und Null, wenn `<Esc>` gedrückt wurde. Es dürfen die gleichen Views- und Funktions-Modifikationen wie bei `mCARD` vorgenommen werden. Beispiel:

```
m%=mPOPUP (20,20,0,"DEM",%m,"Euro",%u)
```

Die linke obere Ecke des Popup-Menüs steht an der Position 20,20. Siehe auch `mINIT`, `mCARD`, `MENU`

**NEWOBJ**

F - SIBO

pobj%=NEWOBJ (num%,clnum%)

Erstellt über die Kategoriennummer `num%` ein neues Objekt der Klasse `clnum%` und liefert den Handle zurück oder 0 wenn nicht genügend freier Speicher vorhanden ist.

**NEWOBJH***F - SIBO*

```
pobj%=NEWOBJH(cat%,clnum%)
```

Erstellt über den Kategoriehandle cat% ein neues Objekt der Klasse clnum% und liefert den Handle zurück oder 0 wenn nicht genügend freier Speicher vorhanden ist.

**NEXT***OPL32 – SIBO*

```
NEXT
```

Macht den nächsten Datensatz in der aktuellen Ansicht zum aktuellen Datensatz. Wenn der aktuelle Datensatz bereits der letzte Datensatz (LAST) ist, hat der aktuelle Datensatz keinen Inhalt. EOF wird auf WAHR, also -1 gesetzt.

Siehe auch BACK, FIRST, LAST

**NUM\$***OPL32 – SIBO*

```
string$=NUM$(zahl,anzahl%)
```

Aus der Fließkommazahl zahl wird eine Integerzahl gebildet, die als String zurückgegeben wird. Die Anzahl der Zeichen in string\$ wird mit anzahl% festgelegt. Ist anzahl% negativ, wird der String rechtsbündig gebildet. Stellen ohne Ziffern werden durch Leerzeichen ersetzt. Reicht der durch anzahl% bestimmte Platz nicht zur Aufnahme der Zahl aus, werden Asterisks (\*) eingesetzt.

Siehe auch: FIX\$, GEN\$, SCI\$

**ODBINFO***F - SIBO*

```
ODBINFO info%()
```

Dieser Befehl ist nur für erfahrene Programmierer gedacht und ermöglicht den Zugriff auf Betriebssystemfunktionen, die nicht in OPL zur Verfügung stehen.

Unter OPL32 stehen Zugriffe auf das Betriebssystem nur über OPX-Module zur Verfügung.

**OFF***OPL32 – SIBO*

```
OFF
OFF x%
```

Schaltet den Psion ab.

Wird der Psion wieder eingeschaltet, führt das Programm den im Programm nach dem OFF- Befehl folgenden Befehl aus:

```
OFF : PRINT "Hello again"
```

Wird x% (2...16383) angegeben, schaltet sich der Psion nach x% Sekunden wieder an.

Vorsicht: Wenn das Programm OFF falsch in einer Schleife aufruft schaltet sich das Gerät immer wieder ab und kann nur durch Reset wieder zurückgeholt werden.

OPL32: Die minimale Ausschaltzeit beträgt hier 5 Sekunden. Der Psion schaltet nicht aus, wenn ein absoluter Zähler programmiert ist, der in weniger als 5 Sekunden fertig gezählt hat.

**ONERR**

```
ONERR fehlerbehandlung::
...
fehlerbehandlung::
ONERR OFF
REM hier folgt die Fehlerauswertung/-behandlung
...
```

ONERR organisiert eine Fehlerbehandlung in einer Prozedur, ohne dass das Programm abbricht. Tritt ein Fehler auf, springt das Programm zur Marke fehlerbehandlung::. Dort wird ONERR durch den Gegenbefehl ONERR OFF wieder außer Kraft gesetzt und die Fehlerbehandlung durchgeführt. ONERR OFF sollte direkt nach der Marke aufgerufen werden um Endlosschleifen zu vermeiden.

Die Markenbezeichnung darf bis zu 32 Zeichen lang sein (OPL32, bei SIBO 8 Zeichen). Sie muss mit einem Buchstaben oder Unterstrich anfangen, außerdem muss sie mit einem zweifachen Doppelpunkt abgeschlossen werden. Der zweifache Doppelpunkt kann im ONERR-Befehl entfallen.

**OPEN**

```
OPEN "meineDB SELECT Feld1,Feld2,Feld3 ... FROM TabName",
                                logName,f1,f2,f3,...
REM die beiden letzten Zeilen in eine schreiben!
bzw.
DBname$= "meineDB"
OPEN DBname$ + " SELECT Feld1,Feld2,Feld3 ... FROM
                                TabName",logName ,f1,f2,f3,...
REM die beiden letzten Zeilen in eine schreiben!
```

Öffnet eine Tabellenansicht der Datenbank Dbname\$ (kann eine komplette Dateispezifikation einschließlich Laufwerk und Pfad sein). Die zu öffnende Tabelle hat den Namen "TabName". Die zu verwendenden Feldbezeichner stehen als kommagetrennte Liste zwischen SELECT und FROM und müssen mit den bei CREATE gewählten übereinstimmen. Der logische Name für die Ansicht ist logName, einer der Buchstaben von "A" bis "Z". Die zu verwendenden passenden Variablen sind f1, f2, f3 usw. Die Variablen müssen nicht vorab deklariert werden und müssen nicht mit den Variablen identisch sein, die bei CREATE verwendet wurden, jedoch muss der Variablentyp stimmen.

Es müssen nicht alle Felder einer Tabelle benannt und verwendet werden. Mit OPEN lassen sich mehrere (unterschiedliche) Ansichten einer Tabelle gleichzeitig öffnen, der logische Name sorgt für die Unterscheidbarkeit.

Will man alle Felder einer Tabelle verwenden, darf die Liste zwischen SELECT und FROM mit dem Ersatzzeichen (Wildcard) "\*" abgekürzt werden.

Eine Ansicht kann auch eine gezielte Sortierung enthalten, wenn man beim Öffnen nach folgendem Beispiel ORDER BY angibt:

```
OPEN DBname$ + " SELECT Index,Name FROM Liste ORDER BY
                                Index DESC,Name ASC",D,i%,n$
REM die beiden letzten Zeilen in eine schreiben!
```

DESC steht für absteigende und ASC für aufsteigende Sortierung.

Siehe auch CREATE, OPENR, USE

**OPEN**

```
OPEN datei$,log,f1,f2...
```

Öffnet eine existierende Datenbankdatei mit Namen datei\$. Der Datei werden der logische Name log und die Felder f1, f2... zugeordnet. Sie müssen beim Öffnen nur die Felder angeben, auf die Sie zugreifen möchten.

Eine geöffnete Datei wird im Programm immer über den logischen Dateinamen (A-D) angesprochen. Es können bis zu 4 Dateien parallel geöffnet sein.

Beispiel:

```
OPEN "Kunden",A,name$,addr$
```

Siehe auch CREATE, USE, OPENR

## OPENR

OPL32 – SIBO

Wie OPEN, nur dass die Datei nur gelesen werden kann. Dieser Befehl bietet den Vorteil, dass andere Programme parallel lesend auf die Datei zugreifen können.

## OS

F - SIBO

```
a%=OS(i%, addr1%)
a%=OS(i%,addr1%,addr2%)
```

Ruft den Systeminterrupt i% auf und liest die Werte aller Register des 8086 aus. Die Eingaberegister werden über addr1% versorgt, die Ausgaberegister über addr2%. Wird addr2% nicht angegeben werden die zurückgelieferten Werte in addr1% abgelegt. Beide Adressen müssen auf ein Array oder auf 6 aufeinanderfolgende Integers zeigen. Die Register werden sequentiell in den Array-Elementen in folgender Reihenfolge abgelegt: AX,BX,CX,DX,SI,DI. Die Funktionsnummer des Interrupts wird, wenn benötigt, an AH übergeben.

Das Ausgabearray muss mindestens 6 Integers aufnehmen können.

OS liefert (in a%) das Flag-Register zurück. Das Carry-Flag, das in den meisten Fällen benötigt wird entspricht Bit 0 des zurückgelieferten Werts. Das Bit ist gesetzt wenn das Carry-Flag gesetzt ist. Die Flags "Zero", "Sign" und "Overflow" stehen in den Bits 6,7 und 10.

Beispielprogramm (Berechnet  $\cos(\pi/4)$ ):

```
PROC add:
  LOCAL ax%,bx%,cx%,x%,si%,di%
  LOCAL ergebnis,cosArg,flags%
  cosArg=PI/4
  si%=ADDR(cosArg)
  di%=ADDR(ergebnis)
  ax%=$0100
  REM AH=1 für Kosinusfunktion
  flags%=os(140,addr(ax%))
```

Die OS-Funktion bedingt fundierte Betriebssystemkenntnisse und sollte nur von erfahrenen Programmierern angewandt werden.

Siehe auch CALL

## OPL32 – SIBO

**PARSE\$**

```
p$=PARSE$(f$,rel$,off%())
```

Liefert die volle Dateispezifikation von f\$, fehlende Informationen werden aus rel\$ eingefügt. Die Positionen der einzelnen Elemente der Dateispezifikation werden in off%() zurückgeliefert. Das Array muss mindestens 6 Integer groß sein.

```
off%(1)  Dateisystem (immer 1)
off%(2)  Laufwerk
off%(3)  Pfad
off%(4)  Dateiname
off%(5)  Dateierweiterung
off%(6)  Flags für Platzhalter im
         zurückgelieferten String
```

Die Flags haben folgende Bedeutung:

```
0  keine Platzhalter
1  Platzhalter im Dateinamen
2  Platzhalter in der Erweiterung
3  Platzhalter in beiden
```

Wenn rel\$ selbst keine volle Dateispezifikation ist, werden die fehlenden Teile aus den Werten für das aktuelle Laufwerk etc. aufgefüllt. f\$ und rel\$ sollten unterschiedliche Strings sein.

Beispiel OPL32:

```
p$=PARSE$("NEU","C:\Documents\*.MBM",x%())
REM p$ wird C:\Documents\NEU.MBM und
REM x%() zu (1,1,3,14,17,0)
```

Beispiel SIBO:

```
f$=PARSE("NEU","LOC::M:\ODB\*.ODB",x%())
REM p$ wird LOC::M:\ODB\NEU.ODB und
REM x%() zu (1,6,8,13,16,0).
```

## SIBO

**PATH**

```
PATH name$
```

Setzt das von der Applikation zu verwendende Verzeichnis. Kann nur zwischen APP und ENDA benutzt werden.



## OPL32 – SIBO

**PAUSE**

PAUSE x%

Hält das Programm für x% (in zwanzigstel Sekunden) an. Ist x%=0, wird lediglich auf einen Tastendruck gewartet. Ist der Wert negativ, gilt die angegebene Zeitspanne, wahlweise kann mit einem Tastendruck abgebrochen werden.

Wenn x% kleiner oder gleich Null ist, bekommt man über die Verwendung von GET, GET\$, KEY oder KEY\$ heraus, welche Taste gedrückt wurde. Interessiert das aber nicht, sondern erst der Tastendruck danach, muss man den Tastaturpuffer zuerst mit einem einfachen KEY leeren:

```
PAUSE -100
a$= GET$
PRINT a$
PAUSE -100 : KEY
a%= GET
PRINT a%
```

PAUSE überschreibt andere Events, daher nicht mit im Zusammenhang mit GETEVENT/GETEVENT32 verwenden!

## F - OPL32 – SIBO

**PEEK Funktionen**

p%=PEEKB(x&) liefert den Wert des Bytes an der Adresse x& als integer Zahl,  
 p%=PEEKW(x&) liefert den Wert der zwei Bytes an der Adresse x& als Integer-Zahl,  
 p%=PEEKL(x&) liefert den Wert der long integer Zahl an der Adresse x&,  
 p=PEEKF(x&) liefert den Wert des floating point Zahl an der Adresse x&.

Unter SIBO kann die Long-Integer-Variable x& durch eine normale Integer-Variable ersetzt werden, z.B. x%.

Sie können die Adresse einer Variablen über die Funktion ADDR herausfinden (z.B.: ADDR(a%)) und dann die Abfrage starten.

Beispiel für p%= 7:

```
i%= PEEKW(ADDR(p%))
REM i% --> 7
```

Die verschiedenen Typen werden unterschiedlich im Speicher abgelegt:

Integers werden in zwei Bytes abgelegt, das niederwertige Byte zuerst. ADDR liefert die Adresse des ersten Bytes.

Long-Integers werden in vier Bytes abgelegt, das niederwertigste Byte zuerst, das höchstwertige zuletzt. ADDR liefert die Adresse des ersten Bytes.

Strings werden fortlaufend mit einem Byte je Zeichen abgelegt. Im ersten Byte, dessen Adresse von ADDR geliefert wird, ist die Länge des Strings gespeichert. Jedes Zeichen wird mit seinem Zeichencode abgelegt. "ABC" würde abgelegt als 3,65,66,67

Fließkomma-Zahlen werden im IEEE-Format abgelegt. Ein Floating point belegt dabei 8 Bytes. PEEKF liest automatisch alle 8 Bytes und konvertiert sie in einen korrekten Fließkomma-Wert.

Querverweis POKE, ADDR

**PI***OPL32 – SIBO*`y=PI`

Gibt den Wert der Zahl  $\pi$  (3,14... ) zurück.

**POINTERFILTER***F - OPL32*`POINTERFILTER filter%,mask%`

Gestattet die Ein- oder Ausfilterung von Events des "Zeigers" (also hier des Stiftes) im aktuellen Fenster.

Folgende Werte, die addiert werden können, gelten für filter% und flag%:

Event	Wert
keiner	0
Aufsetzen/Abheben	1
Ziehen	4

In filter% gesetzten Bits sagen, welcher Event ausgefiltert werden soll, 1 setzt den Filter, 0 entfernt den Filter. Ob der Filter tatsächlich auch angewendet wird, wird mit den gleichen Bits in flag% festgelegt: bei 1 ist die Filterung aktiv, bei 0 nicht.

Beispiel:

```
mask%=5
REM =1+4 - gestattet die Änderung von Aufsetzen/Abheben
REM und Ziehen 4
POINTERFILTER 1,mask%
REM filtert Aufsetzen/Abheben, aber nicht Ziehen
...
POINTERFILTER 4,mask%
REM filtert Ziehen, setzt Aufsetzen/Abheben wieder ein
```

Normalerweise werden Events nicht ausgefiltert.

Siehe auch GETEVENT32, GETEVENTA32

**POKE Befehle***F - OPL32 – SIBO*

Die POKE-Befehle speichern Werte an den angegebenen Adressen.

```
POKEB x&,y% speichert ein Byte mit Wert y% (<256) an der Adresse x&
POKEW x&,y% speichert den Integer y% in zwei aufeinanderfolgenden Bytes
               an der Adresse x&. Das niederwertige Byte wird an Adresse x&,
               das höherwertige an x&+1 gespeichert.
POKEL x&,y& speichert den long integer Wert y& in vier
               aufeinanderfolgenden Bytes an der Adresse x&.
POKEF x&,y  speichert den Floating point Wert y in
               aufeinanderfolgenden Bytes ab Adresse x&.
POKE$ x&,y$ speichert den String y$ in
               aufeinanderfolgenden Bytes ab Adresse x&.
```

Unter SIBO können die Adressangaben durch eine Integervariable ersetzt werden, z.B. x%.

Mit ADDR können Sie die Adresse einer Variablen ermitteln.

Vorsicht: Falsche Anwendung des Befehls kann zu Datenverlust führen!

Sehen Sie unter PEEK nach, wenn Sie mehr über die einzelnen Datentypen und Ihre Speicherung erfahren wollen.

**POS***SIBO – ( OPL32)*

p%=POS

Liefert die Datensatznummer des aktuellen Datensatzes in der aktuellen Datei. 1 entspricht dem ersten Datensatz, 2 dem zweiten etc.

Eine Datei kann bis zu 65534 Datensätze enthalten. Integers können jedoch nur Werte zwischen -32768 und +32767 speichern. Datensatznummern über 32767 werden daher als negative Zahlen wie folgt ausgegeben:

```
32767 --> 32767
32768 --> -32768
32769 --> -32767
32770 --> -32766
...    --> ...
65634 --> -2
```

Um die Datensatznummer auszugeben, können Sie folgende Routine verwenden:

```
IF POS < 0
  PRINT 65536+POS
ELSE
  PRINT POS
ENDIF
```

Achtung unter OPL32!

Unter OPL32 kann die Anzahl der Datensätze größer als 65535 sein, das kann zu falsche Ergebnissen bei p% führen. Wenn Sie also unter OPL32 mit einer größeren Anzahl von Datensätzen arbeiten wollen, wird dringend empfohlen, mit Bookmarks und evtl. mehreren Ansichten zu arbeiten. Verwenden Sie:

FIRST, LAST, BOOKMARK, GOTOMARK, KILLMARK.

**POSITION***SIBO – (OPL32)*

POSITION x%

Macht den Datensatz mit Nummer x% zum aktuellen Datensatz. Wenn x% größer als die Anzahl der Datensätze in der Datei ist, liefert EOF den Wert WAHR.

OPL32:

Wie der Befehl POS existiert POSITION hauptsächlich nur aus Kompatibilitätsgründen zum Serie3xx. Unter OPL32 verwenden Sie besser nur noch:

BOOKMARK, GOTOMARK, KILLMARK

**POSSPRITE***F - SIBO*

POSSPRITE x%,y%

Setzt die Position des aktuellen Sprites auf die Pixelposition x%,y%.

Unter OPL32 bestehen Möglichkeiten zur Spritebehandlung mit Hilfe der eingebauten OPX-Module.

## OPL32 – SIBO

**PRINT**

PRINT Liste von Ausdrücken

Zeigt eine Liste von Ausdrücken auf dem Bildschirm an. Die Liste kann wie folgt durch Interpunktion formatiert werden:

- werden die Einträge der Liste durch Kommas getrennt, werden die angezeigten Werte mit einem Leerzeichen als Zwischenraum angezeigt.
- sind die Einträge durch Strichpunkte getrennt, wird kein Zwischenraum zwischen den Werten ausgegeben.

Wird eine PRINT-Anweisung nicht mit einem Strichpunkt oder Komma abgeschlossen, wird am Ende des Befehls an den Anfang der nächsten Zeile gesprungen.

Sie können beliebig viele Einträge in die Liste schreiben, wird PRINT jedoch ohne Argument verwendet, so wird lediglich eine Leerzeile ausgegeben.

Beispiel:

Am 1. Januar 1993 würden die Befehle:

```
PRINT "Heute ist der",
PRINT DAY; " . " ; MONTH; " . " ; YEAR
```

"Heute ist der 1.1.1993" ausgegeben.

Siehe auch LPRINT, gUPDATE, gPRINT, gPRINTB, gPRINTCLIP, gXPRINT

## OPL32

**PUT**

PUT

Beendet das Einsetzen (INSERT) bzw. Ändern (MODIFY) eines Datensatzes und schreibt die Daten in die aktuelle Tabellenansicht einer geöffneten Datenbank.

Siehe auch INSERT, MODIFY, CANCEL

## OPL32 – SIBO

**RAD**

y=RAD(x)

Wandelt x (in Grad) in das Bogenmaß.

## OPL32 – SIBO

**RAISE**

RAISE fehler%

Es wird ein Fehler mit der Fehlernummer fehler% künstlich hervorgerufen. Dabei darf fehler% den Wert der festgelegten OPL-Fehler als auch selbstdefinierte Werte annehmen. OPL-Fehler haben alle einen negativen Wert - siehe Liste im Anhang.

Wird RAISE nicht innerhalb einer ONERR .. ONERR OFF Fehlerbehandlung verwendet, bricht das Programm ab und zeigt eine System-Fehlermeldung.

**RANDOMIZE**

OPL32 – SIBO

```
RANDOMIZE x&    OPL32
RANDOMIZE x%     SIBO
```

Gibt einen Startwert für RND vor. Benutzt man eine feste Zahl, wird die Anwendung von RND in der gleichen Art und Weise auch immer ein gleiches Ergebnis liefern - also auf keinen Fall eine echte Zufallszahl. Um von vornherein unterschiedliche RND-Werte zu erhalten, verknüpft man `RANDOMIZE` mit einem nichtstabilen Wert, z.B. der Systemzeit oder Teilen davon:

```
RANDOMIZE HOUR + MINUTE + SECOND
```

So kann man dann mit RND doch quasi-Zufallszahlen erzeugen, echte sind es auch dann noch nicht. `RANDOMIZE` in dieser Variante verwendet man zumeist einmal am Programmstart.

Siehe auch: `RND`

**REALLOC**

F - OPL32 – SIBO

```
pcelln&=REALLOC(pcell&,size&)    OPL32
pcelln%=REALLOC(pcell%,size%)     SIBO
```

Ändert die Größe der reservierten Zelle an Adresse `pzell%` auf die neue Größe `size%` und liefert die neue Adresse in `pzelln%` zurück. Ist nicht genug Speicher frei um die geänderte Größe zu reservieren, wird 0 zurückgeliefert.

**RECSIZE**

OPL32 – SIBO

```
r%=RECSIZE
```

Liefert die Größe des aktuellen Datensatzes der aktuellen Datei in Byte.

Sie können den Befehl zum Beispiel verwenden um zu überprüfen, ob Sie Daten an den Datensatz anhängen können, ohne die Grenze von 1022 Zeichen zu überschreiten.

Beispielprogramm:

```
PROC dstest:
  LOCAL n$(20)
  OPEN "name",A,name$
  PRINT "Name eingeben:",
  INPUT n$
  IF RECSIZE<=(1022-LEN(n$))
    A.name$=n$
    APPEND
  ELSE
    PRINT "Datensatz zu lang"
  ENDIF
ENDP
```

Unter EPOC32 nicht erforderlich, da Datensatzbegrenzung bei 32\*256 Zeichen liegt

**REM**

OPL32 – SIBO

```
REM text
```

Kennzeichnet den nachfolgenden Text als Bemerkung. Der Übersetzer ignoriert alle nach `REM` stehenden Zeichen bis zum Zeilenende.

Wenn Sie REM nach einem Befehl in die selbe Zeile schreiben wollen, können Sie den Doppelpunkt auch weglassen:

```
INPUT a
b=a*.15    REM b=Steuer
INPUT a
b=a*.15 : REM b=Steuer
```

## RENAME

OPL32 – SIBO

```
RENAME name1$,name2$
```

Benennt die Datei name1\$ um in name2\$. Sie können jeden beliebigen Dateityp umbenennen. Platzhalter sind in den Argumenten nicht gestattet.

Sie können verzeichnisübergreifend umbenennen, z.B.:

```
RENAME "M:\dat\x.dbf", "M:\x.dbf"
```

In diesem Fall können Sie den eigentlichen Dateinamen ändern, aber auch beibehalten.

Beispiel:

```
PRINT "Alter Name:", : INPUT a$
PRINT "Neuer Name:", : INPUT b$
RENAME a$,b$
```

## REPT\$

OPL32 – SIBO

```
multistring$=REPT$(string$,anzahl%)
```

Der String string\$ erscheint mehrfach wiederholt in multistring\$. Die Anzahl der Wiederholungen geben Sie mit anzahl% vor.

## RETURN

OPL32 – SIBO

```
RETURN
RETURN var
```

Bricht eine aufgerufene Prozedur vorzeitig ab und kehrt zur aufrufenden Prozedur zurück. Die Rückkehr aus einem Unterprogramm zum aufrufenden Programm wird auch automatisch durch ENDP bewirkt. In der Form RETURN var ist man in der Lage, einen Wert an den Aufrufer zurückzugeben (var= Integer, Fließkommazahl, String, jedoch kein Feld!)

```
PROC Hauptprogramm:
...
diff%=Unterprogramm%:(a%,b%)
PRINT Hole_Text$:(diff%)
...
ENDP

PROC Unterprogramm%:(x%,y%)
IF x%>y%
RETURN x%-y%
ELSE
RETURN y%-x%
ENDIF
ENDP
```

```

PROC Hole_Text$: (d%)
  IF d%=1
    RETURN "Text1"
  ELSEIF d%=2
    RETURN "Text2"
  ENDIF
  RETURN "Text3"
ENDP

```

**RIGHT\$**

OPL32 – SIBO

```

teilstring$=RIGHT$(string$,x%)

```

Gibt x% Zeichen der Zeichenkette string\$ von ganz rechts beginnend an teilstring\$ zurück.

Beispiel:

```

string$="Mein Text" : x%=4
teilstring$=RIGHT$(string$,x%)
Ergebnis: teilstring$ enthält "Text"

```

Siehe auch: LEFT\$, MID\$

**RND**

OPL32 – SIBO

```

y=RND

```

Ergibt bei jedem Aufruf eine andere zufällige Fließkommazahl im Bereich von Null (einschließlich) bis 1 (ausschließlich). Die Anfangsbedingungen für den Zufallsgenerator sind bei jedem Programmstart gleich, dadurch können unerwünscht immer gleiche Werte entstehen. Beispiel:

```

PROC Start:
  GLOBAL code%
  code%=1+INT(RND*9999) REM ganzz.Zufallszahl 1 .. 9999
  PRINT code%
  GET
ENDP

```

Auf dem gleichen Rechner ist code% immer gleich. Erst RANDOMIZE kann Abhilfe schaffen:

```

PROC Start:
  GLOBAL code%
  RANDOMIZE HOUR + MINUTE + SECOND
  code%=1+INT(RND*9999) REM ganzz. Zufallszahl 1 .. 9999
  PRINT code%
  GET
ENDP

```

Jetzt ist jeder code%-Wert immer ein anderer.

**RMDIR**

OPL32 – SIBO

```

RMDIR v$

```

Löscht das Verzeichnis v\$. Sie können nur leere Verzeichnisse löschen.

**ROLLBACK**

ROLLBACK

OPL32

Bricht die mit BEGINTRANS gestartete Transaktion ab. Bereits vorgenommene Datenänderungen, die ab dem Befehl BEGINTRANS stattfanden, werden verworfen.

Siehe auch BEGINTRANS, COMMITTRANS, INTRANS

**SCI\$**

OPL32 – SIBO

```
string$=SCI$(x,y%,z%)
```

Macht aus der Fließkommazahl x einen String im wissenschaftlichen Format, der y% Dezimalstellen anzeigt und bis zu z% Zeichen lang ist. Ist z% zu klein, werden Asterisks ("Sternchen") angezeigt. Bei negativem z%-Wert wird der String rechtsbündig dargestellt, den Rest des Platzes nehmen Leerzeichen ein. Beispiele:

```
SCI$(12345,3,8)    --> "1.235E+04"
SCI$(12345,3,7)    --> "*****"
SCI$(12345,3,-16)  --> "          1.235E+04"
```

Siehe auch FIX\$, GEN\$, NUM\$

**SCREEN**

OPL32 – SIBO

```
SCREEN weite%,hoehe%
SCREEN weite%,hoehe%,x%,y%
```

Ändert die Größe des Textfensters, das ohne eine Veränderung durch SCREEN den gesamten Bildschirm einnimmt. Die Angaben von weite% und hoehe% sind in Zeichen (und nicht als Pixel) anzugeben. Durch x% und y% (ebenfalls in "Zeichen") wird die Position festgelegt, ohne die Angabe wird das neue Textfenster zentriert auf dem Bildschirm dargestellt.

Für die maximal möglichen Werte für die Zeichenanzahl informiere man sich mit SCREENINFO. Die Werte hängen von der Bildschirmgröße und dem eingestellten Font ab.

Siehe auch FONT, SCREENINFO

**SCREENINFO**

OPL32 – SIBO

```
SCREENINFO info%()
```

Liefert Informationen über den Textbildschirm (der von PRINT etc. benutzt wird).

Dieser Befehl ermöglicht Ihnen, Text und Grafik zu mischen. Er wird benötigt, da das Standardfenster zwar die selbe Größe wie der gesamte Bildschirm hat, das Textfenster hingegen etwas kleiner ist und zentriert im Standardfenster dargestellt wird. Die Lücke zwischen Textfenster und Standardfenster (linker bzw. oberer Rand) hängt vom benutzten Font ab. Das Array info%(), das mindestens 10 Elemente haben muss, enthält nach Aufruf von SCREENINFO folgende Werte:

SIBO:

```
info%(1)  linker Rand in Pixel.
info%(2)  oberer Rand in Pixel
info%(3)  Breite des Textfensters in Zeichen.
info%(4)  Höhe des Textfensters in Zeichen, "Zeichen" steht für die Rastergröße
           der Einteilung und orientiert sich an den Abmassen des aktuellen Fonts)
info%(5)  Reserviert (Windowserver ID des Standardfensters)
info%(6)  Font ID
info%(7)  Breite der Zeichenzelle des Textfensters (in Pixel)
```



info%(8) Höhe einer Textzeile des Textfensters (in Pixel)  
 info%(9) reserviert  
 info%(10) reserviert

**OPL32:**

info%(6) nicht benutzt  
 info%(9) die niederwertigen 16 Bits der Font ID  
 info%(10) die höherwertigen 16 Bits der Font ID

Sie sollten SCREENINFO zur Information jedesmal neu aufrufen, wenn Sie den verwendeten Font, die Größe des Fensters oder ähnliche Werte geändert haben.

Siehe auch FONT, SCREEN

**SECOND***OPL32 – SIBO*

s%=SECOND

Liefert die Sekunden der Systemzeit (0-59).

**SECSTODATE***OPL32 – SIBO*

SECSTODATE s&,ja%,mo%,ta%,st%,mi%,se%,jtag%

Variable	für
ja%	Jahr
mo%	Monat
ta%	Tag
st%	Stunde
mi%	Minute
se%	Sekunde
jtag%	Tag im Jahr (1 .. 366)

Setzt die übergebenen Variablen auf die den angegebenen Sekunden (s&) entsprechenden Werte für Datum und Uhrzeit. Die Werte werden berechnet über s&=Sekunden seit dem 1.1.1970, 00:00.

s& ist ein vorzeichenfreier long integer Wert. Um Werte größer als +2.147.483.647 zu verwenden ziehen Sie einfach 4.294.967.296 ab.

Siehe auch DATETOSECS, HOUR, MINUTE, SECOND, dDATE, DAYS

**SEND***F - SIBO*

ret%=SEND(pobj%,m%,p1,...)

Sendet eine Nachricht an das Objekt pobj%, um die Methode mit Nummer m% aufzurufen. Die Argumente p1, p2 etc. hängen von den Anforderungen der gewünschten Methode ab

EPOC32: Funktion nur über OPX-Aufruf möglich.

**SETDOC**

```
SETDOC datei$
```

Kennzeichnet eine Datei mit Namen `datei$` als Dokument. Dieser Befehl sollte kurz vor dem Anlegen der Datei ausgeführt werden. SETDOC kann im Zusammenhang mit den Befehlen `CREATE`, `gSAVEBIT` und `IOOPEN` verwendet werden.

Der String, der an SETDOC übergeben wird, muss identisch sein mit dem, der an den nachfolgenden `CREATE`- oder `gSAVEBIT`-Befehl übergeben wird, ansonsten wird eine Nicht-Dokument-Datei erstellt.

Beispiel:

```
SETDOC "meineDB"
CREATE "meineDB",a,a$,b$
```

SETDOC sollte ebenfalls aufgerufen werden, nachdem eine Datei erfolgreich geöffnet wurde, damit der korrekte Name in der Liste der offenen Tasks angezeigt wird.

Wenn das Anlegen oder Öffnen einer Datei fehlschlägt, sollte man wie folgt reagieren:

- Beim Anlegen: Versuchen Sie, die vorher benutzte (Dokumenten-)Datei zu öffnen. Wenn auch das fehlschlägt, zeigen Sie einen entsprechenden Fehlerdialog an. Beim Wiederöffnen der vorher benutzten Datei achten Sie darauf, SETDOC mit dem richtigen Namen aufzurufen, damit die Taskliste korrekte Einträge aufweist.

- Beim Öffnen: Wie beim Anlegen, es ist allerdings nicht nötig, SETDOC ein weiteres Mal aufzurufen.

Datenbank-Dokumente, die mit `CREATE` erstellt wurden, und Multi-Bitmap-Dokumente, die mit `gSAVEBIT` erstellt wurden, bekommen automatisch die UID der Applikation in ihren Header geschrieben.

Bei Binär- und Text-Datei-Dokumenten, die mit `IOOPEN` und `LOPEN` erzeugt wurden, muss der Programmierer selbst dafür sorgen, dass ein entsprechender Header geschrieben wird. Das ist ziemlich einfach, das Beispiel zeigt es:

1. Erzeugen Sie ein Datenbank- oder Bitmap-Dokument mit Hilfe von SETDOC wie oben beschrieben.
2. Benutzen Sie einen Hexeditor, um sich die ersten sechzehn Bytes der Datei anzusehen oder lassen Sie untenstehendes Programm laufen, um die vier UIDs im Long-Integer-Format anzeigen zu lassen.
3. Schreiben Sie diese vier Long-Integers an den Anfang der Datei, die Sie mit IOOPEN erzeugen.

```
INCLUDE "Const.opb"
PROC Main:
    LOCAL f$(255)
    WHILE 1
        dINIT "UIDs im Dokument-Header"
        dPOSITION 1,0
        dFILE f$,"Dokument,Ordner,Laufwerk",0
        IF DIALOG=0
            RETURN
        ENDIF
        ReadUids:(f$)
    ENDWH
ENDP

PROC ReadUids:(file$)
    LOCAL ret%,h%
    LOCAL uid$(4),i%
    ret%=IOOPEN(h%,file$, KIoOpenModeOpen% OR KIoOpenFormatBinary%)
```

```

IF ret%>=0
  ret%=IOREAD(h%,ADDR(uid&()),16)
  PRINT "Reading ";file$
  IF ret%=16
    WHILE i%<4
      i%=i%+1
      PRINT "  Uid"+num$(i%,1)+"=",hex$(uid&(i%))
    ENDWH
  ELSE
    PRINT "  Fehler: ";
    IF ret%<0
      PRINT err$(ret%)
    ELSE
      PRINT "Liest ";ret%;" Bytes statt 4 Long-Int's"
    ENDIF
  ENDIF
  IOCLOSE(h%)
ELSE
  PRINT "Fehler beim Öffnen: ";ERR$(ret%)
ENDIF
ENDP

```

Das Anlegen von Text-Dokumenten mit IOOPEN oder LOPEN hat zwei spezielle Anforderungen:

- Der erforderliche Header muss ein echter Texteintrag sein, d.h. nach den UIDs müssen die Befehle für "Wagenrücklauf" und "Zeilenfortschaltung" (CR – Carriage Return, LF – Linefeed, in Hex : 0D 0A) folgen, um eine Zeilenbegrenzung zu erzeugen.
- Die spezifischen ersten 16 Bytes mit den vier UIDs können unter Umständen selber die Kombination "0D 0A" enthalten. Sie müssen darauf beim Auslesen Rücksicht nehmen. Wenn die Umstände es erfordern, lassen Sie sich lieber eine neue UID geben, anstatt einen Lade-Fehler zu provozieren.

Sie auch GETDOC\$

## SETFLAGS

F - OPL32

SETFLAGS flags&

Setzt Flags, anhand derer bestimmte Effekte für laufende Programme erreicht werden. Mit CLEARFLAGS werden alle gesetzten Flags wieder gelöscht.

Folgende Effekte können erreicht werden:

flags&	Effekt
1	begrenzt den zur Verfügung stehenden Speicherraum für die Applikation auf 64 KB und emuliert somit einen Serie 3xx. Dieses Flag sollte einmalig am Anfang des Programmes gesetzt werden, mehrmaliges Ändern des Wertes hat unvorhersagbare Effekte zur Folge.
2	veranlasst die automatische Komprimierung von Datenbank-Dateien, wenn sie geschlossen werden. Das kann zwar das Programm verlangsamen, ist aber trotzdem empfehlenswert, wenn üblicherweise viele Änderungen am Datenbankinhalt vorgenommen werden.
4	veranlasst – ebenfalls aus Kompatibilitätsgründen mit dem Serie 3xx – einen Overflow-Fehler, wenn Fließkommazahlen den Wert von 1.0E+100 annehmen oder überschreiten. OPL32 rechnet an dieser Stelle mit dreistelligem Exponenten weiter.
\$10000	versetzt GETEVENT, GETEVENT32 und GETEVENTA32 in die Lage, den Event-Code \$403 an ev&(1) zurückzuliefern, wenn die Maschine einschaltet

Standardmäßig sind die Flags nicht gesetzt.

Siehe auch GETEVENT32, CLEARFLAGS

**SETNAME***SIBO*`SETNAME name$`

Setzt den Namen der laufenden Applikation auf name\$. Der Name wird z.B. im Statusfenster unter dem Programmsymbol angezeigt.

Unter OPL32 erfüllt SETDOC eine ähnliche Funktion.

**SETPATH***OPL32 – SIBO*`SETPATH name$`

Wählt das Verzeichnis, das als Standardverzeichnis für Dateizugriff verwendet wird.

Beispiel :

```
SETPATH "C:\DOKUMENTE"
```

LOADM wird weiterhin das Programmverzeichnis für Aufrufe von Modulen verwenden, alle anderen Befehle wie OPEN etc. greifen jedoch in Zukunft auf das angegebene Verzeichnis zu.

**SIN***OPL32 – SIBO*`y=SIN(x)`

Gibt den Sinus-Wert von x zurück (x im Bogenmaß angeben!). Zum Wandeln von Grad in Bogenmaß benutze man die RAD-Funktion.

**SPACE***OPL32 – SIBO*`s&=SPACE`

Liefert die Anzahl der freien Bytes des Laufwerks auf dem sich die aktuell geöffnete Datei befindet.

Beispiel (für SIBO):

```
PROC Boerse:
  OPEN "A:boerse",A,a$,b%
  WHILE 1
    PRINT "Aktienname:";
    INPUT A.a$
    PRINT "Nummer:";
    INPUT A.b%
    IF RECSIZE>SPACE
      PRINT "Disk ist voll"
      CLOSE
      BREAK
    ELSE
      APPEND
    ENDIF
  ENDWH
ENDP
```

**SQR***OPL32 – SIBO*`y=SQR(x)`

Zieht die Quadratwurzel aus x.

**STATUSWIN**

SIBO

```
STATUSWIN ON, type%
STATUSWIN ON
STATUSWIN OFF
```

Zeigt ein permanentes Statusfenster an oder entfernt es wieder. Für type%=1 wird ein kleines, für type%=2 ein großes Statusfenster angezeigt. STATUSWIN ON allein zeigt immer ein passendes Statusfenster an, auf dem Serie 3xx ist das immer ein großes Statusfenster.

Das Statusfenster wird immer hinter allen anderen Fenstern angezeigt, um es sehen zu können müssen Sie FONT (oder SCREEN und gSETWIN) benutzen, um die Größe der Text- und Grafikfenster zu verändern. Es liegt in Ihrer Verantwortung, dass keine Fenster das Statusfenster verdecken.

OPL32 benutzt anstelle des Statusfensters den Toolbar!

**STATWININFO**

SIBO

```
t%=STATWININFO (typ%,xy%())
```

Liefert in xy%(1), xy%(2), xy%(3) und xy%(4) die dem Statusfenster mit Typ typ% entsprechenden Werte für x/y-Koordinaten der linken oberen Ecke und Breite und Höhe des Statusfensters. typ%=1 steht für das kleine, 2 für das große, 3 für das Serie 3-kompatible und -1 für das gerade aktive Statusfenster. In typ% wird der Typ des aktuellen Statusfensters zurückgeliefert oder 0, wenn kein Statusfenster aktiv ist. OPL32 benutzt anstelle des Statusfensters den Toolbar!

**STD**

OPL32 – SIBO

```
y=STD(liste)
y=STD(array(), element)
```

Berechnet die (empirische) Standardabweichung von einer Werte- bzw. Ausdrucksliste. Die Werteliste ist entweder eine kommagetrennte Liste:

```
y=STD(22, 2*34, 180/PI, 17, 4)
```

oder ein Array von Fließkommazahlen:

```
LOCAL wert(20), n%
REM Werte in wert() setzen nicht vergessen
n%=10
y=STD(wert(), n%)
```

Das erste Argument ist der Fließkomma-Array-Name, danach folgt mit n% die Angabe der Elemente, die ab dem ersten Element mit in die Berechnung einbezogen werden sollen.

Die zugrunde liegende Funktion lautet:

$$y = \text{SQR}(\sum (x_i - \bar{x})^2 / (n-1)) \text{ mit } \bar{x} = \sum x_i / n$$

Um die Varianz zu erhalten, müssen Sie y noch mit  $\text{SQR}((n-1) / n)$  multiplizieren.

**STOP**

OPL32 – SIBO

```
STOP
```

Beendet das laufende Programm.

**STYLE**

OPL32 – SIBO

`STYLE stil%`

Setzt im Textfenster den Fontstil. Es sind nur zwei Werte für `stil%` möglich: 2 (unterstrichen) und 4 (invers).

**SUM**

OPL32 – SIBO

```
y=SUM(liste)
y=SUM(array(),elemente)
```

Liefert den Summen-Wert einer Werte-Liste. Die Werteliste ist entweder eine kommagetrennte Liste:

```
y=SUM(22, 2*34, 180/PI,17,4)
```

oder ein Array von Fließkommazahlen:

```
LOCAL wert(20), n%
REM Werte in wert() setzen nicht vergessen
n%=10
y=SUM(wert(),n%)
```

Das erste Argument ist der Fließkomma-Array-Name, danach folgt mit `n%` die Angabe der Elemente, die ab dem ersten Element mit in die Bewertung einbezogen werden sollen.

**TAN**

OPL32 – SIBO

`y=TAN(x)`

Gibt den Tangens-Wert von `x` zurück (`x` im Bogenmaß angeben!). Zum Wandeln von Grad in Bogenmaß benutze man die `RAD`-Funktion.

**TESTEVENT**

OPL32 – SIBO

`t%=TESTEVENT`

Liefert "WAHR" wenn ein Ereignis aufgetreten ist, sonst "FALSCH". Das Ereignis selbst wird nicht ausgelesen. Hierfür können Sie `GETEVENT`, `GETEVENT32` oder `GETEVENTA32` verwenden, wenn `TESTEVENT` erfolgreich war.

Achtung: Für OPL32 wird empfohlen, nur `GETEVENT32` oder `GETEVENTA32` ohne `TESTEVENT` zu benutzen, weil `TESTEVENT` insbesondere in Schleifen sehr viel Batterie-Strom zieht.

**TRAP**

OPL32 – SIBO

`TRAP befehl`

`TRAP` verhindert, dass durch den in dieser Zeile gegebenen Befehl das Programm abbricht, wenn ein Fehler auftritt. Will man z.B. mit `gCLOSE win%` ein nicht geöffnetes Fenster `win%` schließen, tritt ein Fehler auf, der zu einer Fehlermeldung mit gleichzeitigem Programmabbruch führt. Ein `TRAP gCLOSE win%` verhindert das zwar, aber die Fehlernummer wird trotzdem in `ERR` abgelegt und kann dann selbst verarbeitet werden. Das Programm wird im Fehlerfall in der Zeile nach `TRAP ...` fortgeführt.

`TRAP ...` schaltet ein aktives `ONERR` für die Dauer des `geTRAP`ten Befehls aus.

`TRAP` arbeitet mit diesen OPL-Befehlen zusammen:

Datenbank-Befehle: APPEND, UPDATE, BACK, NEXT, LAST, FIRST, POSITION, USE, CREATE, OPEN, OPENR, CLOSE, DELETE, MODIFY, INSERT, PUT, CANCEL

Dateibefehle: COPY, ERASE, RENAME, LOPEN, LCLOSE, LOADM, UNLOADM, COMPRESS

Verzeichnisbefehle: MKDIR, RMDIR

Dateingabebefehle: EDIT, INPUT

Grafikbefehle: gSAVEBIT, gCLOSE, gUSE, gUNLOADFONT, gFONT, gPATT, gCOPY

Nützliches Beispiele:

```
TRAP INPUT wert% REM Eingabe ohne Werteeingabe mit
                  REM <Enter> oder <Esc> beendbar
TRAP EDIT string$ REM Eingabe mit <Esc> beendbar
```

### TRAP EDIT

OPL32 – SIBO

Wird die Taste ESC gedrückt, während a\$ leer ist, wird der Fehler "ESC gedrückt" (-114) von ERR ausgegeben, wenn Sie EDIT mit TRAP verwendet haben. Sie können so den Abbruch von EDIT über ESC implementieren.

Siehe auch INPUT, EDIT

### TRAP INPUT

OPL32 – SIBO

Wenn Sie TRAP verwenden, liefert INPUT nur einen Fehler zurück, gibt die Kontrolle aber zur nächsten Zeile des Programms weiter. Sie können also den Fehler selbst behandeln. Fehler treten auf, wenn eine falsche Eingabe (z.B. String statt Zahl) gemacht wurde oder im leeren Eingabefeld ESC gedrückt wurde (Fehler -114, "ESC gedrückt"). Die Fehler werden von ERR zurückgeliefert.

Siehe auch EDIT

### TRAP RAISE

OPL32 – SIBO

```
TRAP RAISE fehler%
```

Der Wert von ERR wird auf fehler% gesetzt, besonders sinnvoll um ERR zu löschen (auf Null zu setzen).

### TYPE

SIBO

```
TYPE num%
```

Legt über num% den Applikationstyp fest (0-4). Auf dem Serie 3xx sollten Sie jeweils \$1000 zum Typ addieren, Sie können so ein grau/schwarzes Programmsymbol mit einer Auflösung von 48x48 Pixel benutzen.

Der Befehl kann nur zwischen APP und ENDA eingesetzt werden.

**UADD***OPL32 – SIBO*

```
i%=UADD (wert1%,wert2%)
```

Addiert wert1% und wert2% als vorzeichenfreier Integer-Werte. Der Befehl dient zum Beispiel zur Addition von Adressen und verhindert dort einen Integer-Überlauf bei Wertebereichsüberschreitung. Sie sollten also zum Beispiel `UADD (ADDR (text$) , 1)` statt `ADDR (text$) +1` schreiben.

Ein Argument wird meist eine Adresse, das andere ein Offset sein.

UADD sollte nicht unter OPL32 benutzt werden, wenn nicht mit `SETFLAG` das Flag für die Beschränkung des Speichers auf 64 KB gesetzt wurde.

Siehe auch `USUB`

**UNLOADLIB***F - SIBO*

```
ret%=UNLOADLIB (cat%)
```

Entfernt eine DYLIB aus dem Speicher und liefert bei Erfolg 0 zurück.

**UNLOADM***OPL32 – SIBO*

```
UNLOADM modul$
```

Entfernt das Modul modul\$ aus dem Speicher. modul\$ ist dabei der Name des übersetzten Moduls. Ist das Modul entfernt, besteht keine Zugriffsmöglichkeit mehr auf die Prozeduren des Moduls.

Siehe auch `LOADM`

**UNTIL***OPL32 – SIBO*

Siehe `DO...UNTIL`

**UPDATE***OPL32 – SIBO*

```
UPDATE
```

Löscht den aktuellen Datensatz der aktuellen Datei und speichert die aktuellen Feldinhalte als neuen Datensatz am Ende der Datei. Der neu gespeicherte Datensatz ist nun aktuell.

Beispiel:

```
A.zahl=129
A.name="Schmidt"
UPDATE
```

Unter OPL32 verwende man `MODIFY`, `PUT`, `CANCEL`.

Siehe auch: `APPEND`

**UPPER\$***OPL32 – SIBO*

```
grossezeichen$=UPPER$(string$)
```

Wandelt Kleinbuchstaben in Großbuchstaben.

Siehe auch: `LOWER$`



**USE**

OPL32 – SIBO

USE logName

Macht eine geöffnete Ansicht zur aktuellen - alle Datenbankbefehle beziehen sich danach auf diese Ansicht. Dabei ist logName der logische Name, der beim Öffnen der Ansicht verwendet wurde.

**USESPRITE**

F - SIBO

USESPRITE id%

Macht das Sprite mit der ID-Nummer id% zum aktuellen Sprite

Unter OPL32 werden Sprites über das eingebaute Sprite-OPX/DLL-Modul verwaltet.

**USR**

F - SIBO

u%=USR(pc%,ax%,bx%,cx%,dx%)

Führt ein von Ihnen geschriebenes Maschinenprogramm aus. Das Programm muß mit einem far RET die Kontrolle an das rufende Programm zurückgeben, sonst erfolgt ein Programmabsturz.

Die Werte von ax%, bx%... werden in die AX,BX... Register übergeben. Der Prozessor führt dann ab Adresse pc% das Maschinenprogramm aus. Am Ende der Prozedur wird der Wert des AX-Registers in u% zurückgeliefert.

Vorsicht: Die Prozedur kann bei falscher Benutzung zu Systemabsturz und Datenverlust führen.

Das folgend Beispielprogramm zeigt eine einfache Operation, die mit einem far RET endet.

```
PROC trivial:
  LOCAL t%(2),u%,ax%
  t%(1)=$c032 REM xor al,al
  t%(2)=$cb REM retf
  ax%=$lab
  u%=usr(addr(t%(1)),ax%,0,0,0)
  REM Übergibt (ax% und $FF00)
  PRINT u% REM 256 ($100)
  GET
ENDP
```

Siehe auch USR\$, ADDR, PEEK, POKE

**USR\$**

F - SIBO

u\$=USR\$(pc%,ax%,bx%,cx%,dx%)

Führt wie USR ein Maschinenprogramm aus, liefert jedoch einen String zurück. Das Programm muss mit einem far RET die Kontrolle an das rufende Programm zurückgeben, sonst erfolgt ein Programmabsturz.

Die Werte von ax%, bx%... werden in die AX,BX... Register übergeben. Der Prozessor führt dann ab Adresse pc% das Maschinenprogramm aus. Am Ende der Prozedur enthält ax% die Adresse des Strings u\$.

Vorsicht: Die Prozedur kann bei falscher Benutzung zu Systemabsturz und Datenverlust führen.

Siehe auch USR, ADDR, PEEK, POKE

**USUB***OPL32 – SIBO*

```
i%=USUB(wert1%, wert2%)
```

Zieht wert2% von wert1% ab. Die Funktion behandelt die Argumente als vorzeichenfreie Integers und kann so ohne Integer-Überlauf zur Subtraktion von Adressen eingesetzt werden, da sie den Wertebereich auf 0-65535 erweitert.

USUB sollte nicht unter OPL32 benutzt werden, wenn nicht mit SETFLAG das Flag für die Beschränkung des Speichers auf 64 KB gesetzt wurde.

Siehe auch UADD

**VAL***OPL32 – SIBO*

```
zahl=VAL(numstring$)
```

Übergibt die Zahl, die in numstring\$ enthalten ist, an zahl als Fließkommawert. Dabei darf numstring\$ kein Ausdruck sein.

```
zahl= VAL("123.45")
PRINT zahl*2
REM Ergebnis: 246,9
```

Siehe auch: EVAL

**VAR***OPL32 – SIBO*

```
y=VAR(list)
y=VAR(array(),element)
```

Liefert die (empirische) Varianz einer Werte-Liste. Die Werteliste ist entweder eine kommagetrennte Liste:

```
y=VAR(22, 2*34, 180/PI,17,4)
```

oder ein Array von Fließkommazahlen:

```
LOCAL wert(20), n%
REM Werte in wert() setzen nicht vergessen
n%=10
y=VAR(wert(),n%)
```

Das erste Argument ist der Fließkomma-Array-Name, danach folgt mit n% die Angabe der Elemente, die ab dem ersten Element mit in die Bewertung einbezogen werden sollen.

Die zugrundeliegende Formel lautet:

$$y = \sum (x_i - \bar{x})^2 / (n-1) \text{ mit } \bar{x} = \sum x_i / n$$

Um die Varianz, die bei der Wahrscheinlichkeitsrechnung benutzt wird, zu erhalten, müssen Sie y noch mit (n-1)/n multiplizieren.

**VECTOR***OPL32 – SIBO*

```
VECTOR zeiger%
marke_1,marke_2,...,marke_n
ENDV
```

Die Zeile `marke_1 ...` stellt zwischen `VECTOR` und `ENDV` eine kommagetrennte Liste mit Sprungmarkennamen bereit, die eben diesen Namen tragen und zu dem das Programm verzweigen kann. Zu welcher Marke das Programm wirklich springt, wird durch die Variable `zeiger%` bestimmt, denn sie enthält die Position des Sprungmarkennamens in der Liste. Das Programm erfährt so, zu welcher Marke es springen soll. Ist `zeiger%=2`, springt das Programm zu der Marke `marke_2`.

Die Sprungmarkenliste darf mehrzeilig sein, die Zeilen dürfen jedoch kein Komma am Ende haben.

```
VECTOR zeiger%
    marke_1,marke_2,marke_3
    marke_4, marke_n
ENDV

marke_1::
    RETURN "Text1"
marke_2::
    c%=a%*b%
    RETURN
marke_n::
```

Das Programm setzt jeweils in der Zeile nach der angesprungenen Marke fort und stoppt auch nicht an den folgenden Marken, es sei denn, man sorgt selbst mit `RETURN` dafür.

Ist `zeiger%` kleiner oder größer als die Anzahl der in der Liste befindlichen Sprungziele, setzt das Programm unmittelbar hinter `ENDV` fort.

Diese Konstruktion fördert (vergleichbar `GOTO`) die strukturierte Programmierung nicht und wird deshalb nicht unbedingt zur Benutzung empfohlen, hat aber dennoch gelegentliche Vorteile. Ähnliche Wirkung könnte man mit einer `IF..ELSEIF..ENDIF`-Konstruktion erzielen, die aber weitaus langsamer arbeitet.

## WEEK

*OPL32 – SIBO*

```
w%=WEEK(tag%,monat%,jahr%)
```

Liefert die dem angegebenen Datum entsprechende Kalenderwoche (1-53). Die Werte für `tag%`, `monat%` und `Jahr%` müssen ein gültiges Datum ergeben (1-31, 1-12, 1900-2155).

Der Wochenbeginn ist durch die entsprechende Systemeinstellung in der Applikation "Uhr" festgelegt. Beginnt ein Jahr an einem anderen Tag, wird die Woche als Woche 1 gezählt, wenn 4 Tage bis zum nächsten Wochenbeginn übrig sind.

## WHILE...ENDWH

*OPL32 – SIBO*

```
WHILE bedingung
...
ENDWH
```

Mit `WHILE` beginnt eine Schleife, deren Programm sich bis `ENDWH` erstreckt. Die Schleife wird solange durchlaufen, wie die hinter `WHILE` formulierte Bedingung WAHR ist. Im Gegensatz zur `DO..UNTIL`-Schleife wird bei entsprechender Bedingung die Schleife überhaupt nicht erst betreten. ). Ist die Bedingung nicht mehr erfüllt wird die erste Zeile nach `ENDWH` ausgeführt.

Siehe auch `DO..UNTIL`

## YEAR

*OPL32 – SIBO*

```
j%=YEAR
```

Liefert das Jahr des Systemdatums als Zahlenwert zwischen 1900 und 2155 zurück.

## Anhang 1, Zeichentabelle & Codes

---

Jeder Computer stellt Text mit Hilfe eines Zeichensatzes dar. Dabei handelt es sich um eine Tabelle, in der jedem Zeichen eine Nummer, der Zeichencode, zugeordnet ist. Die Tabelle enthält 256 Zeichen, jedoch lassen sich die ersten 32 (Nr. 0 .. 31) nicht drucken - es handelt sich um sogenannte Steuerzeichen. Alle Zeichen ab Nr. 32 lassen sich über die Tastatur erzeugen. Die Zeichen 0 bis 127 stellen den historisch entstandenen ASCII-Code dar, der Rest ist im Laufe der Entwicklung dazugekommen.

### Die Text-Zeichen

Die untenstehenden Tabellen listen den Zeichensatz. Die linke Spalte enthält den Zeichencode in dezimaler Schreibweise, die mittlere den selben Wert in Hexadezimaldarstellung. In der rechten Spalte sehen Sie das zugehörige Zeichen oder eine Bemerkung, wenn sich das Zeichen nicht darstellen lässt. Dabei stehen "nbs" für no-break space (geschütztes Leerzeichen) und "----" für unbenutzte Positionen.

dez	hex	CHR\$	dez	hex	CHR\$	dez	hex	CHR\$
32	20	Leerz.	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	\$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	löschen

dez	hex	CHR\$	dez	hex	CHR\$	dez	hex	CHR\$
128	80	€	171	AB	«	214	D6	ö
129	81	---	172	AC	¬	215	D7	×
130	82	,	173	AD	-	216	D8	ø
131	83	f	174	AE	@	217	D9	ù
132	84	"	175	AF	-	218	DA	ú
133	85	...	176	B0	°	219	DB	û
134	86	†	177	B1	±	220	DC	ü
135	87	‡	178	B2	²	221	DD	ý
136	88	›	179	B3	³	222	DE	ÿ
137	89	‰	180	B4	´	223	DF	ß
138	8A	š	181	B5	µ	224	E0	à
139	8B	‹	182	B6	¶	225	E1	á
140	8C	œ	183	B7	·	226	E2	â
141	8D	---	184	B8	¸	227	E3	ã
142	8E	ž	185	B9	¸	228	E4	ä
143	8F	---	186	BA	°	229	E5	å
144	90	---	187	BB	»	230	E6	æ
145	91	¸	188	BC	¼	231	E7	ç
146	92	¸	189	BD	½	232	E8	è
147	93	¸	190	BE	¾	233	E9	é
148	94	¸	191	BF	¿	234	EA	ê
149	95	•	192	C0	À	235	EB	ë
150	96	-	193	C1	Á	236	EC	ì
151	97	-	194	C2	Â	237	ED	í
152	98	-	195	C3	Ã	238	EE	î
153	99	™	196	C4	Ä	239	EF	ï
154	9A	š	197	C5	Å	240	FO	ð
155	9B	›	198	C6	Æ	241	F1	ñ
156	9C	œ	199	C7	Ç	242	F2	ò
157	9D	---	200	C8	È	243	F3	ó
158	9E	ž	201	C9	É	244	F4	ô
159	9F	ÿ	202	CA	Ê	245	F5	õ
160	A0	ns	203	CB	Ë	246	F6	ö
161	A1	ı	204	CC	Ì	247	F7	÷
162	A2	ç	205	CD	Í	248	F8	ø
163	A3	£	206	CE	Î	249	F9	ù
164	A4	¤	207	CF	Ï	250	FA	ú
165	A5	¥	208	D0	Ð	251	FB	û
166	A6	¦	209	D1	Ñ	252	FC	ü
167	A7	§	210	D2	Ò	253	FD	ý
168	A8		211	D3	Ó	254	FE	þ
169	A9	©	212	D4	Ô	255	FF	ÿ
170	AA	ª	213	D5	Õ			

## Die Steuerzeichen

dez	hex	Bedeutung
00	00	null
01	01	start of heading
02	02	start of text
03	03	end of text
04	04	end of transmission
05	05	enquiry
06	06	acknowledge
07	07	bell ("Klingel", Piepser)
08	08	backspace (rückwärts löschen)
09	09	horizontal tabulation (Tabulator)
10	0A	line feed (Zeilenvorschub)
11	0B	vertical tabulation
12	0C	form feed (Blattvorschub, Bildschirm löschen)
13	0D	carriage return (Return, Cursor an Zeilenanfang)
14	0E	shift out
15	0F	shift in
16	10	data link escape
17	11	device control one
18	12	device control two
19	13	device control 3
20	14	device control four
21	15	negative acknowledge
22	16	synchronous idle
23	17	end of transmission block
24	18	cancel
25	19	end of medium
26	1A	substitute
27	1B	escape
28	1C	file separator
29	1D	group separator
30	1E	record separator
31	1F	unit separator

Die fett gedruckten Codes können in Zusammenhang mit `PRINT` und `CHR$( )` benutzt werden, um die genannten Effekte hervorzurufen. So z.B. erzeugt `PRINT CHR$(7)` einen kurzen Piepser.

## Spezialtasten

Die oben aufgeführten Zeichencodes sind gleichzeitig genau die Werte , die mit KEY oder GET abgefragt werden können. Die Codes einiger Spezialtasten findet man jedoch nicht in dieser Tabelle. Zusammen mit einigen bekannten Codes sind sie hier gelistet:

dez	hex	Taste
08	08	Entfernen
09	09	TAB
13	0D	ENTER
27	1B	ESC
256	100	PFEILTASTE AUF
257	101	PFEILTASTE AB
258	102	PFEILTASTE RECHTS
259	103	PFEILTASTE LINKS
260	104	BILD AUF
261	105	BILD AB
262	106	POS1
263	107	ENDE
290	122	MENÜ

Werden die Tasten über GETEVENT32 abgefragt, benutze man diese Werte:

dez	hex	Taste
4098	1002	POS1
4099	1003	ENDE
4100	1004	BILD AUF
4101	1005	BILD AB
4103	1007	PFEILTASTE LINKS
4104	1008	PFEILTASTE RECHTS
4105	1009	PFEILTASTE AUF
4106	100A	PFEILTASTE AB
4150	1036	MENÜ
10000	2710	MENÜ via linke Werkzeugleiste
10001	2711	COPY/CUT/PASTE via linke Werkzeugleiste
10002	2712	INFRAROT via linke Werkzeugleiste
10003	2713	ZOOM IN (+) via linke Werkzeugleiste
10004	2714	ZOOM OUT (-) via linke Werkzeugleiste

## Sonderfall

Wenn man Buchstabentasten zusammen mit der gehaltenen STRG-Taste drückt, ergibt sich eine Besonderheit: es wird nicht der Zeichencode zurückgeliefert, sondern die Ordnungszahl des Buchstabens im Alphabet. Beispiel:

"A": Zeichencode = 65, Ordnungszahl = 1,

"B": Zeichencode = 66, Ordnungszahl = 2, usw.

Dummerweise ist die Ordnungszahl für "a" ebenfalls 1. Das ergibt sich aus der Vorschrift, nach der die Ordnungszahl intern ermittelt wird:

```
ASC("A") AND (NOT $60) --> 1
ASC("a") AND (NOT $60) --> 1
```

Wenn Sie die zusammen mit STRG gedrückten Tasten abfragen, z.B. um einen Menü-Shortcut zu realisieren, müssen Sie diesen Umstand berücksichtigen. Testen Sie also mit `KMOD`, ob gleichzeitig auch die Shift-Taste gedrückt wurde! Näheres siehe im Kapitel "Menüs".



## Anhang 2, Binär- und Hex-Zahlen

---

### Dekadisches System

In den Grundlagenkapiteln haben Sie sehen können, dass es gelegentlich praktischer ist, Parameter oder Argumente als hexadezimale (hex-) Zahlen zu übergeben. Sie müssen kein Fachmann für Zahlensysteme werden, aber einen kleinen Überblick möchte ich Ihnen schon geben ...

Sehen wir zuerst auf etwas, das Ihnen vertraut ist – unser dezimales Zahlensystem. Es arbeitet mit 10 Ziffern (0,1, .. 9), die erst durch ihre Stellung in der Zahl ihren wahren Wert erhalten. Zu jeder Stelle (Einer, Zehner, Hunderter, ...) gehört ein Multiplikationsfaktor, der sich von der Basiszahl 10 ableitet. Aufgesplittet bedeutet z.B. 3468:

$$\begin{aligned} 8 * 10^0 &= 8 \\ 6 * 10^1 &= 60 \\ 4 * 10^2 &= 400 \\ 3 * 10^3 &= 3000 \end{aligned}$$

Der Exponent ist also für die Wertigkeit verantwortlich. Das ist bei den anderen Zahlensystemen nicht anders – nur die Zahlenbasis ändert sich.

### Binäres System

Sie haben es hier mit lediglich zwei Ziffern zu tun: Null und Eins. Die Zahlenbasis ist also 2.

Beispiel: 1011 ergibt nach Aufschlüsselung von rechts:

$$\begin{aligned} 1 * 2^0 &= 1 \\ 1 * 2^1 &= 2 \\ 0 * 2^2 &= 0 \\ 1 * 2^3 &= 8 \end{aligned}$$

Summe: 11

In diesem System lassen sich nur Ganzzahlen darstellen. Gerechnet wird wie mit Dezimalzahlen:

	1011	(dez: 8 + 0 + 2 + 1 = 11)
+	1101	(dez: 8 + 4 + 0 + 1 = 13)
(Übertrag	111- )	
Summe	11000	(dez: 16 + 8 + 0 + 0 + 0 = 24)

Die bitweisen Operatoren AND und OR arbeiten wie folgt (s. Kapitel "Verzweigungen"):

	1010
AND	1100
=	1000
	1010
OR	1100
=	1110

Um aus Dezimalzahlen Binärzahlen zu erzeugen, muss man sie Stück um Stück prüfen, welche 2er Potenzen in ihnen stecken. Man beginnt mit der größtmöglichen. Im Beispiel soll die 13 gewandelt werden:

13:      8 ( $=1*2^3$ ) enthalten? → ja    → 4. bin. Stelle = 1  
 13-8=5: 4 ( $=1*2^2$ ) enthalten? → ja    → 3. bin. Stelle = 1  
 5-4=1:  2 ( $=1*2^1$ ) enthalten? → nein → 2. bin. Stelle = 0  
 1:      1 ( $=1*2^0$ ) enthalten? → ja    → 1. bin. Stelle = 1  
 Ergebnis: dez. 13 ist binär: 1101

## Hex-Zahlen

Die hexadezimalen Zahlen rechnen – der Name verrät es bereits – mit 16 Ziffern. Sie müssen neben den Fingern also noch Schuhe und Strümpfe ausziehen, um die Zehen zur Hilfen nehmen zu können ;-)

Die ersten zehn Ziffern sind uns vertraut: 0,1, .. 9. Danach geht es mit Buchstaben weiter:

A = 10  
 B = 11  
 C = 12  
 D = 13  
 E = 14  
 F = 15

Über die 15 hinaus wird die zweite Stelle erforderlich. Die dezimale 16 ist also eine hexadezimale 10.

Hex-Zahlen werden in OPL mit einem Vorsatz gekennzeichnet. Wenn normale Integers als Hex-Zahlen dargestellt werden sollen, wird das Dollarzeichen (\$) verwendet. Für Long-Integers kommt das Kaufmanns-Und (&) zum Einsatz. Beispiele:

Normal-Integer: \$023E (führende Null muss nicht mit angegeben werden):

\$E \*  $16^0$  =    14  
 \$3 \*  $16^1$  =    48  
 \$2 \*  $16^2$  =   512  
 \$0 \*  $16^3$  =     0

dez. Summe:   574

Long-Integer: &0001AF27 023E (führende Nullen müssen nicht mit angegeben werden):

\$7 \*  $16^0$  =       7  
 \$2 \*  $16^1$  =      32  
 \$F \*  $16^2$  =    3840  
 \$A \*  $16^3$  =   40960  
 \$1 \*  $16^4$  =   65536

dez. Summe: 110375

Zur Umwandlung von Dezimalzahlen verwendet man den HEX\$-Befehl, z.B.:

```
PRINT HEX$(110375)
```

Das Ergebnis ist: "1AF27", also die Stringdarstellung der Zahl, das Vorsatzzeichen wird nicht mit dargestellt. Achtung: es können nur Ganzzahlen durch Hex-Zahlen dargestellt werden. Verwendet man als Argument eine Fließkommazahlen, wird erst eine Integer-Zahl gebildet, dann die Umwandlung vorgenommen!

Zwischen Hex-Zahlen und Binär-Zahlen bestehen gewissermaßen verwandtschaftliche Verhältnisse. Eine Hex-Ziffer kann eine vierstellige Binärzahl ersetzen:

$$\text{\$F(hex)} = 1111(\text{bin}) = 15(\text{dez})$$

Das bedeutet, ein Byte nimmt in seinen 8 Bits eine 8stellige Binärzahl auf, die sich wiederum durch eine übersichtliche zweistellige Hex-Zahl darstellen lässt. Der Maximalwert, der sich mit einem Byte darstellen lässt, ist 255(dez):

$$255(\text{dez}) = 11111111(\text{bin}) = \text{\$FF(hex)}$$

## Anhang 3, Fontsliste

---

Die Liste der Fonts stammt aus der Datei CONST.OPH, die jeder Psion unter EPOC32 im ROM mit an Bord hat. Sie enthält alle offiziell von Symbian angelegten Konstantendefinitionen. Die Font-Konstanten, die mit dem Befehl gFONT verwendet werden können (Kapitel "Grafischer Text"), sind also nur ein Auszug aus dieser Liste. Der Name der Fonts mit der Angabe der Pixelgröße ist aus dem Namen der Konstanten zu entnehmen.

Font-Konstante	dez. Wert
KFontArialBold8&=	268435951
KFontArialBold11&=	268435952
KFontArialBold13&=	268435953
KFontArialNormal8&=	268435954
KFontArialNormal11&=	268435955
KFontArialNormal13&=	268435956
KFontArialNormal15&=	268435957
KFontArialNormal18&=	268435958
KFontArialNormal22&=	268435959
KFontArialNormal27&=	268435960
KFontArialNormal32&=	268435961
KFontTimesBold8&=	268435962
KFontTimesBold11&=	268435963
KFontTimesBold13&=	268435964
KFontTimesNormal8&=	268435965
KFontTimesNormal11&=	268435966
KFontTimesNormal13&=	268435967
KFontTimesNormal15&=	268435968
KFontTimesNormal18&=	268435969
KFontTimesNormal22&=	268435970
KFontTimesNormal27&=	268435971
KFontTimesNormal32&=	268435972

Font-Konstante	dez. Wert
KFontCourierBold8&=	268436062
KFontCourierBold11&=	268436063
KFontCourierBold13&=	268436064
KFontCourierNormal8&=	268436065
KFontCourierNormal11&=	268436066
KFontCourierNormal13&=	268436067
KFontCourierNormal15&=	268436068
KFontCourierNormal18&=	268436069
KFontCourierNormal22&=	268436070
KFontCourierNormal27&=	268436071
KFontCourierNormal32&=	268436072
KFontCalc13n&=	268435493
KFontCalc18n&=	268435494
KFontCalc24n&=	268435495
KFontMon18n&=	268435497
KFontMon18b&=	268435498
KFontMon9n&=	268435499
KFontMon9b&=	268435500
KFontTiny1&=	268435501
KFontTiny2&=	268435502
KFontTiny3&=	268435503
KFontTiny4&=	268435504
KFontEiksym15&=	268435661
KFontSquashed&=	268435701
KFontDigital35&=	268435752

**Tipp:**

An die Datei CONST.OPH und weitere interessante OPL-Demo-Dateien kommen Sie, wenn Sie im OPL-Editor das Menü "Extras/Standarddateien erstellen" anwählen. Die Dateien werden dann aus dem ROM in das aktuelle Verzeichnis kopiert und sind damit zugänglich.

## Anhang 4, OPL-Fehlernummern

---

Diese Fehlernummern werden von OPL verwendet:

### **Allgemein**

<u>Fehler nummer</u>	<u>Beschreibung</u>
-1	Allgemeiner Fehler
-2	Unerlaubter Parameter
-3	Systemfehler
-4	Service nicht unterstützt
-5	Unterlauf (Zahl zu klein)
-6	Überlauf (Zahl zu groß)
-7	Bereichsüberschreitung
-8	Nulldivision
-9	Belegt (z.B. serieller Port bereits durch andere Task belegt)
-10	Kein Speicher
-11	Segmenttabelle voll
-12	Zeichentabelle voll
-13	Zu viele Tasks
-14	Quelle bereits geöffnet
-15	Quelle nicht geöffnet
-16	Unerlaubte Imagedatei
-17	Kein Empfänger
-18	Gerätetabelle voll
-19	Dateisystem nicht gefunden (z.B. wenn die PC-Verbindung getrennt ist)
-20	Start unmöglich
-21	Font nicht geladen
-22	Zu breit (Dialogbox)
-23	Zu viele Einträge (Dialogbox)
-24	Batterien zu schwach für Wiedergabe
-25	Batterien zu schwach zum Schreiben des Flash

**Datei- und Gerätefehler**

<u>Fehler nummer</u>	<u>Beschreibung</u>
-32	Datei existiert bereits
-33	Datei existiert nicht
-34	Schreibfehler
-35	Lesefehler
-36	Dateiende (bei dem Versuch, über ein Dateiende hinaus zu lesen)
-37	Disk voll
-38	Unerlaubter Name
-39	Zugriff verweigert (z.B. auf eine geschützte Datei)
-40	Datei in Gebrauch (wird bereits benutzt)
-41	Gerät existiert nicht
-42	Verz. existiert nicht
-43	Eintrag zu lang
-44	Datei nur lesbar
-45	Falscher I/O Aufruf
-46	Lfd. I/O Operation
-47	Ungültiges Laufwerk (z.B. SSD fehlerhaft oder nicht formatiert)
-48	I/O abgebrochen
-50	Unterbrochen
-51	Verbunden
-52	Zu viele Versuche
-53	Verbindungsfehler
-54	Timeout
-55	Falsche Parität
-56	Serielle Einstellung (Baudrate wahrsch. falsch)
-57	Serieller Überlauf (Handshake wahrsch. falsch)
-58	Keine Verbindung (zu externem Modem)
-59	Ext. Modem belegt
-60	Ext. Modem: Keine Antwort
-61	Anschluß gesperrt (nur eine begrenzte Zahl von Eingabeversuchen möglich; Eingabe nach kurzer Wartezeit wiederholen)

**Datei- und Gerätefehler (Fortsetzung)**

<u>Fehler nummer</u>	<u>Beschreibung</u>
-62	Nicht bereit
-63	SSD unbekannt (muß evtl. neu formatiert werden)
-64	Hauptverzeichnis voll (auf einer Einheit nur begrenzter Speicher)
-65	Schreibgeschützt
-66	Datei fehlerhaft
-67	Benutzerabbruch
-68	Löschfehler
-69	Fehler im Dateityp

**Meldungen vom Übersetzer**

<u>Fehler nummer</u>	<u>Beschreibung</u>
-70	Fehlt: "
-71	Zeichenkette zu lang
-72	Unerwarteter Name
-73	Name zu lang
-74	Logische Einheit muss A-Z sein
-75	Falscher Feldname
-76	Falsche Zahl
-77	Syntaxfehler
-78	Falsches Zeichen
-79	Argumentfehler
-80	Artendiskrepanz
-81	Marke fehlt
-82	Doppelter Name
-83	Deklarationsfehler
-84	Fehler Bereichsgröße
-85	Strukturfehler



**Meldungen vom Übersetzer (Fortsetzung)**

<u>Fehler nummer</u>	<u>Beschreibung</u>
-86	Fehlt: ENDP
-87	Syntaxfehler
-88	Diskrepanz: "(" oder ")"
-89	Feldabweichung
-90	Zu komplex
-91	Fehlt: ,
-92	Variablen zu groß
-93	Falsche Zuweisung
-94	Falscher Feldindex
-95	Ungleiche Argumentanzahl

**OPL-Fehler**

<u>Fehler nummer</u>	<u>Beschreibung</u>
-96	Fehlerhafter Objektcode (neu übersetzen!)
-97	Falsche Argumentanzahl
-98	Unbekannte Variable
-99	Prozedur nicht gefunden
-100	Feld nicht gefunden
-101	Datei bereits geöffnet
-102	Datei nicht geöffnet
-103	Datensatz ist für OPL zu groß
-104	Modul bereits geladen
-105	Modulmaximum erreicht
-106	Modul existiert nicht
-107	Inkompatible Übersetzerversion
-108	Modul nicht geladen
-109	Fehler im Dateityp
-110	Datentypdiskrepanz

**OPL-Fehler (Fortsetzung)**

<u>Fehler</u>	
<u>nummer</u>	<u>Beschreibung</u>
-111	Index oder Dimensionsfehler (bei Feldvariablen)
-112	String zu lang (Zeichenk. länger als definiert)
-113	Einheit bereits geöffnet
-114	Esc-Taste wurde gedrückt
-115	Inkompatibles Laufzeitumgebung
-116	ODB Datei(en) nicht geschlossen
-117	Zuviele Zeichenflächen
-118	Zeichenfläche nicht geöffnet
-119	Ungültiges Fenster
-120	Kein Bildschirmzugriff
-121	OPX-Datei existiert nicht
-122	Inkompatible OPX-Version
-123	OPX-Prozedur nicht gefunden
-124	STOP im callback verwendet
-125	Inkompatibler Update-Modus
-126	In Transaktion
-127	INCLUDE-Datei darf keine Prozeduren enthalten
-128	Zu viele OPX geöffnet
-129	Zu viele OPX-Funktionen
-130	Nicht definierte Variable
-131	Nicht definierte Prozedur
-132	Icon ohne Maske
-133	Inkompatible Deklaration

# Index

---

## A

ABS · 15  
 ACOS · 15  
 ADDR · 15  
 ADJUSTALLOC · 15  
 ALERT · 15  
 ALLOC · 16  
 APP ... ENDA · 16  
 APPEND · 16  
 APPENDSPRITE · 17  
 ASC · 17  
 ASIN · 17  
 AT · 17  
 ATAN · 18

---

## B

BACK · 18  
 BEEP · 18  
 BEGINTRANS · 18  
 BOOKMARK · 19  
 BREAK · 19  
 BUSY · 19  
 BYREF · 20

---

## C

CACHE · 20  
 CACHEHDR · 20  
 CACHEREC · 20  
 CACHETIDY · 20  
 CALL · 21  
 CAPTION · 21  
 CHANGESPRITE · 21  
 CHR · 22  
 CLEARFLAGS · 22  
 CLOSE · 22  
 CLOSESPRITE · 22  
 CLS · 22  
 CMD\$ · 22  
 COMMITTRANS · 23  
 COMPACT · 23  
 COMPRESS · 23  
 CONST · 24  
 CONTINUE · 24  
 COPY · 24

COS · 24  
 COUNT · 25  
 CREATE · 25  
 CREATESPRITE · 26  
 CURSOR · 26

---

## D

DATETOSECS · 26  
 DATIM\$ · 27  
 DAY · 27  
 DAYNAME\$ · 27  
 DAYS · 27  
 DAYSTODATE · 28  
 dBUTTONS · 28  
 dCHECKBOX · 29  
 dCHOICE · 29  
 dDATE · 29  
 DECLARE EXTERNAL · 30  
 DECLARE OPX · 30  
 dEDIT · 30  
 dEDITMULTI · 31  
 DEFAULTTWIN · 32  
 DEG · 33  
 DELETE · 33  
 dFILE · 33, 34  
 dFLOAT · 35  
 DIALOG · 35  
 DIAMINIT · 36  
 DIAMPOS · 36  
 dINIT · 36, 37  
 DIR\$ · 37  
 dLONG · 37  
 DO...UNTIL · 38  
 DOW · 38  
 dPOSITION · 38  
 DRAWSPRITE · 38  
 dTEXT · 38, 39  
 dTIME · 40  
 dXINPUT · 40

---

## E

EDIT · 40  
 ELSE/ELSEIF/ENDIF · 41  
 ENDV · 41  
 ENDWH · 41  
 ENTERSEND · 41

ENTERSEND0 · 41  
 EOF · 41  
 ERASE · 42  
 ERR · 42  
 ERR\$ · 42  
 ERRX\$ · 42  
 ESCAPE OFF, ESC ON · 42  
 EVAL · 43  
 EXIST · 43  
 EXP · 43  
 EXT · 43  
 EXTERNAL · 44

---

**F**

FIND · 44  
 FINDFIELD · 45  
 FINDLIB · 45  
 FIRST · 45  
 FIX\$ · 45  
 FLAGS · 46  
 FLT · 46  
 FONT · 46  
 FREEALLOC · 47

---

**G**

gAT · 47  
 gBORDER · 47  
 gBOX · 47  
 gBUTTON · 47, 48  
 gCIRCLE · 48  
 gCLOCK · 49, 51  
 gCLOSE · 53  
 gCLS · 53  
 gCOLOR · 53  
 gCOPY · 54  
 gCREATE · 55, 56  
 gCREATEBIT · 56  
 gDRAWOBJECT · 56  
 gELLIPSE · 57  
 GEN\$ · 57  
 GET · 57  
 GET\$ · 57  
 GETCMD · 58  
 GETDOC\$ · 58  
 GETEVENT · 58  
 GETEVENT32 · 59  
 GETEVENTA32 · 60  
 GETEVENTC · 60  
 GETLIBH · 60  
 gFILL · 60  
 gFONT · 61, 62

gGMODE · 62  
 gGREY · 62  
 gHEIGHT · 63  
 gIDENTITY · 63  
 gINFO · 63  
 gINFO32 · 64  
 gINVERT · 65  
 GIPRINT · 65  
 gLINEBY · 65  
 gLINETO · 65  
 gLOADBIT · 66  
 gLOADFONT · 66  
 GLOBAL · 67  
 gMOVE · 67  
 gORDER · 67  
 gORIGINX · 68  
 gORIGINY · 68  
 GOTO · 68  
 GOTOMARK · 68  
 gPATT · 69  
 gPEEKLINE · 69, 70  
 gPOLY · 70  
 gPRINT · 70  
 gPRINTB · 71  
 gPRINTCLIP · 71  
 gRANK · 71  
 gSAVEBIT · 71  
 gSCROLL · 72  
 gSETPENWIDTH · 72  
 gSETWIN · 72  
 gSTYLE · 72  
 gTMODE · 72  
 gTWIDTH · 73  
 gUNLOADFONT · 73  
 gUPDATE · 73  
 gUSE · 73  
 gVISIBLE · 73  
 gWIDTH · 73  
 gX · 74  
 gXBORDER · 74  
 gXPRINT · 74  
 gY · 75

---

**H**

HEX\$ · 75  
 HOUR · 75

---

**I**

I/O Befehle · 78  
 IABS · 75  
 ICON · 75, 76

IF...ENDIF · 76  
 INCLUDE · 77  
 INPUT · 77  
 INSERT · 78  
 INT · 78  
 INTF · 78  
 INTRANS · 78  
 IOA · 78  
 IOC · 78  
 IOCANCEL · 78  
 IOCLOSE · 79  
 IOPEN · 79  
 IOREAD · 79  
 IOSEEK · 79  
 IOSIGNAL · 79  
 IOW · 79  
 IOWAITSTAT · 79  
 IOWAITSTAT32 · 79  
 IOWRITE · 79

---

## **K**

KEY · 79, 80  
 KEYA · 80  
 KEYC · 80  
 KILLMARK · 80  
 KMOD · 80

---

## **L**

LAST · 80  
 LCLOSE · 81  
 LEFT · 81  
 LEN · 81  
 LENALLOC · 81  
 LINKLIB · 81  
 LN · 81  
 LOADLIB · 81  
 LOADM · 82  
 LOC · 82  
 LOCAL · 82  
 LOCK · 82  
 LOG · 83  
 LOPEN · 83  
 LOWER\$ · 83  
 LPRINT · 83

---

## **M**

MAX · 83  
 mCARD · 84, 85  
 mCASC · 85

MEAN · 85  
 MENU · 86  
 MID\$ · 86  
 MIN · 87  
 mINIT · 87  
 MINUTE · 87  
 MKDIR · 87  
 MODIFY · 88  
 MONTH · 88  
 mPOPUP · 88

---

## **N**

NEWOBJ · 88  
 NEWOBJH · 89  
 NEXT · 89  
 NUM\$ · 89

---

## **O**

ODBINFO · 89  
 OFF · 89  
 ONERR · 90  
 OPEN · 90  
 OPENR · 91  
 OS · 91

---

## **P**

PARSE\$ · 92  
 PATH · 92  
 PAUSE · 93  
 PEEK Funktionen · 93  
 PI · 94  
 POINTERFILTER · 94  
 POKE Befehle · 94  
 POS · 95  
 POSITION · 95  
 POSSPRITE · 95  
 PRINT · 96  
 PUT · 96

---

## **R**

RAD · 96  
 RAISE · 96  
 RANDOMIZE · 97  
 REALLOC · 97  
 RECSIZE · 97  
 REM · 97  
 RENAME · 98  
 REPT · 98

RETURN · 98  
RIGHT\$ · 99  
RMDIR · 99  
RND · 99  
ROLLBACK · 100

---

**S**

SCI\$ · 100  
SCREEN · 100  
SCREENINFO · 100  
SECOND · 101  
SECSTODATE · 101  
SEND · 101  
SETDOC · 102  
SETFLAGS · 103  
SETNAME · 104  
SETPATH · 104  
SIN · 104  
SPACE · 104  
SQR · 104  
STATUSWIN · 105  
STATWININFO · 105  
STD · 105  
STOP · 105  
STYLE · 106  
SUM · 106

---

**T**

TAN · 106  
TESTEVENT · 106  
TRAP · 106  
TRAP EDIT · 107

TRAP INPUT · 107  
TRAP RAISE · 107  
TYPE · 107

---

**U**

UADD · 108  
UNLOADLIB · 108  
UNLOADM · 108  
UNTIL · 108  
UPDATE · 108  
UPPER\$ · 108  
USE · 109  
USESPRITE · 109  
USR · 109  
USR\$ · 109  
USUB · 110

---

**V**

VAL · 110  
VAR · 110  
VECTOR · 110

---

**W**

WEEK · 111  
WHILE...ENDWH · 111

---

**Y**

YEAR · 111